Principles of Algorithmic Problem Solving

Johan Sannemo

2025

Principles of Algorithmic Problem Solving, 1st edition
ISBN 978-91-519-9041-5

Published by Adversarial AB, Solna

Contents

Introduction

Algorithmic problem solving is the art of formulating efficient methods that solve problems of a mathematical nature. From the many numerical algorithms developed by the ancient Babylonians to the founding of graph theory by Euler, algorithmic problem solving has been a popular intellectual pursuit during the last few thousand years. For a long time, it was a purely mathematical endeavor with algorithms meant to be executed by hand. During the recent decades algorithmic problem solving has evolved. What was mainly a topic of research became a mind sport known as competitive programming. As a sport algorithmic problem solving rose in popularity, with the largest competitions attracting tens of thousands of programmers. While its mathematical counterpart has a rich literature, there are only a few books on algorithms with a strong problem solving focus.

The purpose of this book is to contribute to the literature of algorithmic problem solving in two ways. First of all, it tries to fill in some holes in existing books. Many topics in algorithmic problem solving lack any treatment at all in the literature – at least in English books. Much of the content is instead documented only in blog posts and solutions to problems from various competitions. While this book attempts to rectify this, it is not to detract from those sources. Many of the best treatments of an algorithmic topic I have seen are as part of a well-written solution to a problem. However, there is value in completeness and coherence when treating such a large area. Secondly, I hope to provide another way of learning the basics of algorithmic problem solving by helping the reader build an intuition for problem solving. A large part of this book describes techniques using worked-through examples of problems. These examples attempt not only to describe the manner in which a problem is solved, but also to show how a thought process might be guided to yield the insights necessary to arrive at a solution.

This book is different from pure programming books and most other algorithm textbooks. Programming books are mostly either in-depth studies of a specific programming language or describe various programming paradigms. A single language is used in this book – C++. The chapters on C++ exists for the sole purpose of enabling those readers without prior programming experience to implement the solutions to algorithm problems. Such a treatment is necessarily minimal and teaches neither good coding style nor advanced programming concepts. Algorithm textbooks teach primarily algorithm analysis, basic algorithm design, and some standard algorithms and data structures. They seldom include as much problem solving as this book does. The book also falls somewhere between the practical nature of a programming book and the heavy theory of algorithm textbooks. This is in part due to the book's dual nature of being not only about algorithmic problem solving, but also competitive programming to some extent. As such there is more

real code in the form of C++ implementations of algorithms included compared to most algorithm books.

Acknowledgments. First and foremost, thanks to Per Austrin who provided much valuable advice and feedback during the writing of this book. Thanks to Simon and Mårten who have competed with me for several years as Omogen Heap. A lot of the knowledge in this book has its roots in you. Thanks to the Kattis team for providing a great online system for all the exercises, especially Greg Hamerly who carefully reviewed the over one hundred exercises that was added to Kattis for this book. Finally, thanks to several others who have read through drafts and caught numerous of my mistakes.

Reading this Book

This book consists of three parts. The first part contains preliminary background, such as algorithm analysis, basic programming in C++ and introductions to data structures, recursion and graph theory. With an undergraduate education in computer science most of the content in these chapters is probably familiar to you. It is recommended that you at least skim through the first part since the remainder of the book assumes you know the contents of the preliminary chapters.

The second part deals with basic paradigms in problem solving. They give you important techniques and principles that are reused in many of the later topics as well. Some of it should be familiar if you have taken a course in algorithms and data structures. The take on those topics is a bit different compared to such a course, however. The chapters in this part are structured so that a chapter builds upon only the preliminaries and previous chapters to the largest extent possible.

In the third part, we study various topics that build upon these common techniques, many which are computational applications of different mathematical areas. While they are still organized to depend only on earlier chapters to the largest extent possible, there might sometimes be cross-references that are hard to avoid. You can to a larger degree choose what topics you wish to study, though you will benefit greatly from having read the first two parts.

The book ends with several appendices: tips and strategy for participating in algorithmic problem solving competitions, mathematical background for preuniversity students, and hints and solutions for selected exercises.

When reading this book, know that every problem and technique was chosen with care; every step on the way to a solution added to provide value. Sometimes, this can make the book feel boring – a solution can take a long time tracing out the intuition behind some small step, or show partial solutions that are unused in the final result. At other times, missing a single sentence can leave you with a crucial gap in your knowledge. I have tried to make sure that every sentence written is important. When the book is

long-winding, trust that it is useful, and when difficult, endure to make sure you attain the deep understanding I hope this book will be able to provide.

Similarly, the exercises are meant as attempts for you to construct some crucial knowledge on your own. There may be fewer end-of-chapter exercises than you might be used to in a textbook, and more exercises inlined in chapters. This is because we expect you to solve **all** inline exercises as part of the reading of the book. Sometimes, the text after an exercise will assume that you have read and solved the exercise. The lecture analogue would be the lecturer pausing to ask the class a question; only giving an answer if none is provided by the class. Since this is a book, you are blessed with unlimited time to think in contrast to the lecture setting, where you typically get on the order of minutes. Some exercises took the author a long time to solve at first, so do not feel disparaged when you find one difficult. At the back of the book, you can find hints and solutions for selected exercises. If you fail to solve an exercise, first check if it has a hint, and give it another attempt.

This book can also be used to improve your competitive programming skills. Some parts are unique to competitive programming (in particular Appendix A on contest strategy). This knowledge is extracted into *competitive tips*:

Competitive Tip

A competitive tip contains some information specific to competitive programming. These can be safely ignored if you are interested only in the problem solving aspect and not implementing solutions in code.

The book often refers to coding exercises that are available online, that you can submit implemented solutions to:

Problem 0.1.

Problem Name problemid

They are available mainly on the Kattis online judge, but sometimes on other online judges as well. The URL of such a problem is `heap.link/problem/problemid`. To make sure that links can be kept up-to-date, they are all linked through the book's web page rather than to judges directly. Certain problems are split up into different *subtasks* that include constraints to make the problem easier. When we give an exercise where you are meant only to do some subtasks, we state this along the exercise like so:

Problem 0.2.

Problem Name problemid *(subtasks 2, 3)*

Part I

Preliminaries

Chapter 1

Algorithms and Problems

The greatest technical invention of the last century was probably the digital general purpose computer. It was the start of the revolution that provided us with the Internet, smartphones, tablets, and the computerization of society.

To harness the power of computers, we use *programming*. Programming is the art of developing a solution to a *computational problem* in the form of a set of instructions that a computer can execute. These instructions are what we call *code*, and the language in which they are written a *programming language*. The abstract method that such code describes is what we call an *algorithm*.

The aim of *algorithmic problem solving* is thus to, given a computational problem, devise an algorithm that solves it. One does not necessarily need to complete the full programming process (i.e. write code that implements the algorithm in a programming language) to enjoy solving algorithmic problems. However, it often provides more insight and trains you at finding simpler algorithms to problems.

In this chapter, we begin our journey into algorithmic problem solving by taking a closer look at these concepts and showing a solution to a common problem.

1.1 Computational Problems

A ***computational problem*** generally consists of two parts. First, it needs an ***input description***, such as "a sequence of integers", "a text string", or some other kind of mathematical object. Using this input, we have a goal which we want to accomplish defined by an ***output description***. For example, a computational problem might require us to sort a given sequence of integers. This particular problem is called the *Sorting Problem*:

Sorting

Given a sequence of N integers $a_0, a_1, ..., a_{N-1}$, sort it in ascending order, i.e. from the lowest to the highest.

A particular input to a computational problem is called an ***instance*** of the problem. To the sorting problem, the sequence $3, 6, 1, -1, 2, 2$ is an instance. The correct output for this particular instance is $-1, 1, 2, 2, 3, 6$.

Exercise 1.1. If you were given cards with 5 different integers between 1 and 1 000 000 written on them, how would you sort them in ascending order? How would your approach change if you had 30 integers? 1000? 1 000 000?

Some variations of this problem format appears later (such as problems without inputs) but in general this is what the problems look like.

Competitive Tip

Problem statements sometimes contain huge amounts of text. Skimming through the input and output sections before any other text in a problem can often give you a quick idea about its topic and difficulty. This helps with determining what problems to solve first when posed with a large number of problems and little time.

Exercise 1.2. What are the input and output descriptions for the following computational problems?

1. Compute the greatest common divisor (see Def. 17.5, page 318 if you are not familiar with the concept) of two numbers.
2. Find a root (i.e. a zero) of a polynomial.
3. Multiply two numbers.

Exercise 1.3. Consider the following problem. I am thinking of an integer between 1 and 100. Your task is to find this number by giving me integers, one at a time. I will tell you whether the given integer is higher, lower or equal to x.

This is an *interactive*, or *online*, computational problem. How would you describe the input and output to it? Why do you think it is called interactive?

1.2 Algorithms

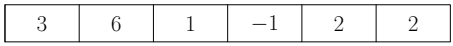
Algorithms are solutions to computational problems. They define methods that use the input to a problem in order to produce the correct output. A computational problem can have many solutions. Efficient algorithms to solve the sorting problem form an entire research area! Let us look at one possible sorting algorithm, called selection sort, as an example.

Selection Sort

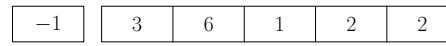
We construct the answer, the sorted sequence, iteratively one element at a time, starting with the smallest.

Assume that we have chosen and sorted the K smallest elements of the original sequence. Then, the smallest unchosen element remaining in that sequence must be the $(K+1)$'st smallest element of the original sequence. By finding that element and appending it to the already sorted K smallest elements we get the sorted $K+1$ smallest elements of the output.

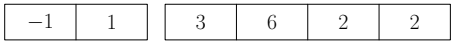
If we repeat this process N times, the result is the N numbers of the original sequence, but sorted. ■



(a) The starting sequence $(3, 6, 1, -1, 2, 2)$.

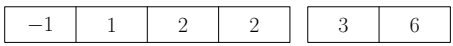


(b) The smallest element of the sequence is -1, so this is the first element of the sorted sequence.

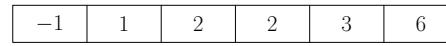


(c) The next element is found by removing the -1 and finding the smallest remaining element, 1.

(d) Here, there is no unique smallest element. We can choose any of the two 2's in this case.



(e) The next element is the other 2.



(f) The last two elements chosen are 3 followed by the 6, completing the sort.

Figure 1.1: An example execution of selection sort.

You can see this algorithm performed on our previous example instance in Figures 1.1a-1.1f.

So far, we have been vague about what exactly an algorithm is. Looking at our selection sort example, we do not have any particular structure or rigor in the description of our method. There is nothing inherently wrong with describing algorithms this way. It is easy to understand and gives the writer an opportunity to provide context as to why certain actions are performed, making the correctness of the algorithm more obvious. The main downsides of such a description are ambiguity and a lack of detail.

Until an algorithm is described in sufficient detail, it is possible to accidentally abstract away operations we may not know how to perform behind a few English words. As a somewhat contrived example, our plain text description of selection sort includes actions such as "choosing the smallest number of a sequence". While such an operation may seem very simple to us humans, algorithms are generally constructed with regards to some kind of computer. Unfortunately, computers can not map such English expressions to their code counterparts yet. Instructing a computer to execute an algorithm requires us to formulate our algorithm in steps small enough that even a computer knows how to perform them. In this sense, a computer is rather stupid.

The English language is also ambiguous. We are sloppy with references to "this variable" and "that set", relying on context to clarify meaning for us. We use confusing terminology and frequently misunderstand each other. Real code does not have this problem. It forces us to be specific with what we mean. However, as all programmers know, we often manage to construct highly specific algorithms that *do the wrong thing* due to our own erroneous thought processes.

In the book, we generally describe our algorithms in a representation called pseudo

code (Section 1.4), accompanied by an online exercise to implement the code. Sometimes, we instead give explicit code that solves a problem. This is the case whenever an algorithm is very complex, or care must be taken to make the implementation efficient. The goal is that you should get to practice understanding pseudo code, while still ending up with correct implementations of the algorithms (thus the online exercises).

Exercise 1.4. What common algorithms do you know, for example from school?

Exercise 1.5. Attempt to write down formally the descriptions of your approaches to sorting from Exercise 1.1.

Exercise 1.6. Construct an algorithm that solves the guessing problem in exercise 1.3 using as few questions as possible. How many questions does it use?

Correctness

One subtle, albeit important, point that we glossed over is what it means for an algorithm to actually be ***correct***.

There are two common notions of correctness – ***partial correctness*** and ***total correctness***. Partial correctness requires an algorithm to, upon termination, have produced an output that fulfills all the criteria laid out in the output description. Total correctness additionally requires an algorithm to finish within finite time. When we talk about correctness of our algorithms later on, we generally focus on the partial correctness. Termination is instead proved implicitly, as we consider a more granular measure of efficiency (called *time complexity*, in Chapter 5) than just finite termination. This measure implies the termination of the algorithm, completing the proof of total correctness.

Proving that the selection sort algorithm finishes in finite time is quite easy. It performs one iteration of the selection step for each element in the original sequence (which is finite). Furthermore, each such iteration can be performed in finite time by looking at each remaining element of the selection when finding the smallest one. The remaining sequence is a subsequence of the original one and is therefore also finite.

Proving that the algorithm produces the correct output is a bit more difficult to prove formally. The main idea behind a formal proof is contained within our description of the algorithm itself.

While this definition seems clear enough – our algorithm should simply do what the problem asks of it! – we occasionally compromise on both conditions at later points in the book. Generally, we are satisfied with an algorithm terminating in expected finite time or answering correctly with, say, probability 0.75 for every input. Similarly, we are sometimes happy to find an *approximate* solution to a problem. What this means more concretely becomes clear in due time when we study such algorithms.

Competitive Tip

Proving your algorithm correct is sometimes quite difficult. In a competition, a correct algorithm

is correct even if you cannot prove it. If you have an idea you *think* is correct it may be worth testing. This is not a strategy without problems though, since it makes distinguishing between an incorrect algorithm and an incorrect implementation even harder.

Exercise 1.7. Prove the correctness of your algorithm to the guessing problem from Exercise 1.6 and your sorting algorithms from Exercise 1.5.

Exercise 1.8. Why would an algorithm that is correct with e.g. probability 0.75 still be very useful to us?

Why is it important that such an algorithm is correct with probability 0.75 on *every* problem instance, instead of always being correct for 75% of all cases?

1.3 Programming Languages

The purpose of ***programming languages*** is to formulate methods at a level of detail where a computer could perform them. When describing algorithms to other people, we're mostly happy with describing *what* we want to do. Programming languages instead require considerably more constructive descriptions. Computers are quite basic creatures compared to us humans. They only understand a very limited set of instructions such as adding numbers, multiplying numbers, or moving data around within its memory. The syntax of programming languages often seems a bit arcane at first, but it grows on you with coding experience.

To complicate matters further, programming languages themselves define a spectrum of expressiveness. On the lowest level, programming deals with electrical current in your processor. Current above or below a certain threshold is used to represent the binary digits 0 and 1. Above these circuit-level electronics lies a processor's own programming, often called *microcode*. Using this, a processor implements *machine code*, such as the x86 instruction set. Machine code is often written using a higher-level syntax called *Assembly*. While some code is written in this rather low-level language, we mostly abstract away details of them in high-level languages such as C++ (this book's language of choice).

This knowledge is somewhat useless from a problem solving standpoint, but intimate knowledge of how a computer works is of high importance in software engineering, and is occasionally helpful in programming competitions. Therefore, you should not be surprised about certain remarks relating to these low-level concepts.

These facts also provide some motivation for why we use something called *compilers*. When programming in C++ we can not immediately tell a computer to run our code. As you now know, C++ is code at a higher level than what the processor of a computer can run. A compiler takes care of this problem by translating our C++ code into the machine code that the processor knows how to handle. It is a program of its own and takes the code files we write as input and produces *executable* files that we can run on the computer. The process and purpose of a compiler is somewhat like what we do ourselves when translating

a method from English sentences or our own thoughts into the lower level language of C++.

Other than C++ which the book uses, there are two other popular languages for competitive programming called Python and Java. They are slightly easier languages than C++, but tend to produce slower code.

1.4 Pseudo Code

Somewhere in between describing algorithms in English text and in a programming language we find something called ***pseudo code***. As hinted by its name it is not quite real code. The instructions we write are not part of the programming language of any particular computer. The point of pseudo code is to be independent of the computer it is implemented on. Instead, it tries to convey the main points of an algorithm in a detailed manner so that it can easily be translated into any particular programming language. We sometimes fall back to the liberties of the English language in pseudo code. As the book progresses, the pseudo code assumes more about what we can implement in code without a detailed description. For a completely new programmer, translating the simple "choose the smallest number in a sequence" to computer code is hard, while you in a few chapters will find "check if the graph is a tree" to be a perfectly good level of abstraction.

With an explanation of this distinction in hand, let us look at a concrete example of pseudo code. The honor of being an example again falls upon selection sort, now described in pseudo code:

```
1:  procedure SelectionSort(sequence A)
2:      Let A' be an empty sequence
3:      while A is not empty do
4:          minIndex ← 0
5:          for every element A_i in A do
6:              if A_i < A_minIndex then
7:                  minIndex ← i
8:          Append A_minIndex to A'
9:          Remove A_minIndex from A
10:     return the sequence A'
```

Pseudo code reads somewhat like our English language description of the algorithm, except the actions are broken down into much smaller pieces. Most of the constructs in pseudo code are more or less obvious. The values in parenthesis after the procedure name is the input to the procedure, while the "return" keyword tells us what value the procedure produces. The notation *variable* ← $value$ is how we denote an *assignment* in pseudo code. For those without programming experience, this means that the variable named *variable* now takes the value $value$. When referring to elements in lists, the indices start with 0.

Pseudo code appears in the book when we try to explain some part of a solution in

great detail but programming language specific aspects would draw attention away from the algorithm itself.

Competitive Tip

In team competitions where a team shares a single computer, they will often have solved problems waiting to be coded. Writing pseudo code of the solution to one of these problems while waiting for computer time is an efficient way to parallelize your work. This can be practiced by writing pseudo code on paper even when you are solving problems by yourself.

Exercise 1.9. Write pseudo code for your algorithm to the guessing problem from Exercise 1.6.

1.5 Online Judges

The problem exercises in this book are located on so called ***online judges***, primarily the Kattis online judge at `heap.link/judge:kattis`. They contain large collections of computational problems, and allow you to submit programs you have written to solve the problems. A judge runs your program on a large number of predetermined instances of the problem called the problem's ***test data***.

Problems on an online judge include some additional information compared to our example problem. Since actual computers only have a finite amount of time and memory, the amount of these resources available to our programs are limited when solving an instance of a problem. This also means that the size of inputs to a problem need to be constrained as well. A more complete version of the sorting problem as given in a competition would include these constraints along with exactly how input and output is specified. It could look something like this:

Sorting – sorting

Time: 1s, memory: 100 MB

Your task is to sort a sequence of integers in ascending order, i.e. from the lowest to the highest.

Input

The input is a sequence of N integers ($1 \leq N \leq 1000$) $a_0, a_1, ..., a_{N-1}$ ($|a_i| \leq 10^9$).

Output

Output a permutation a' of the sequence a, such that $a'_0 \leq a'_1 \leq ... \leq a'_{N-1}$.

For these problems, we typically give the problem ID in the title as well, accessible at `https://heap.link/problem/problemid`.

If your program exceeds the allowed resource limits (i.e. takes too much time or memory), crashes, or gives an invalid output, the judge will tell you so with a *rejected judgment*. There are many kinds of rejected judgments, such as *Wrong Answer*, *Time Limit Exceeded*, and *Run-time Error*. These mean that your program gave an incorrect output,

took too much time, and crashed, respectively. If your program passes all the instances, it will be be given the *Accepted* judgment.

Getting a program accepted by an online judge is not the same as having a correct program – it is a necessary but not sufficient criterion for correctness. This can sometimes be exploited during competitions by writing a knowingly incorrect solution that you think passes all test cases designed by the judges of the competition.

We strongly recommend that you get accounts on the online judges used in the problems so that you can follow along with the book's exercises.

Exercise 1.10. Register an account on the Kattis online judge.

Many other online judges exists, for example Codeforces (`heap.link/judge:codeforces`) and AtCoder (`heap.link/judge:atcoder`). Both are platforms which run regular contests and have large archives of problems.

Additional Exercises

Exercise 1.11. Pick two sorting algorithms from Wikipedia's list of sorting algorithms: `en.wikipedia.org/wiki/Category:Sorting_algorithms`. Try to understand them and their proof of correctness. Use them by hand to sort the integers $5, 1, 2, 7, 5, 6, 2, 9$.

Exercise 1.12. Consider the following problem:

Palindrome

A word is a palindrome if it reads the same forwards and backwards, for example `tacocat`, `madam`, or `abba`. Determine if a word is a palindrome.

---

1. Devise an algorithm to solve it,
2. formalize the algorithm and write it down in pseudo code, and
3. prove the correctness of the algorithm.

Notes

The introductions given in this chapter are very bare, mostly stripped down to what you need to get by when solving algorithmic problems.

Many other books delve deeper into the theoretical study of algorithms than we do, in particular regarding subjects not relevant to algorithmic problem solving. *Introduction to Algorithms* [11] is a rigorous introductory text book on algorithms with both depth and breadth.

For a gentle introduction to the technology that underlies computers, *CODE* [40] is a well-written journey from the basics of bits and bytes all the way up to assembly code and operating systems. It requires no knowledge of programming to read.

Chapter 2

Programming in C++

In this chapter we learn the basics of the C++ programming language. It is the most common programming language within the competitive programming community for a few reasons (aside from C++ being a popular language in general). Programs coded in C++ are generally somewhat faster than those written in most other competitive programming languages. There are also many routines in the accompanying standard code libraries that are useful when implementing algorithms.

Of course, no language is without downsides. C++ is difficult to learn as your first programming language, to say the least. Its error management is unforgiving, often causing erratic behavior in programs instead of crashing with an error. Programming certain things becomes quite verbose, compared to many other languages.

After bashing the difficulty of C++, you might ask if it really is the best language in order to get started with algorithmic problem solving. While there certainly are simpler languages we believe that the benefits outweigh the disadvantages in the long run even though it demands more from you as a reader right now. Either way, it is definitely the language we have the most experience of teaching problem solving with.

When you study this chapter, you will see a lot of example code. **Type this code and run it**. We can not really stress this point enough. Learning programming from scratch – in particular a complicated language such as C++ – is not possible unless you try the concepts yourself. We strongly recommend that you do **every** exercise in this chapter, even more so than in the other chapters.

Finally, know that our treatment of C++ is minimal. We do not explain all the details behind the language, nor do we teach good coding style or general software engineering principles. In fact, we frequently make use of bad coding practices. If you want to delve deeper into programming, you can find more resources in the chapter notes.

2.1 Hello World!

Before we start coding you need to install a compiler for C++ and (optionally) a code editor. We recommend the editor ***Visual Studio Code*** that you can download from `heap.link/ide:vscode`. Installation procedures for a C++ compiler for different operating systems tend to rot quickly and requires a lot of pasting commands, so we have placed them all online at `heap.link/ide:install` to try and keep them up-to-date as much as possible. If you are unable to install new programs on your computer, `heap.link/ide:online` contains

references to some online editors and compilers that you can use instead. They give you a bit less control over compilation and makes it harder to run programs using local files for input, things that can be beneficial when solving programming problems.

Exercise 2.1. Follow the above instructions to get a C++ compiler and editor ready.

Now that you have a compiler and editor ready, it is time to learn the basic structure of a C++ program. Our first example is a classical one when learning a new programming language: printing the text `Hello World!`. We also get to solve our first online judge problem in this section.

Exercise 2.2. You will learn many concepts within C++ throughout this chapter. Take written notes of how they are used as you go along to make your own cheat sheet as a future reference (you will need it).

Start by opening your editor and creating a new file. Save the file as `hello.cpp`. Make sure to save it somewhere you can find it.

Now, type the code from Snippet 2.1 into your editor.

Snippet 2.1: Hello World!

```
#include <iostream>

using namespace std;

int main() {
  // Print Hello World!
  cout << "Hello World!" << endl;
}
```

Compile and run the program using the online instructions at `heap.link/ide:compile`. The output window should now contain the text `Hello World!`. If it doesn't, you probably got either an error indicating that your compiler setup does not work, or that the compilation failed because you mistyped the program.

Coincidentally, there is an online judge problem whose output description dictates that your program should print the text `Hello World!`. How convenient. This is a great opportunity to get familiar with submitting solutions to judges.

Problem 2.3.
Hello World! hello

When you submit your solution, the judge grades it and gives you its judgment. If you typed everything correctly, the judge tells you that the solution is Accepted. Otherwise, you probably got Wrong Answer, meaning your program output the wrong text (and you mistyped the code).

Now that you have solved the problem, it is time to talk about the code you typed. The first line of code,

```
#include <iostream>
```

is used to include the `iostream` – *input and output stream* – file from the so-called *standard library* of C++. The standard library is a large collection of ready-to-use algorithms, data structures, and other routines which you can use when coding. For example, there are sorting routines in the C++ standard library, meaning you do not need to implement your own sorting algorithm when coding solutions.

Later on, we will see other useful examples of the standard library and include many more files. The `iostream` file contains routines for reading and writing data to your screen. Your program used code from this file when it printed `Hello World!` upon execution.

On some platforms, there is a special include file called `bits/stdc++.h`. This file includes the entire standard library. You can check if it is available on your platform by including it using

```
#include <bits/stdc++.h>
```

in the beginning of your code. If your program still compiles, you do not need to include anything else when using utilities from the standard library in Chapter 3.

The third line,

```
using namespace std;
```

tells the compiler that we want to use code from the **st**an**d**ard library. If we did not add it, we would have to specify this every time we used code from the standard library later in our program by prefixing what we use from the library with `std::` (for example `std::cout`).

The fifth line defines our ***main function***. When instructed to run our program, the computer starts looking at this point for code to execute. The first line of the main function is where the program starts to run, with further lines in the function executed sequentially. Later on we learn how to define and use additional functions as a way of structuring our code. Note that the code in a function – its *body* – must be enclosed by curly brackets. Without them, we wouldn't know which lines belonged to the function.

On line 6, we wrote a ***comment***:

```
// Print Hello World!
```

Comments are explanatory lines not executed by the computer. Their purpose is to explain what the code around them do and why. They start with two slashes `//` and continue until the end of the current line.

It is not until the seventh line that things start happening in the program. We use the standard library utility `cout` to print text to the screen. This is done by writing for example:

```
cout << "this is text you want to print. ";
cout << "you can " << "also print " << "multiple things. ";
cout << "to print a new line" << endl << "you print endl" << endl;
cout << "without any quotes" << endl;
```

Lines that do things in C++ are called ***statements***. **Note the semi colon** at the end of the line! Semi colons are used to specify the end of a statement. They are mandatory.

Exercise 2.4. Must the main function be named `main`? Try changing the name to something else and running your program.

Exercise 2.5. Play around with `cout` a bit, printing various things. For example, you can print a pretty haiku.

2.2 Variables and Types

When solving mathematical problems, it's often useful to introduce all kinds of names for known and unknown values. Math problems may be about school classes of N students, trains with a velocity v_{train} km/h, and candy prices of p_{candy} \$/kg. In mathematics, they are known as ***variables***.

This concept naturally translates into C++ but with a twist. In most programming languages, we first need to say what ***type*** a variable has. We do not bother with this in mathematics. There, we just write "let $x = 5$". In C++, we need to be a bit more verbose and write that "I want to introduce a variable x now. It is going to be an integer that initially has the value 5". Once we have decided what kind of value x will be (in this case integer) it will always be an integer. We cannot go ahead and say "oh, I've changed my mind. $x = 2.5$ now!" since 2.5 is of the wrong type (a decimal number rather than an integer).

Snippet 2.2: Variables

```
#include <iostream>
using namespace std;

int main() {
  int five = 5;
  cout << five << endl;
  int seven = 7;
  cout << seven << endl;
  five = seven + 2; // = 7 + 2 = 9
  cout << five << endl;
  seven = 0;
  cout << five << endl; // five is still 9
  cout << 5 << endl; // we print the integer 5 directly
}
```

Another major difference is that variables in C++ are not tied to a single value during their entire lifespans. Instead, we are able to modify the value that a variable has using what is called ***assignment***. Some languages do not permit this, preferring their variables to be ***immutable***.

In Snippet 2.2 we demonstrate how variables are used in C++. Type this program into your editor and run it. What is the output? What did you expect the output to be?

The *first* time we use a variable in C++ we must decide what kind of values it may contain. This is called ***declaring*** the variable of a certain type. For example the statement

```
int five = 5;
```

declares an integer variable `five` and assigns the value 5 to it. The **`int`** part is C++ for integer and is what we call a type. After the type, we write the name of the variable – in this case `five`. Finally, we may assign a value to the variable. Note that further use of the variable never includes the **`int`** part. We declare the type of a variable once and only once.

Later on in Snippet 2.2 we decide that 5 is a somewhat small value for a variable called `five`. We can change the value of a variable by using the *assignment operator* – the equality sign `=`. The assignment

```
five = seven + 2;
```

states that from now on the variable `five` should take the value given by the expression `seven + 2`. Since (at least for the moment) `seven` has the value 7 the expression evaluates to 7 + 2 = 9. Thus `five` will actually be 9, explaining the output we get from line 12.

On line 14 we change the value of the variable `seven`. Note that line 15 still prints the value of `five` as 9. Some people find this model of assignment confusing. We first performed the assignment `five = seven + 2;`, but the value of `five` did not change with the value of `seven`. This is mostly an unfortunate consequence of the choice of `=` as operator for assignment. One could think that "once an equality, always an equality" – that the value of `five` should always be the same as the value of `seven + 2`. **This is not the case**. An assignment copies into the variable on the left hand side whatever the value of the expression on the right hand side at a particular moment in time, nothing more.

The snippet also demonstrates how to print the value of a variable on the screen – we `cout` it the same way as with text. This also clarifies why text needs to be enquoted. Without quotes, the compiler can't distinguish between the text string `"hi"` and the variable `hi`.

Note that it is possible to **declare a variable without assigning a value to it**. When this is done, the variable may receive an *arbitrary* value instead. This is useful when you immediately assign a value input by the user to a variable (see the next Section 2.3).

Exercise 2.6. What values will the variables a, b, and c have after executing the following code:

```
int a = 4;
int b = 2;
int c = 7;
b = a + c;
c = b - 2;
a = a + a;
b = b * 2;
c = c - c;
```

Here, the operator - denotes subtraction and `*` represents multiplication. Once you have arrived at an answer, type this code into the main function of a new program and print the values of the variables. Did you get it right?

Exercise 2.7. What happens when an integer is divided by another integer? Try printing the result of the following divisions: $\frac{5}{3}$, $\frac{15}{5}$, $\frac{2}{2}$, $\frac{7}{2}$, $\frac{-7}{2}$, and $\frac{7}{-2}$.

Exercise 2.8. C++ allows declarations of immutable (constant) variables, using the keyword **`const`**. For example

```
const int FIVE = 5;
```

What happens if you try to perform an assignment to such a variable?

There are many other types than **`int`**. We have seen one (although without its correct name), the type for text. You can see some of the most common types in Snippet 2.3.

Snippet 2.3: Types

```
1  string text = "Johan said: \"heya!\" ";
2  cout << text << endl;
3
4  char letter = '@';
5  cout << letter << endl;
6
7  int number = 7;
8  cout << number << endl;
9
10 long long largeNumber = 8888888888888;
11 cout << largeNumber << endl;
12
13 double decimalNumber = 513.23;
14 cout << decimalNumber << endl;
15
16 bool thisisfalse = false;
17 bool thisistrue = true;
18 cout << thisistrue << " and " << thisisfalse << endl;
```

The text data type is called `string`. Values of this type must be enclosed with double quotes. To include double quotation marks in strings, they must be *escaped* using a backslash: `"a \"quote\" in a string"`.

There exists a data type containing one single letter, the **`char`**. Such a value is surrounded by single quotes. The **`char`** value containing the single quotation mark is written `'\''`, similarly to how we included double quotes in strings.

Then comes the **`int`**, which we discussed earlier. The **`long long`** type contains integers just like the **`int`** type. They differ in how large integers they can contain. An **`int`** can only contain integers between -2^{31} and $2^{31} - 1$ while a **`long long`** extends this range to -2^{63} to $2^{63} - 1$.

Exercise 2.9. Since backslashes are used to escape quotes inside a string, we can not include backslashes in a string like any other character. Find out how to include a literal backslash in a string (for example by searching the web or thinking about how we included the different quotation mark).

Exercise 2.10. Write a program that assigns the minimum and maximum values of an **`int`** to am **`int`** variable `x`. What happens if you increment or decrement this value using `x = x + 1;` or `x = x - 1;` respectively and print its new value?

Competitive Tip

One of the most common sources for errors in code is trying to store an integer value outside the range of the type. Always make sure your values fit inside the range of an **`int`** if you use it. Otherwise, use **`long long`**s!

One of the reasons for not always using **`long long`** all the time is that some operations involving **`long long`**s can be slower using **`int`**s under certain conditions. The cost of representing many more integers it that **`long long`**s take twice the amount of memory that **`int`**s do.

Next comes the **`double`** type. This type represents decimal numbers. The decimal sign in C++ is a **dot**, not a comma. There is also another similar type called the **`float`**. The difference between these types are similar to that of the **`int`** and **`long long`**. A **`double`** can represent "more" decimal numbers than a **`float`**. This may sound weird considering that there is an infinite number of decimal numbers even between 0 and 1. However, a computer can clearly not represent every decimal number – not even all those between 0 and 1. Distinguishing between an infinite number of decimal numbers would require infinite memory. Instead, they represent a limited set of numbers – with about 15 significant digits, and about 308 zeroes to the left or right of those digits. Floats have fewer significant digits, and can only represent smaller numbers (they are seldom used in competitive programming).

The last of our common types is the **`bool`** (short for ***boolean***). This type can only contain one of two values – it is either `true` or `false`. While this may look useless at a first glance, the importance of the boolean becomes apparent later.

Exercise 2.11. In the same way that integer types have a valid range of values, a **`double`** cannot represent arbitrarily large values. Find out what the minimum and maximum values of a **`double`** are.

C++ has a construct called the ***typedef***, short for *type definition*. It enables us to give types new names. Since typing **`long long`** for every large integer variable is very annoying, we could use a type definition to alias it with the much shorter `ll` instead. Such a typedef statement looks like this:

```
typedef long long ll;
```

On every line after this statement, we can use `ll` just as if it were a **`long long`**:

```
ll largeNumber = 888888888888;
```

Sometimes we use types with very long names but do not want to shorten them using type definitions. This could be the case when we use many different such types and typedefing them would take unnecessarily long time. We then resort to using the **`auto`** "type" instead. If a variable is declared as **`auto`** and assigned a value at the same time its type is inferred from that of the value. This means we could write

```
auto val = 123;
```

instead of

```
int val = 123;
```

2.3 Input and Output

In the previous sections we occasionally printed things onto our screen. To spice our code up a bit we are now going to learn how to do the reverse – reading values that we type on our keyboards into a running program! When we run a program we may type things in the command terminal that appears. Pressing the Enter key allows the program to read what we have written so far.

Reading input data is done just as you would expect, almost entirely symmetric to printing output. Instead of `cout` we use `cin`, and instead of `<< variable` we use `>> variable`, i.e.

```
cin >> variable;
```

Type in the program from Snippet 2.4 to see how it works.

Snippet 2.4: Input

```
#include <iostream>
using namespace std;

int main() {
  string name;
  cout << "What's your first name?" << endl;
  cin >> name;
  int age;
  cout << "How old are you?" << endl;
  cin >> age;
  cout << "Hi, " << name << "!" << endl;
  cout << "You are " << age << " years old." << endl;
}
```

Exercise 2.12. What happens if you enter an invalid input, such as your first name instead of your age?

Exercise 2.13. What happens if you enter two words when asked for your first name?

We revisit more advanced input and output concepts in Section 3.5 about the standard library. For example, we learn how to read entire lines of text and not only single words.

Problem 2.14.
Odd Echo oddecho *(subtask 1)*

2.4 Operators

Earlier we saw examples of so-called ***operators***, such as the assignment operator `=`, and the arithmetic operators `+ - * /`, which stand for addition, subtraction, multiplication and division. They work almost like they do in mathematics, and allow us to write code such as the one in Snippet 2.5.

Snippet 2.5: Operators

```
#include <iostream>
using namespace std;

int main() {
  int a = 0;
  int b = 0;
  cin >> a >> b;
  cout << "Sum: " << (a + b) << endl;
  cout << "Difference: " << (a - b) << endl;
  cout << "Product: " << (a * b) << endl;
  cout << "Quotient: " << (a / b) << endl;
  cout << "Remainder: " << (a % b) << endl;
}
```

Exercise 2.15. Type in Snippet 2.5 and test it on a few different values. Most importantly, test:

- $b = 0$,
- negative values for `a` and/or `b`, and
- values where the expected result is outside the valid range of an **`int`**.

As you probably noticed in the exercise, the division operator of C++ performs so-called *integer division*. This means the answer is rounded to an integer (towards 0). Hence `7 / 3 = 2`, with remainder 1, and `-7 / 3 = -2`.

Exercise 2.16. If division rounds down towards zero, how do you compute $\frac{x}{y}$ rounded to an integer *away from* zero?

The snippet also introduces the ***modulo*** operator, `%`. It computes the remainder of the first operand when divided by the second. As an example, `7 % 3 = 1`. Different programming languages have different behaviors regarding modulo operations on negative integers. In C+C++ the value of a modulo operation can be negative when including negative operands.

In case we want the result of a division to be a decimal number one of the operands must be a **`double`** (Snippet 2.6).

Snippet 2.6: Division Operators

```
int a = 6;
int b = 4;
cout << (a / b) << endl;

double aa = 6.0;
double bb = 4.0;
cout << (aa / bb) << endl;
```

Some common operations have additional shorthand operators. Check out Snippet 2.7 for some examples. Each arithmetic operator has a corresponding combined assignment operator. Such an operator, e.g. `a += 5;` is equivalent to `a = a + 5;`. They act as if the variable on the left hand side is also the left hand side of the corresponding arithmetic operator and assign the result of this computation to said variable. Hence, the above statement increases the variable `a` with 5.

Snippet 2.7: Shorthand Operators

```
int num = 0;
num += 1;
cout << num << endl;
num *= 2;
cout << num << endl;
num -= 3;
cout << num << endl;
cout << num++ << endl;
cout << num << endl;
cout << ++num << endl;
cout << num << endl;
cout << num-- << endl;
cout << num << endl;
```

Addition and subtraction with 1 are fairly common operations. So common, in fact, that shorthand operators were introduced for the purpose of saving an entire character compared to the highly verbose `+=1` operator. These operators consist of two plus signs or two minus signs. For instance, `a++` increments the variable by 1.

These expressions also evaluate to a value. What value this is depends on whether we put the operator before or after the variable name. By putting `++` before the variable, the value of the expression is the incremented value. If we put it afterwards we get the original value. To get a better understanding of how this works it is best if you type the code in Snippet 2.7 in yourself and analyze the results.

We end the discussion on operators by saying something about *operator precedence*, i.e. the order in which operators are evaluated in expressions. In mathematics, there is a well-defined precedence: brackets go first, then exponents, followed by division, multiplication, addition, and subtraction. Furthermore, most operations (exponents being a notable exception) have left-to-right associativity so that $5 - 3 - 1$ equals $((5 - 3) - 1) = 1$ rather

than $(5 - (3 - 1)) = 3$. In C++, there are a lot of operators, and knowing precedence rules can easily save you from bugs in your future code.

Exercise 2.17. Research online C++ documentation on operator precedence to determine what the expression

```
2 * 4 - 7 * 2 % 4 / 2
```

evaluates to in C++. Run it as a program to see if you got it correct.

Problem 2.18.

Two-sum	twosum
Triangle Area	triarea
Bijele	bijele
Digit Swap	digitswap
R2	r2

2.5 If Statements

In addition to assignment and arithmetic, there are a large number of ***comparison operators***. They compare two values and evaluate to a **`bool`** value depending on the result of the comparison.

Snippet 2.8: Comparison Operators

```
1 a == b // check if a equals b
2 a != b // check if a and b are different
3 a > b // check if a is greater than b
4 a < b // check if a is less than b
5 a <= b // check if a is less than or equal to b
6 a >= b // check if a is greater than or equal to b
```

A **`bool`** can also be ***negated*** using the `!` operator. So the expression `!false` (which we read as "not false") has the value `true` and vice versa `!true` evaluates to `false`. The operator works on any boolean expressions, so that if `b` would be a boolean variable with the value `true`, then the expression `!b` evaluates to `false`.

There are two more important boolean operators. The ***and*** operator `&&` takes two boolean values and evaluates to `true` if and only if both values are `true`. Similarly, the ***or*** operator `||` evaluates to `true` if and only if at least one of its operands are `true`.

Exercise 2.19. Write a program that reads two integers as input, and prints the result of the different comparison operators from Snippet 2.8, e.g

```
cout << (a == b) << endl;
```

Note the parenthesis used due to operator precedence!

A major use of boolean variables is in conjunction with ***if*** statements (also called ***conditional*** statements). They come from the necessity of executing certain lines of code **if** (and only if) some condition is true. Let us write a program that takes an integer as input, and tells us whether it is odd or even. We can do this by computing the remainder of the input when divided by 2 (using the modulo operator) and checking if it is 0 (even number), 1 (positive odd number) or, –1 (negative odd number). An implementation of this can be seen in Snippet 2.9.

Snippet 2.9: Odd or Even

```
int input;
cin >> input;
if (input % 2 == 0) {
    cout << input << " is even!" << endl;
}
if (input % 2 == 1 || input % 2 == -1) {
    cout << input << " is odd!" << endl;
}
```

An if statement consists of two parts – a ***condition***, given inside brackets after the **`if`** keyword, followed by a body – some lines of code surrounded by curly brackets. The code inside the body will be executed in case the condition evaluates to `true`.

Our odd or even example contains a certain redundancy. If a number is not even we know it is odd. Checking this explicitly using the modulo operator is a bit unnecessary. Indeed, there is a construct that saves us from this verbosity – the ***else*** statement. It is used after an if statement and contains code that should be run if the condition given to the condition of an if statement is `false`. We can adopt this to simplify our odd and even program to the one in Snippet 2.10.

Snippet 2.10: Odd or Even 2

```
int input;
cin >> input;
if (input % 2 == 0) {
    cout << input << " is even!" << endl;
} else {
    cout << input << " is odd!" << endl;
}
```

There is one last if-related construct – the ***else if***. Since code is worth a thousand words, we demonstrate how it works in Snippet 2.11 by implementing a helper for the children's game *FizzBuzz*. In FizzBuzz, one goes through the natural numbers in increasing order and say them out loud. When the number is divisible by 3 you instead say Fizz. If it is divisible by 5 you say Buzz, and if it is divisible by both you say FizzBuzz.

Snippet 2.11: Else If

```
int input;
cin >> input;
if (input % 15 == 0) {
    cout << "FizzBuzz" << endl;
} else if (input % 5 == 0) {
    cout << "Buzz" << endl;
} else if (input % 3 == 0) {
    cout << "Fizz" << endl;
} else {
    cout << input << endl;
}
```

Exercise 2.20. Run the program in Snippet 2.11 with the values 30, 10, 6, 4. Explain the output you get.

Problem 2.21.

Sort Two Numbers	sorttwonumbers
Expected Earnings	expectedearnings
Quadrant Selection	quadrant
Grading	grading
Spavanac	spavanac
Cetvrta	cetvrta

2.6 For Loops

Another rudimentary building block of programs is the ***for loop***. A for loop is used to execute a block of code multiple times. The most basic loop repeats code a fixed number of times as in Snippet 2.12.

Snippet 2.12: For Loops

```
int repetitions = 0;
cin >> repetitions;
for (int i = 0; i < repetitions; i++) {
    cout << "This is repetition " << i << endl;
}
```

A for loop consists of four parts. The first three parts are the semi-colon separated statements immediately after the **`for`** keyword. In the first of them, you write some statement, such as a variable declaration. In the second part, you write an expression that evaluates to a **`bool`**, such as a comparison between two values. In the third part you write another statement.

The first statement is executed only once – it is the first thing that happens in a loop. In this case, we declare a new variable `i` and set it to 0. The loop will then be repeated until the condition in the second part is false. Our example loop will repeat until `i` is no longer

less than `repetitions`. The third part executes after each execution of the loop. Since we use the variable `i` to count how many times the loop has executed, we want to increment this by 1 after each repetition.

Together, these three parts make sure our loop will run exactly `repetitions` times. The final part of the loop is the statements within curly brackets. Just as with the if statements, this is called the body of the loop. It contains the code that to be executed in each repetition. A repetition of a loop is in algorithm language referred to as an ***iteration***.

Exercise 2.22. What happens if you enter a negative value as the number of loop repetitions?

Exercise 2.23. Design a loop that instead counts backwards, from *repetitions* – 1 to 0.

Problem 2.24.

N-Sum	nsum	
Building Pyramids	pyramids	*(both subtasks)*
Odd Echo	oddecho	*(both subtasks)*
Cinema Crowds	cinema	
Refrigerator Transport	refrigerator	*(both subtasks)*

Within a loop, two useful keywords can be used to modify the loop – **`continue`** and **`break`**. Using **`continue`**`;` inside a loop exits the current iteration and starts the next one. **`break`**`;` on the other hand, exits the loop altogether. For an example, consider Snippet 2.13.

Snippet 2.13: Break and Continue

```
int check = 35;

for (int divisor = 2; divisor * divisor <= check; ++divisor) {
  if (check % divisor == 0) {
    cout << check << " is not prime!" << endl;
    cout << "It equals " << divisor << " x "
      << (check / divisor) << endl;
    break;
  }
}

for (int divisor = 1; divisor <= check; ++divisor) {
  if (check % divisor == 0) {
    continue;
  }
  cout << divisor << " does not divide " << check << endl;
}
```

Exercise 2.25. What will the following code snippet output?

```
for (int i = 0; false; i++) {
  cout << i << endl;
}

for (int i = 0; i >= -10; --i) {
```

```
  cout << i << endl;
}

for (int i = 0; i <= 10; ++i) {
  if (i % 2 == 0) continue;
  if (i == 8) break;
  cout << i << endl;
}
```

Problem 2.26.

Cinema Crowds 2	cinema2	
Lamps	lamps	*(both subtasks)*

2.7 While Loops

There is a second kind of loop, which is simpler than the for loop. It is called a ***while loop***, and works like a for loop where the initial statement and the update statement are removed, leaving only the condition and the body. It can be used when you want to loop over something until a certain condition is false (Snippet 2.14).

Snippet 2.14: While

```
int num = 9;
while (num != 1) {
    if (num % 2 == 0) {
        num /= 2;
    } else {
        num = 3 * num + 1;
    }
    cout << num << endl;
}
```

The `break;` and `continue;` statements work the same way as the do in a for loop.

Problem 2.27.

3D Printed Statues	3dprinter
Hailstone Sequences	hailstone2
Soda Slurper	sodaslurper

2.8 Functions

In mathematics, a ***function*** is something that takes one or more arguments and computes some value based on them. Common functions include the squaring function $\text{square}(x) = x^2$, the addition function $\text{add}(x, y) = x + y$ or, the minimum function $\min(a, b)$ which evaluates to the smallest of its arguments.

Functions exists in programming as well but work slightly differently. Indeed, we have already seen a function – the `main()` function. We have implemented the example functions in Snippet 2.15.

Snippet 2.15: Functions

```
#include <iostream>

using namespace std;

int square(int x) {
  return x * x;
}

int min(int x, int y) {
  if (x < y) {
    return x;
  } else {
    return y;
  }
}

int add(int x, int y) {
  return x + y;
}

int main() {
  int x, y;
  cin >> x >> y;
  cout << x << "^2 = " << square(x) << endl;
  cout << x << " + " << y << " = " << add(x, y) << endl;
  cout << "min(" << x << ", " << y << ") = " << min(x, y) << endl;
}
```

In the same way that a variable declaration starts by declaring what data type a variable contains, a function declaration states what data type the function evaluates to. Afterwards, we write the name of the function followed by its arguments, written as a comma-separated list of variable declarations. Finally, we give it a body of code wrapped in curly brackets.

All of these functions contain a statement with the **`return`** keyword, unlike our main function. A **`return`** statement says "stop executing this function, and return the following value". Thus, when we call the squaring function by `square(x)`, the function will compute the value `x * x` and make sure that `square(x)` evaluates to just that.

Why have we left a return statement out of the main function? In `main()`, the compiler inserts an implicit **`return`** `0;` statement at the end of the function.

Exercise 2.28. What does the following function call evaluate to?

```
min(square(10), add(square(9), 23));
```

Exercise 2.29. We declared all of the new arithmetic functions above our `main` function in the example. Why did we do this? What happens if you move one below the `main` function instead?

Exercise 2.30. Research online what a *forward declaration* of a function is, and how it resolves the problem from Exercise 2.29.

Problem 2.31.

Arithmetic Functions arithmeticfunctions

An important caveat when calling functions is that the arguments we send along are copied. If we try to change them by assigning values to our arguments, we will not change the original variables in the calling function (see Snippet 2.16 for an example).

Snippet 2.16: Argument Copying

```
void change(int val) {
  val = 0;
}

int main() {
  int variable = 100;
  change(variable);
  cout << "Variable is " << variable << endl;
}
```

We can choose not to return *anything* with the **void** return type. This may seem useless since nothing ought to happen if we call a function but does not get anything in return. However, there are ways we can affect the program without returning.

The first one is by using ***global*** variables. They are variables declared outside of a function, available to every function in your program. Changes to a global variable by one function are also seen by other functions (try out Snippet 2.17 to see them in action).

Snippet 2.17: Global Variables

```
int currentMoney = 0;

void deposit(int newMoney) {
  currentMoney += newMoney;
}
void withdraw(int withdrawal) {
  currentMoney -= withdrawal;
}

int main() {
  cout << "Currently, you have " << currentMoney << " money" << endl;
  deposit(1000);
  withdraw(2000);
  cout << "Oh-oh! Your balance is " << currentMoney << " :(" << endl;
}
```

Problem 2.32.

Counting Days countingdays

Secondly, we can change the variables passed to a function as arguments by declaring them as ***references***. Such an argument is written by adding a `&` before the variable name, for example `int &x`. If we perform assignments to the variable `x` within the function we

change the variable used for this argument in the calling function instead. Snippet 2.18 contains an example of using references.

Snippet 2.18: References

```
// Note &val instead of val
void change(int &val) {
  val = 0;
}

int main() {
  int variable = 100;
  cout << "Variable is " << variable << endl;
  change(variable);
  cout << "Variable is " << variable << endl;
}
```

Problem 2.33.
Logic Functions logicfunctions

Exercise 2.34. Why is the function call `change(4)` not valid C++?

2.9 Structures

Algorithms operate on data, usually lots of it. Programming language designers therefore came up with many ways of organizing the data that programs use. One of these constructs is the ***structure*** (also called a record, and in C++ almost equivalent to something called a class). Structures are a special kind of data type that can contain variables (then called a ***member variable***) inside the structure, and ***member functions*** which can operate on member variables.

The basic syntax used to define a structure looks like this:

```
struct Point {
    double x;
    double y;
};
```

This structure contains two member variables, `x` and `y`, representing the coordinates of a point in the Euclidean plane.

Once a structure is defined we can create ***instances*** of it. Every instance has its own copy of the member variables of the structure. Structures essentially encapsulate concepts, like what a "book" is, while instances of the structure represent individual, particular books (like this one!).

To create an instance of a struct, use the same syntax as with other variables. We can get the value of a member variable of a structure using the syntax `instance.variable`.

Snippet 2.19: Using structures

```
Point origin; // create an instance of the Point structure

// set the coordinates to (0, 0)
origin.x = 0;
origin.y = 0;

cout << "The origin is (" << origin.x << ", "
     << origin.y << ")." << endl;
```

As you can see structures allow us to group certain kinds of data together in a logical fashion. Later on, this will simplify the coding of certain algorithms and data structures immensely.

There is an alternate way of constructing instances called ***constructors***. A constructor looks like a function inside our structure and allows us to pass arguments when we create a new instance of a struct. The constructor can use these arguments to help set up the instance.

Snippet 2.20: Constructors

```
struct Point {
  // ... everything previously defined

  Point(double theX, double theY) {
    x = theX;
    y = theY;
  }
};
```

The newly added constructor lets us pass two arguments when constructing the instance to set the coordinates correctly. With it, we avoid the two extra statements to set the member variables.

```
Point p(4, 2.1);
cout << "The point is (" << p.x << ", " << p.y << ")." << endl;
```

Structure values can also be constructed outside of a variable declaration using the syntax

```
Point(1, 2);
```

so that we can reassign a previously declared variable with

```
p = Point(1, 2);
```

We can also define functions inside the structure. These functions work just like any other except they can also access the member variables of the instance that the member function is called on. For example, we might want a convenient way to mirror a certain point in the X-axis, scale a certain point with an integer coefficient or print the coordinates of the point. This could be accomplished by adding member functions.

Snippet 2.21: Member functions

```
struct Point {
  // ... everything previously defined

  // This creates a new, mirrored point.
  Point mirror() {
    return Point(x, -y);
  }

  // This changes the point itself.
  void scale(int scale) {
    x = x * scale;
    y = y * scale;
  }

  // No changes and returns nothing.
  void print() {
    cout << "(" << x << "," << y << ")" << endl;
  }
};
```

To call member functions, we type its name and provide the list with arguments, for example:

```
Point p(1, 2);
p.print();
Point mirrored = p.mirror();
mirrored.print();
// mirror() returns a new point, so p is unchanged
p.print();
p.scale(4);
// scale() changes the member variables, so p is changed
p.print();
```

In this example we see yet another use of a **`void`** function. Such member functions can still modify the member variables of the struct the belong to.

Exercise 2.35. Add a `translate` member function to the point structure. It should take two double values `x` and `y` as arguments and return a new point that is the instance point translated by (x, y).

Similarly to how the **`const`** modifier could be added to a variable declaration, a member function can also be declared to be **`const`**:

```
Point mirror() const {
    return Point(x, -y);
}
```

The keyword must be added right before the last brace. Such a function is unable to modify any of the member variables. It can not call other member functions that are not declared as **`const`** either. There is one context where you need to worry about **`const`**, namely when using your own structures with the C++ standard library in certain ways.

Exercise 2.36. What happens if we try to change a member variable in a `const` member function?

Finally, C++ has a powerful mechanism called ***operator overloading***. It allows us to define how various operators such as `+` should behave if we apply them to instances of a struct. For example, we could define what happens when we write `a + b` where `a` and `b` are `Point`s. The syntax for the binary operators looks like this:

Snippet 2.22: Operator Overloading

```
Point operator+(Point other) {
    double newX = x + other.x;
    double newY = y + other.y;
    return Point(newX, newY);
}
```

Try this function out by defining two points and computing their sum.

Exercise 2.37. One can use operator overloading for binary operators where the types are different as well. For example,

```
Point operator*(double m) { ... }
```

would define what happens if you multiply a point by a double. Add such a function to your point, that returns a point with its coordinates scaled by the given double.

Exercise 2.38. Fill in the remaining code to implement this structure:

```
1  struct Quotient {
2      // .. member variables?
3      // Construct a new Quotient with the given numerator and denominator
4      Quotient(int n, int d) { }
5      // Return a new Quotient, this instance plus the "other" instance
6      Quotient add(const Quotient &other) const { }
7      // Return a new Quotient, this instance times the "other" instance
8      Quotient multiply(const Quotient &other) const { }
9      // Output the value on the screen in the format n/d
10     void print() const { }
11 };
```

2.10 Arrays

In the sorting problem from Chapter 1 we often spoke of the data type "sequence of integers". Until now, none of the data types we have seen in C++ represents this kind of data. We present the ***array***. It is a special type of variable capable of storing a large number of variables of the same type. For example, it could be used to represent the recurring data type "sequence of integers" from the sorting problem in Chapter 1. To declare an array, we specify the type of variable it should contain, its name, and its size using the syntax:

```
type name[size];
```

For example, an integer array of size 10 named `seq` would be declared with

```
int seq[10];
```

This creates 10 integer "variables" which we can refer to using the syntax `seq[index]`, starting from zero (they are zero-indexed). Thus we can use `seq[0]`, `seq[1]`, etc., all the way up to `seq[9]`. The values are called the ***elements*** of the array.

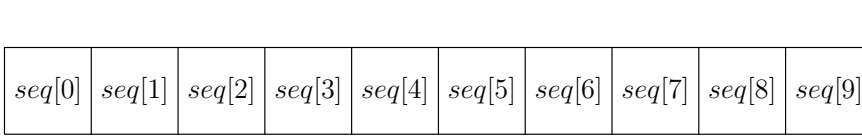


Figure 2.1: A 10-element array called `seq`.

Be aware that using an index outside the valid range for a particular array (i.e. below 0 or above the size – 1) can cause erratic behavior in the program without crashing it.

If you declare a global array all elements get a default value. For numeric types this is 0, for booleans this is `false`, for strings this is the empty string and so on. If, on the other hand, the array is declared in the body of a function that guarantee does not apply. Instead of being zero-initialized, the elements *can have random values*. For this reason, arrays are mostly declared globally in competitive programming.

You can see an example of arrays in action in Snippet 2.23, which computes a few of the the possible scores of a roll in the dice game Yahtzee.

Snippet 2.23: Arrays

```
#include <iostream>
using namespace std;

int rolls[7];

int main() {
    cout << "Enter 5 dice rolls between 1 and 6: " << endl;
    for (int i = 0; i < 5; i++) {
        int roll;
        cin >> roll;
        rolls[roll]++;
    }
    cout << "Yatzee scores: " << endl;
    for (int i = 1; i <= 6; i++) {
        cout << i << "'s: " << (i * rolls[i]) << endl;
    }
}
```

Later on (Section 3.1.2) we transition from using arrays to a much more powerful structure from the standard library that serves the same purpose – the *vector*.

Problem 2.39.

Reverse	ofugsnuid
Modulo	modulo
Booking a Room	bookingaroom

2.11 Lambdas

We now briefly discuss a somewhat complex language construct – ***lambdas***. They are seldom strictly necessary to solve problems, but make working with the standard library a bit simpler at times. They are also good to know of when reading C++ code others wrote.

A lambda expression is essentially an *unnamed* function that can be defined within another function and assigned to a variable of the `function` type:

Snippet 2.24: Lambda Functions

```
#include <iostream>
#include <functional>
using namespace std;

int main() {
    function<int(int, int)> op = [](int a, int b) -> int {
        return a * b + a + b;
    };
    cout << op(5, op(1, 2)) << endl;
}
```

Here, we have defined a function that takes two values `a` and `b` and returns the value `a * b + a + b`. We have assigned the function to the variable `op`. It can be invoked as if it was a regular function with that name.

Generally, definitions look simpler than this. If the compiler can figure out what type all the return values have, we can ignore the `-> int` part, which is otherwise used to specify the type of the lambda's return value. We also tend to use the **`auto`** type instead of the more convoluted `function<...>` type, as long as **the lambda does not call itself** through the name of the variable to which it is assigned.

Thus, the declaration may also look like this:

```
auto op = [](int a, int b) {
    return a * b + a + b;
};
```

What is the point of doing this rather than simply using regular functions? Lambdas can also be given access to variables of *the enclosing function*:

```
int x = 5;
auto addToX = [&](int y) {
    x += y;
};
```

Here, note the added ampersand in `[&]`. This means that all variables defined before the lambda in the function should be accessible within the lambda **as references**.

Exercise 2.40. Use the internet to figure out:

- how to only make a single variable from the enclosing function available in a lambda,
- how to make variables within the enclosing function available **as copies** rather than as references, and
- how lambdas can be passed as arguments to other functions.

2.12 The Preprocessor

C++ has a powerful tool called the ***preprocessor***. This utility is able to read and modify your code using certain rules during compilation. The commonly used `#include` is a preprocessor directive that includes a certain file in your code.

Besides file inclusion, we mostly use the `#define` directive. It allows us to replace certain tokens in our code with other ones. The most basic usage is

```
#define TOREPLACE REPLACEWITH
```

which replaces the token `TOREPLACE` in our program with `REPLACEWITH`. The true power of the define comes when using define directives with parameters. These look similar to functions and allows us to replace certain expressions with another one, additionally inserting certain values into it. We call these ***macros***. For example the macro

```
#define rep(i,a,b) for (int i = a; i < b; i++)
```

means that the expression

```
rep(i,0,5) {
    cout << i << endl;
}
```

is expanded to

```
for (int i = 0; i < 5; ++i) {
    cout << i << endl;
}
```

You can probably get by without ever using macros in your code. The reason we discuss them is similar to that of lambda's: they are used widely in competitive programming, so it is good to at least be familiar with what they do.

Additional Exercises

Problem 2.41.

Solving for Carrots	carrots
Paul Eigon	pauleigon
Stuck In A Time Loop	timeloop
Sibice	sibice
Cold-puter Science	cold
Tarifa	tarifa

Patuljci	patuljci
Left Beehind	leftbeehind
No Duplicates	nodup
I've Been Everywhere, Man	everywhere
Right-of-Way	vajningsplikt

Notes

C++ was invented by Danish computer scientist Bjarne Stroustrup. Bjarne has also published a book on the language, *The C++ Programming Language* [51], that contains a more in-depth treatment of the language. It is rather accessible to C++ beginners but is better read by someone who have some prior programming experience (in any programming language).

C++ is standardized by the International Organization for Standardization (ISO). These standards are the authoritative source on what C++ is. The final drafts of the standards can be downloaded at the homepage of the Standard C++ Foundation[1].

There are many online references of the language and its standard library. The two we use most are:

- `heap.link/cpp:cppreference`
- `heap.link/cpp:cplusplus`

[1] `heap.link/cpp:iso`

Chapter 3

The C++ Standard Library

In this chapter we study parts of the C++ standard library – that is, data structures, algorithms and utilities that are already provided for us without having to code them ourselves.

We start by examining a number of basic *data structures*. Data structures help us organize the data we work with in the hope of making processing both easier and more efficient. Different data structures serve widely different purposes and solve different problems. Whether a data structure fits our needs depends on what operations we wish to perform on the data. We consider neither the efficiency of the various operations nor how they are implemented in this chapter. These concerns are postponed until Chapter 6, when we have the tools to analyze the efficiency of data structures.

The standard library also contains many useful algorithms such as sorting and various mathematical functions. These are discussed after the data structures.

In the end, we take a deeper look at string handling in C++ and some more input/output routines.

3.1 Data Structures

Aside from input and output, the most important part of the standard library is its many ***containers***. They range from the very simple to structures highly difficult to understand and implement efficiently if we were to do so ourselves. A majority of the problems you solve requires use of common data structures, so it is important that you are well acquainted with them.

Pairs

As a small taste of what is to come, we start by looking at the `pair`. It is the simplest of the data structures, containing a pair of two values of any types. A pair containing variables of types `A` and `B` is declared by

```
pair<A, B> name;
```

This angled bracket syntax to specify the type of value stored in a container appears for every data structure from the standard library. To use pairs, you must include `#include<utility>`.

It also has a constructor that takes to values and stores them in the pair directly:

```
pair<string, int> bovine("cow", 4);
```

The two variables in a pair are stored in the member variables `first` and `second`, so they can be updated and retrieved easily:

```
pair<int, int> p(5, 4);
int sum = p.first + p.second;
p.first = 10;
```

There is also a function that can construct a pair for us without us having to write out the types:

```
pair<string, string> dict("hello", "hejsan");
dict = make_pair("goodbye", "au revoir");
```

Vectors

One of the latter things discussed in the C++ chapter was the fixed-size array. As you might remember, the array is a special kind of data type that allows us to store multiple values of the same data type inside what appeared as a single variable. Arrays are a bit awkward to work with in practice. When passing them as parameters we must also pass along the size of the array. We are unable to change the size of arrays once declared nor can we easily remove or insert elements, or copy arrays.

The *dynamic array* is a special type of array that can change size (hence the name dynamic). It also supports operations such as removing and inserting elements at any position in the list.

The C++ standard library includes a dynamic array called a `vector`, which is an alternative name for dynamic arrays in some languages. To use it you must include the `vector` file by adding the line

```
#include <vector>
```

among your other includes at the top of your program.

When declaring vectors, they need to know what type of data they should store, just like pairs. To create a vector containing strings named `words` we write

```
vector<string> words;
```

Once a vector is created elements can be appended to it using the `push_back` member function. The following four statements would add the words

```
Simon is a fish
```

as separate elements to the vector:

```
words.push_back("Simon");
words.push_back("is");
words.push_back("a");
words.push_back("fish");
```

To refer to a specific element in a vector you can use the same operator `[]` as for arrays. Thus, `words[i]` refers to the *i*'th value in the vector (starting at 0).

```
cout << words[0] << " " << words[1] << " "; // Prints Simon is
cout << words[2] << " " << words[3] << " "; // Prints a fish
```

Like arrays, accessing indices outside the valid range of the vector can cause weird behavior in your program.

We can get the current size of an array using the `size()` member function:

```
cout << "The vector contains " << words.size() << " words" << endl;
```

There is also an `empty()` function that can be used to check if the vector contains no elements. These two functions are part of almost every standard library data structure.

Problem 3.1.
Vector Functions vectorfunctions

You can also create dynamic arrays that already contain a number of elements. This is done by passing an integer argument when first declaring the vector. They are filled with the same default value as (global) arrays are when created:

```
vector<int> vec(5); // creates a vector containing 5 zeroes
```

The value that an array should be filled with can be set explicitly by using a two-argument constructor; the second argument is the value to fill the array with:

```
vector<int> vec(5, -1); // creates a vector containing 5 -1's
```

Exercise 3.2. What happens when you create vectors of a *struct*? Try using structures:

- without a constructor,
- with a zero argument constructor,
- with only non-zero argument constructors, and
- with both zero argument and non-zero argument constructors.

We can create vectors that contain other vectors, to make *multidimensional* vectors. A 2-dimensional vector (i.e. a grid of values) in the following way:

```
vector<vector<int>> grid(7, vector<int>(5));
```

Since we filled the vector with 7 vectors of length 5, we get a 7×5 grid of integers. The values in the grid are then referred to by `grid[a][b]` where $0 \leq a < 5$ and $0 \leq b < 7$.

Similarly, one can create N-dimensional vectors by creating vectors of vectors of ... and so on.

Problem 3.3.
Cinema Seating cinemaseating

Other occasionally useful functions supported by vectors are:

- `pop_back()`: remove the last element of a vector,

- `clear()`: remove all elements of a vector,
- `front()`: get the first element of a vector,
- `back()`: get the first element of a vector, and
- `assign(n, val)`: replace the contents of the vector with n copies of `val`.

Iterators

A concept central to the standard library is the ***iterator***. An iterator is an object which "points to" an element in some kind of data structure (such as a vector). Essentially, they are a generalization of the role played by an integer representing an index of a vector. The reason we could not simply eliminate them and use integer indexes wherever iterators appear is that some data structures do not support accessing values directly by their index. Not all data structures support iterators either.

The type of an iterator for a data structure of type `t` is `t::iterator`. An iterator of a `vector<string>` thus has the type `vector<string>::iterator`. Most of the time we instead use the **`auto`** type since this is very long to type.

To get an iterator to the first element of a `vector`, we use `begin()`:

```
auto first = words.begin();
```

We can get the value that an iterator points at using the `*` operator:

```
cout << "The first word is " << *first << endl;
```

If we have an iterator `it` pointing at the i'th element of a vector we can get a new iterator pointing to another value in one of two ways. For iterators of a vector, we add or subtract an integer value to the iterator. For example, `it + 4` points to the $(i + 4)$'th element of the vector, and `it - 1` is the iterator pointing to the $(i - 1)$'st element.

For those structures that do not support access by indexes, the iterators can instead by *moved* forwards and backwards using the `++` and `--` operators, i.e. by writing `it++` and `it--`.

There is a special kind of iterator that points to the first position *after* the last element. We get this iterator by using the function `end()`. It allows us to iterate through a vector in the following way:

```
for (auto it = words.begin(); it != words.end(); it++) {
        string value = *it;
        cout << value << endl;
    }
```

In this loop we start by creating an iterator which points to the first element of the vector. The update statement repeatedly moves the iterator to the next element in the vector. The loop condition ensures that the loop breaks when the iterator first points to the element past the end of the vector.

In modern C++ language versions, there is a shorter construct that is equivalent to this loop:

```
for (auto value : words) {
        cout << value << endl;
    }
```

In addition to the `begin()` and `end()` pair of iterators, there is also `rbegin()` and `rend()`. They work similarly, except that they are *reverse iterators* - they iterate in the other direction. Thus, `rbegin()` actually points to the last element of the vector, and `rend()` to an imaginary element before the first element of the vector. If we move a reverse iterator in a positive direction, we will actually move it backwards (i.e. adding 1 to a reverse iterator makes it point to the element *before* it in the vector).

Exercise 3.4. Use the `rbegin()/rend()` iterators to code a loop that iterates through a vector in the reverse order.

Certain operators on a vector require the use of vector iterators. The `insert` and `erase` member functions, used to insert and erase elements at arbitrary positions, take iterators to describe positions. When removing the third element, we write

```
words.erase(words.begin() + 2);
```

The `insert()` function uses an iterator to know at what position an element should be inserted. If it is passed the `begin()` iterator, the new element will be inserted at the start of the array. Similarly, as an alternative to `push_back()` we could have written

```
words.insert(words.end(), "food");
```

to insert an element at the end of the vector.

Exercise 3.5. After adding these two lines, what would the loop printing every element of the vector `words` output?

Problem 3.6.
Cut in Line cutinline

Queues

The `queue` structure corresponds to a plain, real-life queue. It supports mainly two operations: appending an element to the back of the queue, and extracting the first element of the queue. The structure is in the `queue` file so it must be included using

```
#include<queue>
```

As with all standard library data structures, when declaring a queue we provide the data type that we wish to store in it. A queue storing **`int`**s is created by

```
queue<int> q;
```

We use mainly five functions when dealing with queues:

- `push(x)`: add the element `x` to the **back of the queue**,

- `pop()`: remove the element from the **front of the queue**,
- `front()`: return the element from the **front of the queue**,
- `empty()`: return `true` if and only if the queue is empty, and
- `size()`: return the number of elements in the queue.

Exercise 3.7. There is a similar data structured called a ***dequeue***. The standard library version is named after the abbreviation `deque` instead. Use one of the C++ references from the C++ chapter notes (p. 35) to find out what this structure does and what its member functions are called.

Priority Queues

The queue structure is arguable unnecessary, since it can be easily emulated using a vector (see Section 6.3). This is not the case for the next structure, the `priority_queue`.

The structure is similar to a queue, but instead of insertions and extractions happening at one of the endpoints of the structure, the **greatest** element is always returned during the extraction.

The structure is located in the same file as the `queue` structure, so add

```
#include<queue>
```

to use it.

To initialize a priority queue, use the same syntax as for the other structures:

```
priority_queue<int> pq;
```

This time there is one more way to create the structure that is important to remember. It is not uncommon to prefer the sorting to be done according to some other order than descending. For this reason there is another way of creating a priority queue. One can specify a ***comparison function*** that takes two arguments of the type stored in the queue and returns true if the first one should be considered less than the second. This function can be given as an argument to the type in the following way:

```
bool cmp(int a, int b) {
        return a > b;
    }

priority_queue<int, vector<int>, cmp> pq;
// or equivalently
priority_queue<int, vector<int>, greater<int>> pq;
```

By default, a priority queue returns its greatest element. If we want it to return the smallest element, the comparison function needs to say that the smallest of the two elements actually is the greatest, somewhat counter-intuitively.

Interactions with the queue is similar to that of the other structures:

- `push(x)`: add the element `x` to the priority queue,

- `pop()`: remove the **greatest** element from the priority queue,
- `top()`: return the **greatest** element from the priority queue,
- `empty()`: return `true` if and only if the priority queue is empty, and
- `size()`: return the number of elements in the priority queue.

Problem 3.8.

Akcija	akcija
Cookie Selection	cookieselection
Pivot	pivot

Sets and Maps

The final data structures in this chapter are also the most powerful: the `set` and the `map`.

The set structure is similar to a mathematical set (Section B.1), in that it contains a collection of unique elements. Unlike the vector, particular positions in the structure can not be accessed using the `[]` operator. This may make sets seem worse than vectors. The advantage of sets is twofold. First, we can determine membership of elements in a set much more efficiently compared to when using vectors (in Chapters 5 and 6, what this means will become clear). Secondly, sets are automatically sorted. This means we can quickly find the smallest and greatest values of the set.

Elements are accessed only through iterators, obtained using the `begin(), end()` and `find()` member functions. These iterators can be moved using the `++` and `--` operators, allowing us to navigate through the set in sorted (ascending) order (with `begin()` referring to the smallest element).

Elements are inserted using the `insert` function and removed using the `erase` function. A concrete example usage is found in Snippet 3.1

Snippet 3.1: Sets

```
set<int> s;
s.insert(4);
s.insert(7);
s.insert(1);

// find returns an iterator to the element if it exists
auto it = s.find(4);
// ++ moves the iterator to the next element in order
++it;
cout << *it << endl;

// if nonexistant, find returns end()
if (s.find(7) == s.end()) {
    cout << "7 is not in the set" << endl;
}

```

```
// erase removes the specified element
s.erase(7);

if (s.find(7) == s.end()) {
    cout << "7 is not in the set" << endl;
}

cout << "The smallest element of s is " << *s.begin() << endl;
```

When looping through a set using the `for (auto v : s)` syntax, the values come in ascending order.

A structure similar to the set is the `map`. It is basically the same as a set, except that elements are called *keys* and have associated *values*. When declaring a map two types need to be provided – that of the key and that of the value. To declare a map with `string` keys and `int` values you write

```
map<string, int> m;
```

Accessing the value associated with a key `x` is done using the `[]` operator, for example, `m["Johan"];` would access the value associated with the `"Johan"` key.

Snippet 3.2: Maps

```
map<string, int> age;
age["Johan"] = 22;
age["Simon"] = 23;

if (age.find("Aron") == age.end()) {
    cout << "No record of Aron's age" << endl;
}

cout << "Johan is " << age["Johan"] << " years old" << endl;
cout << "Anton is " << age["Anton"] << " years old" << endl;

age.erase("Johan");
cout << "Johan is " << age["Johan"] << " years old" << endl;

auto last = --age.end();
cout << (*last).first << " is "
     << (*last).second << " years old" << endl;
```

Take careful note of what the map does when accessing non-existent keys using the `[]` operator.

Exercise 3.9. Maps can also be iterated through using the `for (auto v : m)` syntax. What type will `v` have? Search online for documentation of the map to determine this.

Problem 3.10.

Secure Doors	securedoors
Babelfish	babelfish

3.2 Math

Many algorithmic problems require mathematical functions. In particular there is a heavy use of square roots and trigonometric functions in geometry problems. These of these functions are be found in another library:

```
#include <cmath>
```

It contains many common mathematical functions, such as

- `abs(x)`: computes $|x|$ (x if $x \geq 0$, otherwise $-x$)
- `sqrt(x)`: computes $\sqrt{x}$
- `pow(x, y)`: computes x^y
- `exp(x)`: computes e^x
- `log(x)`: computes $\ln(x)$
- `cos(x) / acos(x)`: computes $\cos(x)$ and $\arccos(x)$ respectively
- `sin(x) / asin(x)`: computes $\sin(x)$ and $\arcsin(x)$ respectively
- `tan(x) / atan(x)`: computes $\tan(x)$ and $\arctan(x)$ respectively
- `ceil(x) / floor(x)`: computes $\lceil x \rceil$ and $\lfloor x \rfloor$ respectively

There are also `min(x, y)` and `max(x, y)` functions which compute the minimum and maximum of the values x and y respectively. These are not in the `cmath` library however. Instead, they are in `algorithm`.

Problem 3.11.

Vacuumba	vacuumba
Ladder	ladder
Half a Cookie	halfacookie

3.3 Algorithms

A majority of the algorithms we regularly use from the standard library operate on sequences. To use algorithms, you need to include

```
#include <algorithm>
```

Sorting

Sorting a sequences is very easy in C++. The function for doing so is named `sort`. It takes two iterators marking the beginning and end of the interval to be sorted and sorts it in-place in ascending order. To sort the first 10 elements of a vector named `v` you would use

```
sort(v.begin(), v.begin() + 10);
```

Note that the right endpoint of the interval is *exclusive* – it is not included in the interval itself. This means that you can provide `v.end()` as the end of the interval if you want to sort the entire vector.

As with `priority_queue`s or `set`s, the sorting algorithm can take a custom comparator if you want to sort according to some other order than that defined by the `<` operator. For example,

```
sort(v.begin(), v.end(), greater<int>());
```

would sort the vector `v` in descending order. You can provide other sorting functions as well. To sort numbers by their absolute value you can use the following comparator:

```
bool cmp(int a, int b) {
        return abs(a) < abs(b);
    }
sort(v.begin(), v.end(), cmp);
```

What happens if two values have the same absolute value when sorted with the above comparator? With `sort`, this behavior is not specified: they can be ordered in any way. Occasionally you want that values compared by your comparison function as equal are sorted in the same order as they were given in the input. This is called a ***stable sort***, and is implemented in C++ with the function `stable_sort`.

To check if a vector already is sorted, the `is_sorted` function can be used. It takes the same arguments as the `sort` function.

A typical use case is to want to sort a vector, but do the sorting by another value associated to each vector element. There are two main ways to do this. A map can store the association between the vector elements and the comparison elements, and looked up using a comparator function. Other times, it is easier to use the built-in `pair` data structure to store both values together in a vector. When sorting a `vector` of `pair`s, it first uses the first value in the pair, and to break ties, sorts based on the second one.

Problem 3.12.

Shopaholic	shopaholic
Busy Schedule	busyschedule
Sort of Sorting	sortofsorting

Searching

The most basic search operation is the `find` function. It takes two iterators representing an interval and a value. If one of the elements in the interval equals the value, an iterator to the element is returned. In case of multiple matches the first one is returned. Otherwise, the iterator provided as the end of the interval is returned. The common usage is

```
find(v.begin(), v.end(), 5);
```

which would return an iterator to the first instance of `5` in the vector.

To find out how many times an element appears in a vector, the `count` function takes the same arguments as the `find` function and returns the total number of matches.

If the array is sorted, you can use the much faster *binary search* operations instead. The `binary_search` function takes as argument a **sorted** interval, given by two iterators, and a value. It returns true if the interval contains the value. The `lower_bound` and `upper_bound` functions takes the same arguments as `binary_search`, but instead returns an iterator to the first element **not less** and **greater** than the specified value, respectively. For more details on how these are implemented, read Section 12.3.

A typical use case is to find the index in a vector of a given element. After retrieving an vector iterator, the correct index can be retrieved by subtracting `vec.begin()` from the vector.

Problem 3.13.

Massive Card Game massivecardgame

Permutations

In some problems, the solution involves iterating through all *permutations* (Section 18.2) of a vector. As one of few languages, C++ has a built-in functions for this purpose: `next_permutation`. The function takes two iterators as arguments and rearranges the interval they specify to be the next permutation in lexicographical order. If there is no such permutation, the interval instead becomes sorted and the function returns `false`. This suggests the following common pattern to iterate through all permutations of a vector `v`:

```
sort(v.begin(), v.end());
do {
        // do something with v
    } while (next_permutation(v.begin(), v.end()));
```

This do-while-syntax is similar to the while loop, except the condition is checked *after* each iteration instead of before. It is equivalent to

```
sort(v.begin(), v.end());
while (true) {
        // do something with v
        if (!next_permutation(v.begin(), v.end())) {
                break;
            }
    }
```

Problem 3.14.

Veci veci

3.4 Strings

We have already used the `string` type many times before. Until now one of the essential features of a string has been omitted – a `string` is to a large extent like a `vector` of **`char`**s. This is especially true in that you can access the individual characters of a string using the `[]` operator. For a string

```
string thecowsays = "boo";
```

the expression `thecowsays[0]` is the character `'b'`. You can even `push_back` new characters to the end of a string.

```
thecowsays.push_back('p');
```

would change the string to `boop`.

Problem 3.15.

Detailed Differences	detaileddifferences
Autori	autori
Skener	skener

Conversions

In some languages, the barrier between strings and e.g. integers is more fuzzy than in C++. In Java, for example, the code `"4" + 2` would append the *character* `'2'` to the string `"4"`, yielding the string `"42"`. This is not the case in C++ (what errors do you get if you try to do this?).

Instead, there are other ways to convert between strings and other types. The easiest way is through using the `stringstream`. A `stringstream` essentially works as a combined `cin` and `cout`. An empty stream is declared by

```
stringstream ss;
```

Values can be written to the stream using the << operator and read from it using the >> operator. This can be exploited to convert strings to numeric types:

```
stringstream numToString;
numToString << 5;
string val;
numToString >> val; // val is now the string "5"
```

and vice versa:

```
stringstream stringToNum;
stringToNum << "5";
int val;
stringToNum >> val; // val is now the integer 5
```

You can use a `stringstream` to determine what type the next word is. If you try to read from a `stringstream` into an **`int`** but the next word is not an integer, the expression will evaluate to false:

```
stringstream ss;
ss << "notaninteger";
int val;
if (ss >> val) {
        cout << "read an integer!" << endl;
    } else {
        cout << "next word was not an integer" << endl;
    }
```

Problem 3.16.

Filip	filip
Stacking Cups	cups

3.5 Input/Output

Input and output is primarily handled by the `cin` and `cout` objects, as previously witnessed. While they are very easy to use, adjustments are sometimes necessary.

Detecting End of File

The first advanced usage is reading input until we run out of input (often called reading until the *end-of-file*). Normally, input formats are constructed so that you always know beforehand how many tokens of input you need to read. For example, lists of integers are often either prefixed by the size of the list or terminated by some special sentinel value. For those few times when we need to read input until the end we use the fact that `cin >> x` is an expression that evaluates to `false` if the input reading failed. This is also the case if you try to read an **`int`** but the next word is not actually an integer. This kind of input loop thus looks something like the following:

```
int num;
while (cin >> num) {
        // do something with num
    }
```

Problem 3.17.

A Different Problem	different
Statistics	statistics

Input Line by Line

As we stated briefly in the C++ chapter, `cin` only reads a single word when used as input to a string. This is a problem if the input format requires us to read input line by line. The solution to this is the `getline` function, which reads text until the next newline:

```
getline(cin, str);
```

Be warned that if you use `cin` to read a single word that is the last on its line, the final newline is not consumed. That means that for an input such as

```
word
blah blah
```

the code

```
string word;
cin >> word;
string line;
getline(cin, line);
```

would produce an *empty* line! After `cin >> word` the newline of the line `word` still remains, meaning that `getline` only reads the (zero) remaining characters until the newline. To avoid this problem, you need to use `cin.ignore();` to ignore the extra newline before your `getline`.

Once a line has been read we often need to process all the words on the line one by one. For this, we can use the `stringstream`:

```
stringstream line(str);
string word;
while (line >> word) {
        // do something with word
    }
```

Problem 3.18.

Bacon Eggs and Spam	baconeggsandspam
Compound Words	compoundwords

Output Decimal Precision

Another common problem is that outputting decimal values with `cout` produces numbers with too few decimals. Many problems stipulate that an answer is considered correct if it is within some specified relative or absolute precision of the judges' answer. The default precision of `cout` is 10^{-6}. If a problem requires higher precision, it must be set manually using e.g.

```
cout << setprecision(10);
```

If the function argument is x, the precision is set to 10^{-x}. This means that the above statement would set the precision of `cout` to 10^{-10}. This precision is normally the relative precision of the output (i.e. the total number of digits to print). If you want the precision to be absolute (i.e. specify the number of digits after the decimal point) you write

```
cout << fixed;
```

Problem 3.19.

A Real Challenge	areal
Pizza Crust	pizza2

Additional Exercises

Problem 3.20.

Apaxiaaaaaaaaaaaans!	apaxiaaans
Different Distances	differentdistances
Odd Man Out	oddmanout
Timebomb	timebomb
Missing Gnomes	missinggnomes
A1 Paper	a1paper

Notes

In this chapter, only the parts from the standard library we deemed most important to problem solving were extracted. The standard library is much larger than this, of course. While you will almost always get by using only what we discussed additional knowledge of the library can make you a faster, more effective coder.

For a good overview of the library, `heap.link/cpp:cppreference` contains lists of the library contents categorized by topic.

Chapter 4

Implementation Problems

The "simplest" kind of problem we solve are those where the problem statement is so detailed that the difficult part is not figuring out the solution, but implementing it in code. This type of problem is mostly given in the form of performing some calculation or simulating some process based on a list of rules stated in the problem.

The Recipe – receptet

By Arash Rouhani. Swedish Olympiad in Informatics 2011, School Qualifiers.

You have decided to cook some food. The dish you are going to make requires $N \leq 10$ different ingredients. For every ingredient i, you know the amount you have at home h_i, how much you need for the dish n_i, and how much it costs to buy per unit c_i.

If you do not have a sufficient amount of some ingredient you need to buy the remainder from the store. Your task is to compute the cost of buying the remaining ingredients.

This problem is not particularly hard. For every ingredient we need to calculate the amount that we need to purchase. The only gotcha in the problem is the mistake of computing this as $n-h$. The correct formula is $\max(0, n-h)$, required in case of the luxury problem of having more than we need. We then multiply this number by the ingredient cost and sum the costs up for all the ingredients. A solution would look something like the following.

```
1: procedure RECIPE(N, H, N, C)
2:     ans ← 0
3:     for i ← 0 to N − 1 do
4:         ans ← ans + max(0, N[i] − H[i]) · C[i]
5:     return ans
```

4.1 Structuring your Code

The implementation problems are typically the easiest in a contest. They do not require much algorithmic knowledge so more contestants are able to solve them. However, not every implementation problem is easy to code. Just because implementation problems are easy to spot, understand, and formulate a solution to, you should not underestimate the difficulty in coding them. Contestants usually fail implementation problems either because the algorithm you are supposed to implement is very complicated with many

easy-to-miss details, or because the amount of code is very large. In the latter case, you are more prone to bugs because more lines of code tend to include more bugs.

Let us study a straightforward implementation problem that turned out to be rather difficult to code.

Game Rank – gamerank

By Jimmy Mårdell. Nordic Collegiate Programming Contest 2016. CC BY-SA 3.0.

The gaming company Sandstorm is developing an online two player game. You have been asked to implement the ranking system. All players have a rank determining their playing strength which gets updated after every game played. There are 25 regular ranks, and an extra rank, "Legend", above that. The ranks are numbered in decreasing order, 25 being the lowest rank, 1 the second highest rank, and Legend the highest rank.

Each rank has a certain number of "stars" that one needs to gain before advancing to the next rank. If a player wins a game, she gains a star. If before the game the player was on rank 6-25, and this was the third or more consecutive win, she gains an additional bonus star for that win. When she has all the stars for her rank (see list below) and gains another star, she will instead gain one rank and have one star on the new rank.

For instance, if before a winning game the player had all the stars on her current rank, she will after the game have gained one rank and have 1 or 2 stars (depending on whether she got a bonus star) on the new rank. If on the other hand she had all stars except one on a rank, and won a game that also gave her a bonus star, she would gain one rank and have 1 star on the new rank.

If a player on rank 1-20 loses a game, she loses a star. If a player has zero stars on a rank and loses a star, she will lose a rank and have all stars minus one on the rank below. However, one can never drop below rank 20 (losing a game at rank 20 with no stars will have no effect).

If a player reaches the Legend rank, she will stay legend no matter how many losses she incurs afterwards.

The number of stars on each rank are as follows:

- Rank 25-21: 2 stars
- Rank 20-16: 3 stars
- Rank 15-11: 4 stars
- Rank 10-1: 5 stars

A player starts at rank 25 with no stars. Given the match history of a player, what is her rank at the end of the sequence of matches?

Input

The input consists of a single line describing the sequence of matches. Each character corresponds to one game; '`W`' represents a win and '`L`' a loss. The length of the line is between 1 and 10 000 characters (inclusive).

Output

Output a single line containing a rank after having played the given sequence of games; either an integer between 1 and 25 or "`Legend`".

A very long problem statement! The first hurdle is finding the energy to read it from

start to finish without skipping any details. Not much creativity is needed here – indeed, the algorithm to implement is given in the statement. Despite this, it is not as easy as one would think. Although it was the second most solved problem at the contest where it was used in, it was also the one with the *worst success ratio*. On average, a team needed 3.59 attempts before getting a correct solution, compared to the runner-up problem at 2.92 attempts. None of the top 6 teams in the contest got the problem accepted on their first attempt. Failed attempts cost a lot. Not only in absolute time, but many forms of competition include additional penalties for submitting incorrect solutions.

Implementation problems get much easier when you know your programming language well and can use it to write good, structured code. Split code into functions, use structures, and give your variables good names and implementation problems become easier to code. A solution to the Game Rank problem which attempts to use this approach is given here:

```
#include <bits/stdc++.h>

using namespace std;

int curRank = 25, curStars = 0, conseqWins = 0;

int starsOfRank() {
  if (curRank >= 21) return 2;
  if (curRank >= 16) return 3;
  if (curRank >= 11) return 4;
  if (curRank >= 1) return 5;
  assert(false);
}

void addStar() {
  if (curStars == starsOfRank()) {
    --curRank;
    curStars = 0;
  }
  ++curStars;
}

void addWin() {
  int curStarsWon = 1;
  ++conseqWins;
  if (conseqWins >= 3 && curRank >= 6) curStarsWon++;

  for (int i = 0; i < curStarsWon; i++) {
    addStar();
  }
}

void loseStar() {
  if (curStars == 0) {
    if (curRank == 20) return;
    ++curRank;
    curStars = starsOfRank();
```

```
38    }
39    --curStars;
40  }
41
42  void addLoss() {
43    conseqWins = 0;
44    if (curRank <= 20) loseStar();
45  }
46
47  int main() {
48    string seq;
49    cin >> seq;
50    for (char res : seq) {
51      if (res == 'W') addWin();
52      else addLoss();
53      if (curRank == 0) break;
54      assert(1 <= curRank && curRank <= 25);
55      assert(0 <= curStars && curStars <= starsOfRank());
56    }
57    if (curRank == 0) cout << "Legend" << endl;
58    else cout << curRank << endl;
59  }
```

Note the use of the `assert()` function. The function takes a single boolean parameter and crashes the program with an assertion failure if the parameter evaluated to `false`. It allows us to verify that assumptions we make regarding the internal state of the program indeed holds. In fact, when the above solution was written the assertions in it caught some bugs before it was submitted!

Next, we work through a complex implementation problem, starting with a long, hard-to-read solution with a few bugs. Then, we refactor it a few times until it is correct and easy to read.

Mate in One – mateinone

Introduction to Algorithms at Danderyds Gymnasium

"White to move, mate in one."

When you are looking back in old editions of the New in Chess magazine, you find loads of chess puzzles. Unfortunately, you realize that it was way too long since you played chess. Even trivial puzzles such as finding a mate in one now far exceed your ability.

But, perseverance is the key to success. You realize that you can instead use your new-found algorithmic skills to solve the problem by coding a program to find the winning move.

You will be given a chess board, which satisfy:

- No player may castle.
- No player can perform an en passant[a].
- The board is a valid chess position.
- White can mate black in a single, unique move.

Write a program to output the move white should play to mate black.

Input

The board is given as a 8×8 grid of letters. The letter `.` represent an empty space, the characters `pbnrqk` represent a white pawn, bishop, knight, rook, queen and king, and the characters `PBNRQK` represents a black pawn, bishop, knight, rook, queen and king.

Output

Output a move on the form `a1b2`, where `a1` is the square to move a piece from (written as the column, `a-h`, followed by the row, `1-8`) and `b2` is the square to move the piece to.

[a] If you are not aware of this special pawn rule, do not worry – knowledge of it is irrelevant with regard to the problem.

Our first solution attempt clocks in at about 300 lines.

```
#include <bits/stdc++.h>
using namespace std;

#define rep(i,a,b) for (int i = (a); i < (b); ++i)
#define trav(it, v) for (auto& it : v)
#define all(v) (v).begin(), (v).end()
typedef pair<int, int> ii;
typedef vector<ii> vii;
template <class T> int size(T &x) { return x.size(); }

char board[8][8];

bool iz_empty(int x, int y) {
  return board[x][y] == '.';
}

bool is_white(int x, int y) {
  return board[x][y] >= 'A' && board[x][y] <= 'Z';
}

bool is_valid(int x, int y) {
  return x >= 0 && x < 8 && y >= 0 && y < 8;
}

int rook[8][2] = {
  {1, 2},
  {1, -2},
  {-1, 2},
  {-1, -2},

  {2, 1},
  {-2, 1},
  {2, -1},
  {-2, -1}
};

void display(int x, int y) {
  printf("%c%d", y + 'a', 7 - x + 1);
}

vii next(int x, int y) {
  vii res;

  if (board[x][y] == 'P' || board[x][y] == 'p') {
    // pawn

    int dx = is_white(x, y) ? -1 : 1;

    if (is_valid(x + dx, y) && iz_empty(x + dx, y)) {
      res.push_back(ii(x + dx, y));
    }

    if (is_valid(x + dx, y - 1)
            && is_white(x, y) != is_white(x + dx, y - 1)) {
```

```
55        res.push_back(ii(x + dx, y - 1));
56      }
57
58      if (is_valid(x + dx, y + 1)
59              && is_white(x, y) != is_white(x + dx, y + 1)) {
60        res.push_back(ii(x + dx, y + 1));
61      }
62
63    } else if (board[x][y] == 'N' || board[x][y] == 'n') {
64      // knight
65
66      for (int i = 0; i < 8; i++) {
67        int nx = x + rook[i][0],
68        ny = y + rook[i][1];
69
70        if (is_valid(nx, ny) && (iz_empty(nx, ny) ||
71            is_white(x, y) != is_white(nx, ny))) {
72          res.push_back(ii(nx, ny));
73        }
74      }
75
76    } else if (board[x][y] == 'B' || board[x][y] == 'b') {
77      // bishop
78
79      for (int dx = -1; dx <= 1; dx++) {
80        for (int dy = -1; dy <= 1; dy++) {
81          if (dx == 0 && dy == 0)
82            continue;
83
84          if ((dx == 0) != (dy == 0))
85            continue;
86
87          for (int k = 1; ; k++) {
88            int nx = x + dx * k,
89              ny = y + dy * k;
90
91            if (!is_valid(nx, ny)) {
92              break;
93            }
94
95            if (iz_empty(nx, ny) || is_white(x, y) != is_white(nx, ny)) {
96              res.push_back(ii(nx, ny));
97            }
98
99            if (!iz_empty(nx, ny)) {
100             break;
101           }
102         }
103       }
104     }
105
106   } else if (board[x][y] == 'R' || board[x][y] == 'r') {
107     // rook
108
109     for (int dx = -1; dx <= 1; dx++) {
110       for (int dy = -1; dy <= 1; dy++) {
111         if ((dx == 0) == (dy == 0))
112           continue;
113
114         for (int k = 1; ; k++) {
115           int nx = x + dx * k,
116             ny = y + dy * k;
117
118           if (!is_valid(nx, ny)) {
119             break;
120           }
121
122           if (iz_empty(nx, ny) || is_white(x, y) != is_white(nx, ny)) {
123             res.push_back(ii(nx, ny));
124           }
125
126           if (!iz_empty(nx, ny)) {
127             break;
128           }
129         }
130       }
131     }
```

```
132
133   } else if (board[x][y] == 'Q' || board[x][y] == 'q') {
134     // queen
135
136     for (int dx = -1; dx <= 1; dx++) {
137       for (int dy = -1; dy <= 1; dy++) {
138         if (dx == 0 && dy == 0)
139           continue;
140
141         for (int k = 1; ; k++) {
142           int nx = x + dx * k,
143             ny = y + dy * k;
144
145           if (!is_valid(nx, ny)) {
146             break;
147           }
148
149           if (iz_empty(nx, ny) || is_white(x, y) != is_white(nx, ny)) {
150             res.push_back(ii(nx, ny));
151           }
152
153           if (!iz_empty(nx, ny)) {
154             break;
155           }
156         }
157       }
158     }
159
160
161   } else if (board[x][y] == 'K' || board[x][y] == 'k') {
162     // king
163
164     for (int dx = -1; dx <= 1; dx++) {
165       for (int dy = -1; dy <= 1; dy++) {
166         if (dx == 0 && dy == 0)
167           continue;
168
169         int nx = x + dx,
170           ny = y + dy;
171
172         if (is_valid(nx, ny) && (iz_empty(nx, ny) ||
173             is_white(x, y) != is_white(nx, ny))) {
174           res.push_back(ii(nx, ny));
175         }
176       }
177     }
178   } else {
179     assert(false);
180   }
181
182   return res;
183 }
184
185 bool is_mate() {
186
187   bool can_escape = false;
188
189   char new_board[8][8];
190
191   for (int x = 0; !can_escape && x < 8; x++) {
192     for (int y = 0; !can_escape && y < 8; y++) {
193       if (!iz_empty(x, y) && !is_white(x, y)) {
194
195         vii moves = next(x, y);
196         for (int i = 0; i < size(moves); i++) {
197           for (int j = 0; j < 8; j++)
198             for (int k = 0; k < 8; k++)
199               new_board[j][k] = board[j][k];
200
201           new_board[moves[i].first][moves[i].second] = board[x][y];
202           new_board[x][y] = '.';
203
204           swap(new_board, board);
205
206
207           bool is_killed = false;
208           for (int j = 0; !is_killed && j < 8; j++) {
```

```
209             for (int k = 0; !is_killed && k < 8; k++) {
210               if (!iz_empty(j, k) && is_white(j, k)) {
211                 vii nxts = next(j, k);
212
213                 for (int l = 0; l < size(nxts); l++) {
214                   if (board[nxts[l].first][nxts[l].second] == 'k') {
215                     is_killed = true;
216                     break;
217                   }
218                 }
219               }
220             }
221           }
222
223           swap(new_board, board);
224
225           if (!is_killed) {
226             can_escape = true;
227             break;
228           }
229         }
230
231       }
232     }
233   }
234
235   return !can_escape;
236 }
237
238 int main()
239 {
240   for (int i = 0; i < 8; i++) {
241     for (int j = 0; j < 8; j++) {
242       scanf("%c", &board[i][j]);
243     }
244
245     scanf("\n");
246   }
247
248   char new_board[8][8];
249   for (int x = 0; x < 8; x++) {
250     for (int y = 0; y < 8; y++) {
251       if (!iz_empty(x, y) && is_white(x, y)) {
252
253         vii moves = next(x, y);
254
255         for (int i = 0; i < size(moves); i++) {
256
257           for (int j = 0; j < 8; j++)
258             for (int k = 0; k < 8; k++)
259               new_board[j][k] = board[j][k];
260
261           new_board[moves[i].first][moves[i].second] = board[x][y];
262           new_board[x][y] = '.';
263
264           swap(new_board, board);
265
266
267           if (board[moves[i].first][moves[i].second] == 'P' &&
268               moves[i].first == 0) {
269
270             board[moves[i].first][moves[i].second] = 'Q';
271             if (is_mate()) {
272               printf("%c%d%c%d\n", y + 'a', 7 - x + 1,
273                       moves[i].second + 'a', 7 - moves[i].first + 1);
274               return 0;
275             }
276
277             board[moves[i].first][moves[i].second] = 'N';
278             if (is_mate()) {
279               printf("%c%d%c%d\n", y + 'a', 7 - x + 1,
280                       moves[i].second + 'a', 7 - moves[i].first + 1);
281               return 0;
282             }
283
284           } else {
285             if (is_mate()) {
286               printf("%c%d%c%d\n", y + 'a', 7 - x + 1,
```

```
287                     moves[i].second + 'a', 7 - moves[i].first + 1);
288               return 0;
289             }
290           }
291
292           swap(new_board, board);
293         }
294       }
295     }
296   }
297
298   assert(false);
299
300   return 0;
301 }
```

That is a lot of code! Note how there are a few obvious mistakes which makes the code harder to read, such as typo of `iz_empty` instead of `is_empty`, or how the list of moves for the knight is called `rook`. Our final solution reduces this to less than half the size.

Exercise 4.1. Read through the above code carefully and consider if there are better ways to solve the problem. Furthermore, it has a bug – can you find it?

First, let us clean up the move generation a bit. Currently, it is implemented as the function `next`, together with some auxiliary data (lines 25-179). It is not particularly abstract, plagued by a lot of code duplication.

The move generation does not need a lot of code. Almost all the moves of the pieces can be described in the same way, as: "pick a direction out of a list D and move at most L steps along this direction, stopping either before exiting the board or taking your own piece, or when taking another piece". For the king and queen, D is all 8 directions one step away, with $L = 1$ for the king and $L = \infty$ for the queen.

Implementing abstraction is done with little code.

```
1  const vii DIAGONAL = {{-1, 1}, {-1, 1}, {1, -1}, {1, 1}};
2  const vii CROSS = {{0, -1}, {0, 1}, {-1, 0}, {1, 0}};
3  const vii ALL_MOVES = {{-1, 1}, {-1, 1}, {1, -1}, {1, 1},
4    {0, -1}, {0, 1}, {-1, 0}, {1, 0}};
5  const vii KNIGHT = {{-1, -2}, {-1, 2}, {1, -2}, {1, 2},
6      {-2, -1}, {-2, 1}, {2, -1}, {2, 1}};
7  vii directionMoves(const vii& D, int L, int x, int y) {
8    vii moves;
9    trav(dir, D) {
10     rep(i,1,L+1) {
11       int nx = x + dir.first * i, ny = y + dir.second * i;
12       if (!isValid(nx, ny)) break;
13       if (isEmpty(nx, ny)) moves.emplace_back(nx, ny);
14       else {
15         if (isWhite(x, y) != isWhite(nx, ny)) moves.emplace_back(nx, ny);
16         break;
17       }
18     }
19   }
20   return moves;
21 }
```

A short and sweet abstraction, that will prove very useful. It handles all possible moves, except for pawns. These have a few special cases.

```
vii pawnMoves(int x, int y) {
  vii moves;
  if (x == 0 || x == 7) {
    vii queenMoves = directionMoves(ALL_MOVES, 16, x, y);
    vii knightMoves = directionMoves(KNIGHT, 1, x, y);
    queenMoves.insert(queenMoves.begin(), all(knightMoves));
    return queenMoves;
  }
  int mv = (isWhite(x, y) ? - 1 : 1);
  if (isValid(x + mv, y) && isEmpty(x + mv, y))  {
    moves.emplace_back(x + mv, y);
    bool canMoveTwice = (isWhite(x, y) ? x == 6 : x == 1);
    if (canMoveTwice && isValid(x + 2 * mv, y) && isEmpty(x + 2 * mv, y)) {
      moves.emplace_back(x + 2 * mv, y);
    }
  }
  auto take = [&](int nx, int ny) {
    if (isValid(nx, ny) && !isEmpty(nx, ny)
      && isWhite(x, y) != isWhite(nx, ny))
      moves.emplace_back(nx, ny);
  };
  take(x + mv, y - 1);
  take(x + mv, y + 1);
  return moves;
}
```

This pawn implementation also takes care of promotion, rendering the logic previously implementing this obsolete.

The remainder of the move generation is now implemented as:

```
vii next(int x, int y) {
  vii moves;
  switch(toupper(board[x][y])) {
    case 'Q': return directionMoves(ALL_MOVES, 16, x, y);
    case 'R': return directionMoves(CROSS, 16, x, y);
    case 'B': return directionMoves(DIAGONAL, 16, x, y);
    case 'N': return directionMoves(KNIGHT, 1, x, y);
    case 'K': return directionMoves(ALL_MOVES, 1, x, y);
    case 'P': return pawnMoves(x, y);
  }
  return moves;
}
```

These functions make up a total of about 50 lines – a reduction to a third of how the move generation was implemented before. The trick was to rework all code duplication into a much cleaner abstraction.

We also have a lot of code duplication in the `main` (lines 234-296) and `is_mate` (lines 181-232) functions. Both functions loop over all possible moves, with lots of duplication. First of all, let us further abstract the move generation to not only generate the moves a

certain piece can make, but all the moves a player can make. This is done in both functions, so we should be able to extract this logic into only one place:

```
vector<pair<ii, ii>> getMoves(bool white) {
  vector<pair<ii, ii>> allMoves;
  rep(x,0,8) rep(y,0,8) if (!isEmpty(x, y) && isWhite(x, y) == white) {
      vii moves = next(x, y);
      trav(it, moves) allMoves.emplace_back(ii{x, y}, it);
  }
  return allMoves;
}
```

We also have some duplication in the code *making* the moves. Before extracting this logic, we will change the structure used to represent the board. A `char[8][8]` is a tedious structure to work with. It is not easily copied or sent as parameter. Instead, we use a `vector<string>`, typedefed as `Board`:

```
typedef vector<string> Board;
```

We then add a function to make a move, returning a new board:

```
Board doMove(pair<ii, ii> mv) {
  Board newBoard = board;
  ii from = mv.first, to = mv.second;
  newBoard[to.first][to.second] = newBoard[from.first][from.second];
  newBoard[from.first][from.second] = '.';
  return newBoard;
}
```

Hmm... there should be one more thing in common between the `main` and `is_mate` functions. Namely, to check if the current player is in check after a move. However, it seems this is not done in the `main` function – a bug. Since we do need to do this twice, it should probably be its own function:

```
bool inCheck(bool white) {
  trav(mv, getMoves(!white)) {
    ii to = mv.second;
    if (!isEmpty(to.first, to.second)
        && isWhite(to.first, to.second) == white
        && toupper(board[to.first][to.second]) == 'K') {
      return true;
    }
  }
  return false;
}
```

Now, the long `is_mate` function is much shorter and readable, thanks to our refactoring:

```
bool isMate() {
  if (!inCheck(false)) return false;
  Board oldBoard = board;
  trav(mv, getMoves(false)) {
```

```
    board = doMove(mv);
    if (!inCheck(false)) return false;
    board = oldBoard;
  }
  return true;
}
```

A similar transformation is now possible of the `main` function, that loops over all moves white make and checks if black is in mate:

```
int main() {
  rep(i,0,8) {
      string row;
      cin >> row;
      board.push_back(row);
  }
  Board oldBoard = board;
  trav(mv, getMoves(true)) {
      board = doMove(mv);
      if (!inCheck(true) && isMate()) {
          outputSquare(mv.first.first, mv.first.second);
          outputSquare(mv.second.first, mv.second.second);
          cout << endl;
          break;
      }
  }
  return 0;
}
```

Now, we have actually rewritten the entire solution. From the 300-line behemoth with gigantic functions, we have refactored the solution into a few, short functions with are easy to follow. The rewritten solution is less than half the size, clocking in at less than 140 lines (the author's own solution is 120 lines). Learning to code such structured solutions comes to a large extent from experience. During a competition, we might not spend time thinking about how to structure our solutions, instead focusing on getting it done as soon as possible. However, spending 1-2 minutes thinking about how to best implement a complex solution could pay off not only in faster implementation times (such as halving the size of the program) but also in being less buggy.

To sum up: implementation problems should not be underestimated in terms of implementation complexity. Work on your coding best practices and spend time practicing coding complex solutions and you will see your implementation performance improve.

Additional Exercises

Problem 4.2.

Flexible Spaces	flexible
Permutation Encryption	permutationencryption
Jury Jeopardy	juryjeopardy

Fun House	funhouse
Settlers of Catan	settlers2
Cross	cross
BASIC Interpreter	basicinterpreter
Cat Coat Colors	catcoat

Notes

There are many good resources to help you become proficient at writing readable and simple code. *Clean Code* [32] describes many principles that helps in writing better code. It includes good walk-throughs on refactoring, and shows in a very tangible fashion how coding cleanly also makes coding easier.

Code Complete [33] is a huge tome on improving your programming skills. While much of the content is not particularly relevant to coding algorithmic problems, chapters 5-19 give many suggestions on coding style.

Different languages have different best practices. Some resources on improving your skills in whatever language you code in are:

C++ *Effective C++* [35], *Effective Modern C++* [36], *Effective STL* [34], by Scott Meyers,

Java *Effective Java* [7] by Joshua Bloch,

Python *Effective Python* [46] by Brett Slatkin, *Python Cookbook* [4] by David Beazley and Brian K. Jones.

Chapter 5

Time Complexity

How can you know if your algorithm is fast enough before coding it? In this chapter we examine this question from the perspective of ***time complexity***, the tool of choice in algorithm analysis to determine how fast an algorithm is.

We start our study of complexity by looking at a new sorting algorithm – ***insertion sort***. Just like selection sort (studied in Chapter 1), insertion sort works by sorting the sequence one element at a time.

5.1 The Complexity of Insertion Sort

Insertion sort iteratively ensure that all of the first i elements of the input sequence are sorted – first for $i = 1$, then for $i = 2$, etc, up to $i = n$, at which point the entire sequence is sorted.

Insertion Sort

We want to sort the list $a_0, a_1, ..., a_{N-1}$ of N integers. If we know that the first K elements $a_0, ..., a_{K-1}$ are sorted, we can make the list $a_0, ..., a_K$ sorted by taking the element a_K and inserting it into the correct position in the already-sorted prefix $a_0, ..., a_{K-1}$.

For example, we know that a list of a single element is always sorted, so we can use that a_0 is sorted as a base case. We can sort a_0, a_1 by checking whether a_1 should be to the left or to the right of a_0. In the first case, we swap the two numbers.

Once we have sorted a_0, a_1, we insert a_2 into the sorted list. If it is larger than a_1, it is already in the correct place. Otherwise, we swap a_1 and a_2, and keep going until we either find the correct location, or determine that the number was the smallest one in which case the correct location is in the beginning.

This procedure is then repeated for every remaining element. ■

In this section we determine how long time insertion sort takes to run. When analyzing an algorithm we do not to compute the actual wall clock time an algorithm takes. Indeed, this would be nearly impossible a priori – modern computers are complex beasts with often unpredictable behavior. Instead, we try to approximate the *growth* of the running time, as a function of the size of the input.

Competitive Tip

While it is difficult to measure the exact wall-clock time of your algorithm just by analyzing the algorithm and code, it is sometimes a good idea to benchmark your solution before submitting it to the judge. This way you trade a few minutes of time (constructing the worst-case input) for avoiding many time limit exceeded verdicts. If you are unsure of your solution and the competition format penalizes you for rejected submissions, this trade-off can have good value.

When sorting fixed-size integers the size of the input would be the number of elements we are sorting, N. We denote the time that the algorithm takes in relation to N as $T(N)$. Since an algorithm often has different behaviors depending on how an instance is constructed, this is taken to be the *worst-case* time, over every instance of N elements.

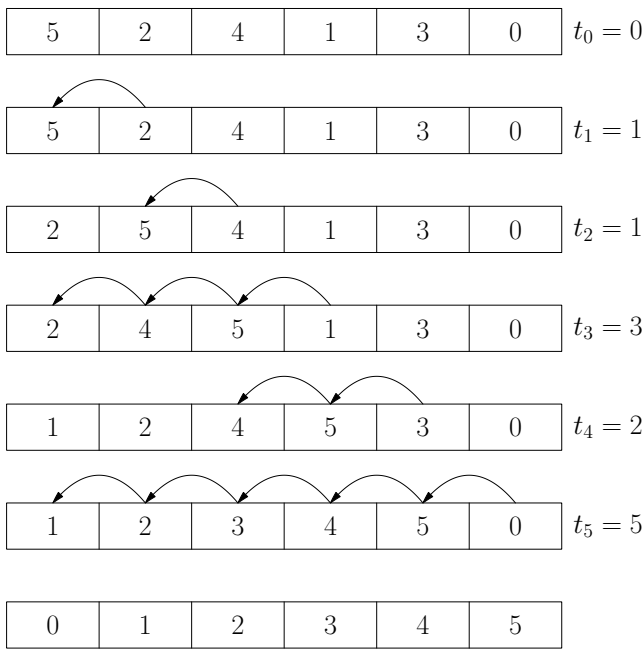


Figure 5.1: Insertion sort sorting the sequence $5, 2, 4, 1, 3, 0$.

To properly analyze an algorithm, we need to be more precise about exactly what it does. We give the following pseudo code for insertion sort:

```
1: procedure InsertionSort(A)
2:     for i ← 0 to N − 1 do
3:         j ← i
4:         while j > 0 and A[j] < A[j − 1] do
```

```
5:          Swap A[j] and A[j − 1]
6:          j ← j − 1
```

To analyze the running time of the algorithm, we make the assumption that any "sufficiently small" operation takes the same amount of time – exactly 1 (of some undefined unit). We have to be careful in what assumptions we make regarding what a sufficiently small operation means. For example, sorting N numbers is not a small operation, while adding or multiplying two fixed-size numbers is. Multiplication of integers of arbitrary size is not a small operation (see the Karatsuba algorithm, Section **??**).

In our program every line happens to represent a small operation. However, the two loops may cause some lines to execute more than once. The outer for loop executes N times. The number of times the inner loop runs depends on how the input looks. We let t_i mean the number of iterations the inner loop runs during the i'th iteration of the outer loop. These are included in Figure 5.1 for every iteration.

Now we can annotate our pseudo code with the number of times each line executes.

```
1: procedure InsertionSort(A)          ▷ Sorts the sequence A containing N elements
2:     for i ← 0 to N − 1 do           ▷ Runs N times, cost 1
3:         j ← i                       ▷ Runs N times, cost 1
4:         while j > 0 and A[j] < A[j − 1] do   ▷ Runs ∑_{i=0}^{N−1} t_i times, cost 1
5:             Swap A[j] och A[j − 1]  ▷ Runs ∑_{i=0}^{N−1} t_i times, cost 1
6:             j ← j − 1               ▷ Runs ∑_{i=0}^{N−1} t_i times, cost 1
```

$T(N)$ can be expressed as

$$T(N) = N + N + \left(\sum_{i=0}^{N-1} t_i\right) + \left(\sum_{i=0}^{N-1} t_i\right) + \left(\sum_{i=0}^{N-1} t_i\right) = 3\left(\sum_{i=0}^{N-1} t_i\right) + 2N.$$

We still have some t_i variables left so we do not truly have a function of N. We can eliminate this by realizing that in the worst case $t_i = i$. This occurs when the list we are sorting is in descending order. Each element must then be moved to the front, requiring i swaps for the i'th element.

With this substitution we can simplify the expression:

$$T(N) = 3\left(\sum_{i=0}^{N-1} i\right) + 2N = 3\frac{(N-1)N}{2} + 2N = \frac{3}{2}(N^2 - N) + 2N$$

$$= \frac{3}{2}N^2 + \frac{N}{2}$$

This function grows quadratically with the number of elements of N. The approximate growth of the time a function takes has an important role in algorithm analysis, so a special notation was developed for it.

5.2 Asymptotic Notation

We almost always express the running time of an algorithm in what is called ***asymptotic notation***. The notation captures the behavior of a function as its arguments grow. For example, the function $T(N) = \frac{3}{2}N^2 + \frac{N}{2}$ that described the running time of insertion sort, is bounded by $c \cdot N^2$ for large N, for some constant c. We write

$$T(N) = O(N^2)$$

when this is the case.

Similarly, the linear function $2N + 15$ is bounded by $c \cdot N$ for large N, with $c = 3$, so $2N + 15 = O(N)$. The notation only captures upper bounds though (and not the actual rate of growth). We could therefore say that $2N + 15 = O(N^2)$, even though this particular upper bound is very lax. However, N^2 is not bounded by $c \cdot N$ for any constant c when N is large, so $N^2 \neq O(N)$. This also corresponds to intuition – quadratic functions grows faster than linear functions.

Definition 5.1 — O-notation
Let f and g be functions from $\mathbb{R}_{\geq 0}$ to $\mathbb{R}_{\geq 0}$. If there exists positive constants n_0 and c such that $f(n) \leq cg(n)$ whenever $n \geq n_0$, we say that $f(n) = O(g(n))$.

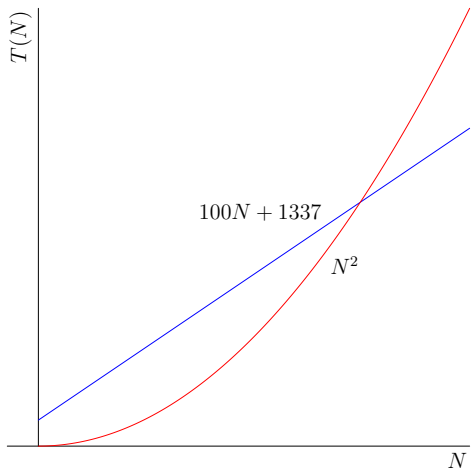


Figure 5.2: All linear functions are eventually outgrown by N^2, so $an + b = O(n^2)$.

Intuitively, the notation means that $f(n)$ grows *slower than or as fast as* $g(n)$, within a constant factor. Any quadratic function $an^2 + bn + c = O(n^2)$. Similarly, any linear function $an + b = O(n^2)$ as well. The definition implies that for two functions f and g which are always within a constant factor of each other, we have that both $f(n) = O(g(n))$ and $g(n) = O(f(n))$.

We can use this definition to prove that the running time of insertion sort is $O(N^2)$, even in the worst case.

Example 5.1 Prove that $\frac{3}{2}N^2 + \frac{N}{2} = O(N^2)$.

Proof. When $N \geq 1$ we have $N^2 \geq N$ (by multiplying both sides with N). This means that

$$\frac{3}{2}N^2 + \frac{N}{2} \leq \frac{3}{2}N^2 + \frac{N^2}{2} = \frac{4}{2}N^2 = 2N^2$$

for $N \geq 1$. Using the constants $c = 2$ and $n_0 = 1$ we fulfill the condition from the definition. □

For constants k we say that $k = O(1)$. This is a slight abuse of notation, since neither k nor 1 are functions, but it is a well-established abuse. If you prefer, you can instead assume we are talking about functions $k(N) = k$ and $1(N) = 1$.

Competitive Tip

The following table describes approximately what complexity you need to solve a problem of size n if your algorithm has a certain complexity when the time limit is about 1 second.

Complexity	n
$O(\log n)$	$2^{(10^7)}$
$O(\sqrt{n})$	10^{14}
$O(n)$	10^7
$O(n \log n)$	10^6
$O(n\sqrt{n})$	10^5
$O(n^2)$	$5 \cdot 10^3$
$O(n^2 \log n)$	$2 \cdot 10^3$
$O(n^3)$	300
$O(2^n)$	24
$O(n2^n)$	20
$O(n^2 2^n)$	17
$O(n!)$	11

Table 5.1: Approximations of needed time complexities

Note that this is in no way a general rule – while complexity does not bother about constant factors, wall clock time does!

Complexity analysis can also be used to determine *lower bounds* on the time an algorithm takes. To reason about lower bounds we use Ω-notation. It is similar to O-notation except it describes the reverse relation.

Definition 5.2 — Ω-notation
Let f and g be non-negative functions from $\mathbb{R}_{\geq 0}$ to $\mathbb{R}_{\geq 0}$. If there exists positive constants n_0 and c such that $cg(n) \leq f(n)$ for every $n \geq n_0$, we say that $f(n) = \Omega(g(n))$.

We know that the complexity of insertion sort has an upper bound of $O(N^2)$ in the worst-case, but does it have a lower bound? It actually has the same lower bound as upper bound, i.e. $T(N) = \Omega(N^2)$.

Example 5.2 Prove that $\frac{3}{2}N^2 + \frac{N}{2} = \Omega(N^2)$.

Proof. When $N \geq 1$, we have $\frac{3}{2}N^2 + \frac{N}{2} \geq N^2$. Using the constants $c = 1$ and $n_0 = 1$ we fulfill the condition from the definition. □

In this case, both the lower and the upper bound on the worst-case running time of insertion sort coincided (asymptotically). We have another notation for when this is the case:

Definition 5.3 — Θ-notation
If $f(n) = O(g(n))$ and $f(n) = \Omega(g(n))$, we say that $f(n) = \Theta(g(n))$.

Thus, the worst-case running time for insertion sort is $\Theta(n^2)$.

There are many ways of computing the time complexity of an algorithm. The most common case is when a program has K nested loops, each of which performs $O(M)$ iterations. The complexity of these loops are then $O(M^K \cdot f(N))$ if the inner-most operation takes $O(f(N))$ time. In Chapter 7 we study the time complexity of some so-called *recursive* functions, and in Chapter 12 we see some ways of computing the time complexity of specific type of recursive solution called divide and conquer algorithms.

Exercise 5.1. Find a lower and an upper bound that coincide for the best-case running time for insertion sort.

Exercise 5.2. Give a $\Theta(n)$ algorithm and a $\Theta(1)$ algorithm to compute the sum of the n first integers.

Exercise 5.3. Prove, using the definition, that $10n^2 + 7n - 5 + \log^2 n = O(n^2)$. What constants c, n_0 did you get?

Exercise 5.4. Prove that $f(n) + g(n) = \Theta(\max\{f(n), g(n)\})$ for non-negative functions f and g.

Exercise 5.5. Determine whether, with proof:

1. Is $2^{n+1} = O(2^n)$?
2. Is $2^{2n} = O(2^n)$?

Exercise 5.6. Prove that $(n + a)^b = \Theta(n^b)$ for positive constants a, b.

Amortized Complexity

Consider the following algorithm that counts the number of times a value v appears in a vector A.

```
1:  procedure CountOccurances(A, v)
2:      N ← length of A
3:      i ← 0
4:      ans ← 0
5:      while i ≠ N do
6:          while i < N and A[i] ≠ v do
7:              i ← i + 1
8:          if i ≠ N then
9:              ans ← ans + 1
10:             i ← i + 1
```

It computes the number of occurrences of a value in a sequence (in a somewhat cumbersome way). We repeatedly scan through the sequence A to find the next occurrence of v. Whenever we find one, we increase the answer by 1 and resume our scanning. This procedure continues until we have scanned through the entire sequence.

What is the time complexity of this procedure? How do we handle the fact that neither the outer nor the inner loop of the algorithm repeats a predictable number of iterations each time? For example, the other loop would iterate N times if every element equals v, but only once if it does not contain the value at all. Similarly, the number of iterations of the inner loop depends on the *position* of the elements in A which equals v.

We find help in a technique called ***amortized complexity***. The concept of amortization is best known from loans. It represents the concept of periodically paying of some kind of debt until it has been paid off in full.

When analyzing algorithms, the "debt" is the running time of the algorithm. We try to prove that the algorithm has a given running time by looking at many, possibly uneven parts that together over the long run sums up to the total running time, similarly to how a loan can be paid of by amortization.

A quick analysis of the counting algorithm gives us a good guess that the algorithm runs in time $\Theta(N)$. It should be clear that it's inner loop at lines 6-7 that dominates the running time of the algorithm. The question is how we compute the number of times it executes even though we do not know how many times it executes nor how many iterations each execution takes. We try the amortization trick by looking at how many iterations it performs over all those executions, no matter how many they are. Assume that the loop is run k times (including the final time when the condition first is false) and each run iterates b_i times ($1 \leq i \leq k$). We claim that

$$b_1 + b_2 + \cdots + b_k = \Theta(N).$$

Our reasoning is as follows. There are two ways the variable i can increase. It can either be increased inside the loop at line 7, or at line 10. If the loop executes N times in total, it will certainly complete and never be executed again since the loop at line 5 completes too. This gives us $\sum_{i=1}^{k} b_i = O(N)$.

On the other hand, we get one iteration for every time i is increased. If i is increased on line 7, it was done within a loop iteration. If i is increased on line 9, we instead count the final check if the loop just before it once. Each addition of i happens together with an iteration of the loop, so $\sum_{i=1}^{k} b_i = \Omega(N)$. Together, these two results prove our claim.

This particular application of amortized complexity is called the *aggregate method*.

Exercise 5.7. Consider the following method of adding 1 to a binary integer represented by a list of its digits.

```
1: procedure BinaryIncrement(D)
2:     i ← 0
3:     while D[i] = 1 do                          ▷ Add 1 to the i'th digit
4:         D[i] = 0          ▷ We add 1 to a 1 digit, resulting in a 0 digit plus a carry
5:         i ← i + 1
6:     D[i] = 1                                   ▷ We add 1 to a digit not resulting in a carry
```

The algorithm is the binary version of the normal addition algorithm where the two addends are written above each other and each resulting digit is computed one at a time, possibly with a carry digit. What is the amortized complexity of this procedure over 2^n calls, if D starts out as 0?

5.3 NP-complete Problems

Of particular importance in computer science are the problems that can be solved by algorithms running in polynomial time, i.e. in $O(n^c)$ time for some constant $c > 0$, often considered to be the "tractable" problems. For some problems we do not yet know if there is an algorithm whose time complexity is bounded by a polynomial. One particular class of these are the ***NP-complete problems***. They have the property that they are all reducible to one another, in the sense that a polynomial-time algorithm to any one of them yields a polynomial-time algorithm to all the others. Many of these NP-complete problems (or problems reducible to such a problem, a property called ***NP-hardness***) appear in algorithmic problem solving. It is good to know that they exist and that it is unlikely you will find a polynomial-time solution. During the course of this book, you will occasionally see such problems with their NP-completeness mentioned.

5.4 Other Types of Complexities

There are several other types of complexities aside from the time complexity. For example, the ***memory complexity*** of an algorithm measures the amount of memory it uses. We

use the same asymptotic notation when analyzing it. In most modern programming competitions, the allowed memory usage is high enough for the memory complexity not be an issue – if you get memory limit problems you also tend to have time limit problems. However, it is still of interest in computer science (and thus algorithmic problem solving) and computer engineering in general.

Another common type of complexity is the ***query complexity***. In some problems (like the guessing problem from Chapter 1), we are given access to some kind of external procedure (called an *oracle*) that computes some value given parameters that we provide. A procedure call of this kind is called a *query*. The number of queries that an algorithm makes to the oracle is called its query complexity. Problems where the algorithm is allowed access to an oracle often bound the number of queries the algorithm may make. In these problems the query complexity of the algorithm is of interest.

5.5 The Importance of Constant Factors

In this chapter, we have essentially told you not to worry about the constant factors that the Θ notation hides from you. While this is true when you are solving problems in theory, only attempting to get a good asymptotic time complexity, constant factors can unfortunately be of large importance when implementing the problems subject to time limits.

Speeding up a program that has the correct time complexity but still gets *time limit exceeded* when submitted to an online judge is half art and half engineering. The first trick is usually to generate the worst-case test instance. It is often enough to create a test case where the input matches the input limits of the problem, but sometimes your program behaves differently depending on how the input looks. In these cases, more complex reasoning may be required.

Once the worst-case instance has been generated, what remains is improving your code until you have gained a satisfactory decrease in time usage. When doing this, focus should be placed on those segments of your code that takes the longest wall-clock time. Decreasing time usage by 10% in a segment that takes 1 second is clearly a larger win than decreasing time usage by 100% in a segment that takes 0.01 seconds.

There are many tricks to improving your constant factors, such as:

- using symmetry to perform less calculations,
- precomputing oft-repeated expressions, especially involving trigonometric functions,
- passing very large data structures by reference instead of by copying,
- avoiding to repeatedly allocate large amounts of memory, and
- using SIMD (single instruction, multiple data) instructions.

Additional Exercises

Exercise 5.8. Prove that if $a(x) = O(b(x))$ and $b(x) = O(c(x))$, then $a(x) = O(c(x))$, i.e. asymptotic notation is a *transitive* property. This means functions can be partially ordered by their asymptotic growth.

Exercise 5.9. Order the following functions by their asymptotic growth *with proof*!

$$x \quad \sqrt{x} \quad x^2 \quad 2^x \quad e^x$$

$$x! \quad \log x \quad \frac{1}{x} \quad x \log x \quad x^3$$

Exercise 5.10.

1. Prove that if $a(x) = O(b(x))$ and $c(x) = O(d(x))$, then

$$a(x) + c(x) = O(b(x) + d(x)).$$

2. Prove that if $a(x) = O(b(x))$ and $c(x) = O(d(x))$, then

$$a(x) \cdot c(x) = O(b(x) \cdot d(x)).$$

Notes

Advanced algorithm analysis sometimes uses complicated discrete mathematics, including number theoretical (as in Chapter 17) or combinatorial (as in Chapter 18) facts. *Concrete Mathematics* [29] by Donald Knuth, et al, does a thorough job on both accounts.

An Introduction to the Analysis of Algorithms [17] by Sedgewick and Flajolet has a more explicit focus on the analysis of algorithms, mainly discussing combinatorial analysis.

The study of various kinds of complexities constitute a research area called *computational complexity theory*. *Computational Complexity* [39] by Papadimitriou is a classical introduction to computational complexity, although *Computational Complexity: A Modern Approach* [2] by Arora and Barak is a more modern textbook, with recent results that the book by Papadimitriou lack.

While complexity theory is mainly concerned about the limits of specific computational models on problems that *can* be solved within those models, what can not be done by computers is also interesting. This is somewhat out of scope for an algorithmic problem solving book, but is still of general interest. A book on automata theory (such as *Introduction to Automata Theory, Languages, and Computation* [53] by Ullman et al) can be a good compromise, mixing both foundations of the theory of computation with topics more applicable to algorithms.

Chapter 6

Data Structures

Solutions to algorithmic problems consist of two constructs – algorithms and ***data structures***. Data structures are used to organize the data that algorithms operate on. We encountered several structures in Chapters 2-3 when learning C++, like the vector.

Many data structures have been developed to handle common operations we may need to perform on data quickly. In this chapter we discuss some of the basic data structures used in programming. We chose an approach that is perhaps more theoretic than many other problem solving texts when it comes to the basic structures. In particular, we explicitly discuss not only the data structures themselves and their complexities, but also their implementations. We do this mainly because we believe that their implementations show useful algorithmic techniques. While you may feel that you can simply skip this chapter if you are familiar with all the data structures, we advise that you still read through the sections for those structures that you lack confidence in implementing.

6.1 Dynamic Arrays

The most basic data structure is the ***fixed-size array***. It consists of a fixed size block of memory and can be viewed as a sequence of N variables of the same type T. It supports the operations:

- T*arr*[] = newT[*size*]: creating a new array, with a given size.
  Complexity: $\Theta(1)$[1]
- delete[]*arr*: deleting an existing array.
  Complexity: $\Theta(1)$
- *arr*[*index*]: accessing the value in a certain access a value.
  Complexity: $\Theta(1)$

In Chapter 2 we saw how to create fixed-size arrays the size was known beforehand. In C++ we can create fixed-size arrays using an expression as size instead. This is done using the syntax above, for example:

[1] This complexity is debatable, and highly dependent on what computational model one uses. In practice, this is roughly "constant time" in most memory management libraries used in C++. In all Java and Python implementations we tried, it is instead linear in the size.

```
int size = 5;
int *arr = new int[size];
arr[2] = 5;
cout << arr[2] << endl;
delete[] arr;
```

In C++, a variable that represents a fixed-size array with an unknown size have an asterisk in front of its name **when it is declared**, like `int *arr`, to distinguish it from a normal array or variable. Other languages, such as Python or Java, don't have different arrays these purposes[2].

Exercise 6.1. What happens if you try to create an array with a negative size?

The fixed-size array can be used to implement a more useful data structure, the ***dynamic array***. This is an array that can change size when needed. For example, we may want to repeatedly insert values in the array without knowing the total number of values beforehand. This is a very common requirement in programming problems. Specifically, we want to support two additional operations in addition to the operations supported by the fixed-size array.

- insert(pos, val): inserting a value in the array at a given position.
  Amortized complexity: $\Theta(size - pos)$
  Worst case complexity: $\Theta(size)$
- remove(pos): erase a position in an array.
  Complexity: $\Theta(size - pos)$

The complexities we list above are a result of the usual implementation of the dynamic array. A key consequence of these complexities is that addition and removal of elements to the end of the dynamic array takes $\Theta(1)$ amortized time.

A dynamic array can be implemented in numerous ways, and underlies the implementation of essentially every other data structure that we use. A naive implementation of a dynamic array is using a fixed-size array to store the data and creating a new one with the correct size whenever we add or remove an element, copying the elements from the old array to the new array. The complexity for this approach is linear in the size of the array for every operation that changes the size since we need to copy $\Theta(size)$ elements during every update. We need to do better than this.

To achieve the targeted complexity, we can modify this naive approach by not creating a new array *every* time we have to change the size of the dynamic array. Whenever we need to increase the size of the dynamic array, we create a fixed-size array that is *larger* than we actually need it to be. For example, if we create a fixed-size array with n more elements than our dynamic array needs to store, we don't have to increase the size of the backing fixed-size array until we have added n more elements to the dynamic array. This means

[2]The difference has to do where the array is stored in memory, an aspect C++ lets us control explicitly.

that a dynamic array does not only have a size, the number of elements we currently store in it, but also a capacity, the number of elements we *could* store in it. See Figure 6.1 for a concrete example of what happens when we add elements to a dynamic array that is both within its capacity and when we exceed it.

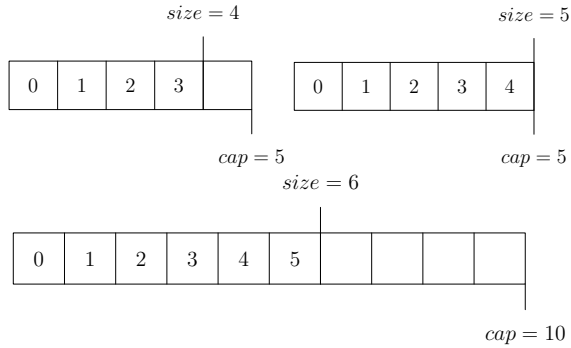


Figure 6.1: The resizing of an array when it overflows it capacity.

To implement a dynamic array in C++, we could use a structure storing as members the capacity, size and backing fixed-size array of the dynamic array. An example of how such a structure could look can be found in Snippet 6.1.

Snippet 6.1: The dynamic array structure

```
1  struct DynamicArray {
2    int capacity;
3    int size;
4    int *backing;
5
6    DynamicArray() {
7      capacity = 10;
8      size = 0;
9      backing = new int[10];
10   }
11
12   ~DynamicArray() {
13     delete[] backing;
14   }
15 };
```

The structure has an extra member function that looks like a constructor with a ~ prepended to its name. This is a *destructor*, invoked when the array is destroyed. In it, we delete the backing array in order to not waste the memory it used.

We are almost ready to add and remove elements to our array now. First, we need to handle the case where insertion of a new element would result in the size of the dynamic array would exceed its capacity, that is when *size* = *capacity*. Our previous suggestion was

to allocate a new, bigger one, but just how big? If we always add, say, 10 new elements to the capacity, we have to perform the copying of the old elements with every 10'th addition. This still results in additions to the end of the array taking linear time on average. There is a neat trick that avoids this problem: creating the new backing array with **double** the current capacity.

This ensures that the complexity of all copying needed for an array up to some certain capacity have an amortized complexity of $\Theta(cap)$. Assume that we have just increased the capacity of our array to *cap*, which required us to copy $\frac{cap}{2}$ elements. Then, the previous increase will have happened at around capacity $\frac{cap}{2}$ and took time $\frac{cap}{4}$. The one before that occurred at capacity $\frac{cap}{4}$ and so on.

We can sum up all of this copying:

$$\frac{cap}{2} + \frac{cap}{4} + \cdots \leq cap$$

using the formula for the sum of a geometric series.

Since each copy is assumed to take $\Theta(1)$ time, the total time to create this array was $\Theta(cap)$. As $\frac{cap}{2} \leq size \leq cap$, this is also $\Theta(size)$, meaning that adding *size* elements to the end of the dynamic array takes amortized $\Theta(size)$ time.

When implementing this in code, we use a function that takes as argument the capacity we require the dynamic array to have and ensures that the backing array have at least this size, possibly by creating a new one double in size until it is sufficiently large. Example code for this can be found in Snippet 6.2.

Snippet 6.2: Ensuring that a dynamic array have sufficient capacity

```
1  void ensureCapacity(int need) {
2    while (capacity < need) {
3      int *newBacking = new int[2 * capacity];
4      for (int i = 0; i < size; i++)
5        newBacking[i] = backing[i];
6      delete[] backing;
7      backing = newBacking;
8      capacity *= 2;
   ↪
9    }
10 }
11
```

With this method in hand, insertion and removal of elements is actually pretty simple. Whenever we remove an element, we simply need to move the elements coming after it in the dynamic array forward one step. See Figure 6.2 for an illustration of an element being removed.

When adding an element, we reverse this process by moving the elements coming after the position we wish to insert a new element at one step towards the back. An example of this is shown in Figure 6.3.

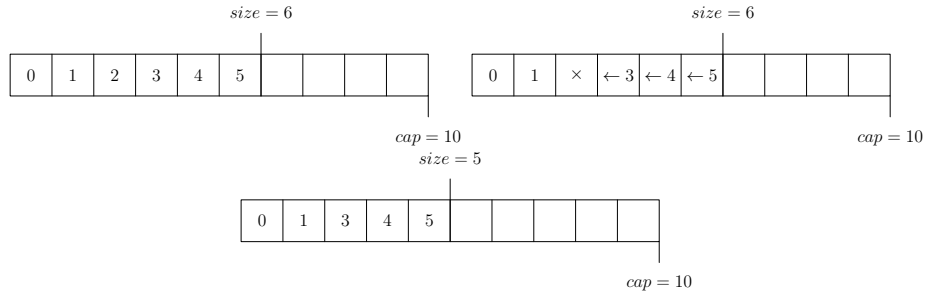


Figure 6.2: The removal of the element 2 at index 2.

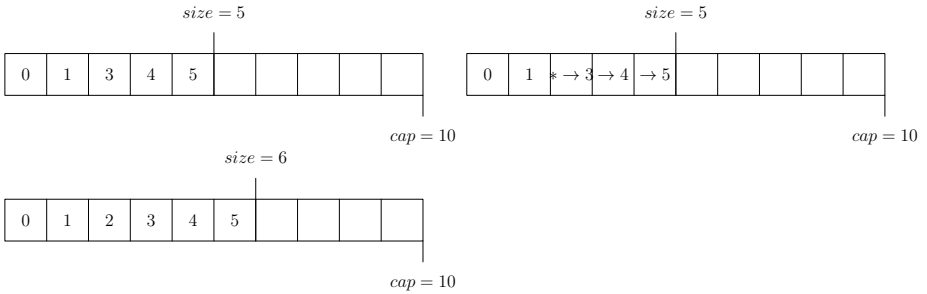


Figure 6.3: The insertion of the element 2 into index 2.

Exercise 6.2. Implement insertion and removal of elements in a dynamic array.

Dynamic arrays are called `vectors` in C++ (Section 3.1.2). They have the same complexities as the one described at the beginning of this section.

Exercise 6.3. How can *any* element be removed in $\Theta(1)$ if we ignore the ordering of values in the array?

6.2 Stacks

The ***stack*** is a data structure that contains an ordered lists of values and supports the following operations:

- push(*val*): inserting a value at the top of the stack.
  Amortized complexity: $\Theta(1)$
- pop(): remove the value at the top of the stack.
  Complexity: $\Theta(1)$
- top(): get the value at the top of the stack.
  Complexity: $\Theta(1)$

The structure is easily implemented with the above time complexities using a dynamic vector. After all, the vector supports exactly the same operations that a stack requires. In C++, the stack is called a `stack`. We did not mention it in the STL chapter since it can be so easily replaced by the vector.

Exercise 6.4. Implement a stack using a dynamic vector.

6.3 Queues

The ***queue*** is, like the vector and the stack, an ordered list of values. Instead of removing and getting values from the end like the stack, it gets the value from the *front*. The supported operations are thus:

- push(*val*): inserting a value at the end of the queue.
  Amortized complexity: $\Theta(1)$
- pop(): remove the value at the front of the queue.
  Amortized complexity: $\Theta(1)$
- front(): get the value at the front of the queue.
  Complexity: $\Theta(1)$

As previously seen, C++ has an implementation of the queue called `queue` (Section 3.1.4).

Implementing a queue can also be done using a vector. After all, the operations and complexities are nearly the same; only removing the value of the front is wrong. To fix this, one can simply hold a *pointer* to what the front of the queue is in the vector. Removing the front element is then equivalent to moving the pointer forward one step. To see this how this would work in practice, see an example push and pop operation in Figure 6.4.

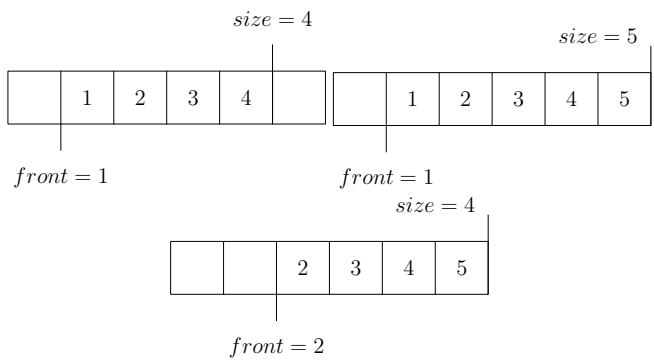


Figure 6.4: Pushing and popping elements in a queue

Exercise 6.5. Implement a queue using a vector.

Exercise 6.6. A queue can be implemented using two stacks in amortized constant time per operation. How?

Exercise 6.7. A stack can be implemented using two queues in amortized linear time per operation. How?

Exercise 6.8. This naive suggestion of a queue implementation suffers a slight problem. After pushing and poping k elements, the backing dynamic array has size at least k even though none of its elements are in use, thus occupying memory unnecessarily. Devise a strategy that ensures the backing dynamic array never uses more than cn elements (where n is the current size of the queue) for some constant c but maintaining the amortized complexity.

6.4 Priority Queues

Now, let us look at our first more complex data structure. The ***priority queue*** is an unordered bag of elements, from which we can get and remove the largest one quickly. It supports the operations

- push(*val*): inserting a value into the heap.
  Complexity: $O(\log n)$
- pop(): remove the largest value in the heap.
  Complexity: $O(\log n)$
- getMax(): get the largest value in the heap.
  Complexity: $\Theta(1)$

This is implemented as `priority_queue` in C++ (Section 3.1.5), although one often instead use a `set` which not only supports the same operations with the same complexities, but also supports erasing elements.

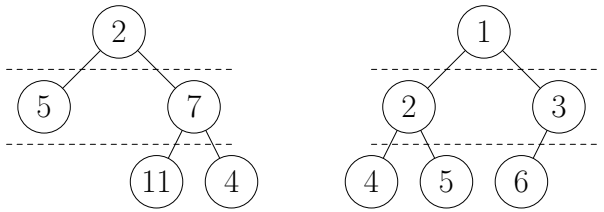
The backing implementation of the priority queue structure we study is called a *heap*. The heap itself will be implemented using another data structure called a *binary tree*, which we need to describe first.

Binary Trees

A ***binary tree*** is a data structure where there one element is designated the *root* of the tree, and each element is given either 0, 1 or 2 children. All elements except the root is a child of another element. In Figure 6.5a, you can see an example of a binary tree.

We call a binary tree *complete* if every level of the tree is completely filled, except possibly the bottom one. If the bottom level is not filled, all elements in it need to be as far as possible to the left. In a complete binary tree, we can order every element as we do in Figure 6.5b, i.e. from the top down, left to right at each layer.

The beauty of this numbering is that we can use it to store a binary tree in a vector. Since each element is given a number, we can map the number of each element into a



(a) A non-complete binary tree with 3 levels.

(b) A complete binary tree, also with 3 levels.

Figure 6.5: Examples of binary trees.

position in a vector. The n elements in a complete binary tree then occupy all the indices $[1 \ldots n]$. An benefit of this numbering is that it is easy to compute the number of the parent, left child and right child of an element. If an element has number i, the parent has number $\lfloor \frac{i}{2} \rfloor$, the left child has number $2i$ and the right child has number $2i + 1$.

Exercise 6.9. Prove that the above properties of the numbering of a complete binary tree hold.

A heap is implemented as a complete binary tree, which we in turn implement by a backing vector in the manner described. In the implementation, we use the following convenience functions:

```
1: function Parent(i) return i/2
2: function Left(i) return 2i
3: function Right(i) return 2i + 1
```

Note: if you use a vector to represent a complete binary tree in this manner it needs to have the size $n + 1$ where n is the number of elements, since the tree numbering is 1-indexed and the vector is 0-indexed!

Heaps

A ***heap*** is a special kind of complete binary tree. More specifically, it should always satisfy the following property: *an element always has a higher value than its immediate children*. Note that this condition acts transitively, which means that an element also has a higher value than its grand-children, and their children and so on. In particular, a consequence of this property is that the root of the three is always be the largest value in the heap. As it happens, this property is exactly what we need to quickly get the maximum value of the heap. You can see an example of a heap in Figure 6.6

We thus start our description of a heap somewhat backwards, with the function needed to get the largest element:

```
1: procedure Get-Max(tree) return tree[1]
```

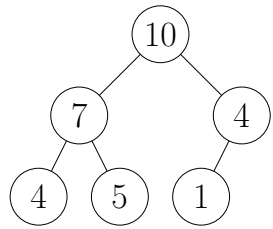


Figure 6.6: A heap of the elements $1, 4, 4, 5, 7, 10$.

The complicated operations on the heap is to add and remove elements while ensuring that the heap keeps satisfying this property. We start with the simplest one, adding a new element. Since we represent the heap using a vector, adding a new element to a heap can be done by appending the element to the vector. In this manner, we ensure that the underlying binary tree is still complete. However, it may be that the value we added is now larger than its parent. If this is the case, we can fix the violation of the heap property by swapping the value with its parent. This does not guarantee that the value still is not larger than its parent. In fact, if the newly added element is largest in the heap, it would have to be repeatedly swapped up to the top! This procedure, of moving the newly added element up in the tree until it is no longer larger than its parent (or it becomes the root) is called *bubbling up*:

```
1: procedure Bubble-Up(idx, vector tree)
2:     while idx > 1 do
3:         if tree[idx] > tree[Parent(idx)] then
4:             Swap tree[idx] and tree[Parent(idx)]
5:             idx ← Parent(idx)
6:         else
7:             break
```

Pushing a value now reduces to appending it to the tree and bubbling it up. You can see this procedure in action in Figure 6.7.

```
1: procedure Push(x, tree)
2:     tree.append(x)
3:     Bubble − Up(tree.size() − 1, tree)
```

Removing a value is slightly harder. First of, the tree will no longer be a binary tree – it is missing its root! To rectify this, we can take the last element of the tree and put it as root instead. This keeps the binary tree complete, but may cause it to violate the heap property since our new root may be smaller than either or both of its children.

The solution to this problem is similar to that of adding an element. Instead of bubbling up, we bubble it down by repeatedly swapping it with one of its children until it no longer

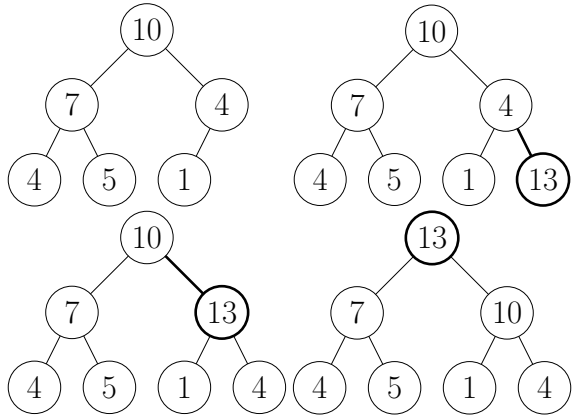


Figure 6.7: Adding a new value and bubbling it up.

is greater than any of its children. The only question mark is which of its children we should bubble down to, in case the element is smaller than both of its children. The answer is clearly the largest of the two children. If we take the smaller of the two children, we will again violate the heap property. Just as with pushing, popping a value is done by a combination of removing the value and fixing the heap to satisfy the heap property again.

```
1:  procedure Remove-Max(x, vector tree)
2:      tree[1] ← tree[tree.size() − 1]
3:      remove the last element of tree
4:      Bubble − Down(1, tree)
5:  procedure Bubble-Down(idx, tree)
6:      while true do
7:          largest ← idx
8:          if Left(idx) < tree.size() and tree[Left(idx)] > tree[largest] then
9:              largest ← Left(idx)
10:         if Right(idx) < tree.size() and tree[Right(idx)] > tree[largest] then
11:             largest ← Right(idx)
12:         if largest = idx then
13:             break
14:         else
15:             Swap tree[idx] and tree[largest]
16:             idx ← largest
```

A final piece of our analysis is missing. It is not yet proven that the time complexity of adding and removing elements are indeed $O(\log n)$. To do this, we first need to state a basic fact of complete binary trees: their height is at most $\log_2 n$. This is easily proven by contradiction. Assume that the height of the tree is at least $\log_2 n + 1$. We claim that

any such tree must have strictly more than n elements. Since all but the last layers of the tree must be complete, it must have at least $1 + 2 + \cdots + 2^{\log_2 n} = 2^{\log_2 n+1} - 1$ elements. But $2^{\log_2 n+1} - 1 = 2n - 1 > n$ for positive n, so the tree has more than n elements. This means that a tree with n elements cannot have more than height $\log_2 n$.

The next piece of the puzzle is analyzing just how many iterations the loops in the bubble up and bubble down procedures can perform. In the bubble up procedure, we keep an index to an element that, for every iteration, moves up in the tree. This can only happen as many times as there are levels in the tree. Similarly, the bubble down procedure tracks an element that moves down in the tree for every iteration. Again, this is bounded by the number of levels in the tree. We are forced to conclude that since the complexity of each iteration is $\Theta(1)$ as they only perform simple operations, the complexities of the procedures as a whole are $O(\log n)$.

Problem 6.10.
Binary Heap heap

Exercise 6.11. Prove that adding an element using Push never violates the heap property.

Exercise 6.12. To construct a heap with n elements by repeatedly adding one at a time takes $O(n \log n)$ time, since the add function takes $O(\log n)$ time in the worst case. One can also construct it in $\Theta(n)$ time in the following way: arbitrarily construct a complete binary tree with all the n elements, and then call Bubble – Down on each of the elements in reverse order $n, n-1, \ldots, 2, 1$. Prove that this correctly constructs a heap, and that is takes $\Theta(n)$ time.

6.5 Bitsets

We move on to the simplest data structure of the chapter. The ***bitset*** can be viewed as a specialization of a static-length array for the case where the values stored are booleans, i.e. supporting only the operations of setting and getting values in the array.

The idea behind it is simple. Booleans are essentially values 0 (false) or 1 (true), i.e. equivalent to a binary digit. Another data type also consists of binary digits – integers. In the chapter on programming, we said that the memory of a computer is just a long sequence of binary digits. Any interpretation of what these digits actually mean is up to us. For example, your typical 64-bit integers could just as well represent an array of 64 booleans, indexed from 0 to 63. That is the *bit* part.

An array of booleans of size N can also be interpreted as a subset of the integers $\{0, 1, \ldots, N-1\}$. If the i'th value of the array is true, we say that i is in the subset, otherwise it is not. This is the *set* part. An example of this equivalence between integers and subsets is visualized in Figure 6.8.

Interpreting N-bit integers as subsets of $\{0, 1, \ldots, N-1\}$ is surprisingly useful. It allows us to easily pass subsets as parameters to functions or use them as indexes into a vector rather than using a map.

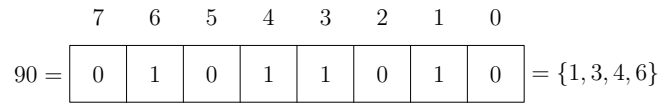


Figure 6.8: The equivalence between 90 and the set of elements $\{1, 3, 4, 6\}$.

To operate on a bitset, we use *bitwise operators* in C++. To construct the representation of the set $\{i\}$, the expression `1 << i` is used. This operator is called the ***left-shift operator***. It works by taking the binary representation of the left operand and moves it i steps to the left, adding i zeroes to the right. Since the binary representation of 1 is 1, the expression results in a binary number with an 1 in its i'th place, which is the correct representation of the set.

To take the union of two bitsets `x` and `y`, we use the ***bitwise or*** operator: `x | y`. This operator takes the two integers and gives us a new one, with a 1 as the i'th digit if either `x` or `y` had a 1 as their i'th digit. Similarly, the ***bitwise and*** operator `x & y` computes the intersection of the two sets. This allows us to check for set membership of i in the bitset `x` using the expression `x & (1 << i)` which is 0 if i was not a member of the set, and `1 << i` otherwise. The symmetric difference of two sets is computed with the ***bitwise exclusive or operator*** (often called "xor"), `x ^ y`. An element is in the symmetric difference if it was a member of exactly one of `x` and `y`. Thus, one can toggle the presence of an element i in a bitset using `x ^ (1 << i)`. Finally, the bitwise negation operator `~x` computes the complement of a set.

There are also a number of built-in functions that are of use. To compute the size of a bitset, the most common compilers support the macro `__builtin_popcount(x)` which returns the number of 1 digits in `x`. To get the index of the lowest set element, we can count the number of trailing zeroes in the bitset using `__builtin_ctz(v)`.

There are several neat tricks involving bitsets. Some worth mentioning are:

- computing the representation of $\{0, 1, \ldots, N-1\}$ with the expression `(1 << N) - 1`,
- removing the lowest-numbered element of the set with `x & (x - 1)`,
- retrieve the lowest-numbered element using `x & -x`, and
- iterating through all (non-empty) subsets of a bitset x using the loop

```
for (int sub = x; sub != 0; sub = (sub - 1) & x)
```

Exercise 6.13. Given a bitset, use bitwise operators to compute the next higher bitset with the same number of elements.

6.6 Hash Tables

In Section 3.1.6, we looked at a data structure called `map`, which stored a mapping from a set of *keys* to their corresponding *values*. The underlying implementation of this data

structure in STL is called a *self-balancing tree*, a quite difficult data structure. If one is willing to forego having the structure being sorted by keys, a ***hash table*** can be used instead. Hash tables are an easier implementation of the map, and can, depending on read and write patterns, be faster than the self-balancing tree implementation. It exists in STL as well, called `unordered_map`.

The operations the hash table supports are:

- set(x, y): set the value of key x to y.
  Complexity: expected $\Theta(1)$
- get(x): return the value of key x.
  Complexity: expected $\Theta(1)$
- erase(x): remove the key x.
  Complexity: expected $\Theta(1)$
- contains(x): returns whether the table contains the key x or not.
  Complexity: expected $\Theta(1)$

The main idea behind supporting these operations quickly is essentially the same as that of the dynamic array. Assume that the set of all possible keys, the *universe*, were the integers $0, 1, \ldots, N$ for some fixed N. If so, we could easily implement the above operations by storing the values in a dynamic array of this size. What do we do if this is not the case?

We apply the concept of ***hashing***. Imagine that your universe consists of all integers that fits in an `int`. A dynamic array can not be used to store a hash table of that size (2^{32}) – at least not in competitions. Instead, the goal of hashing is to shrink this huge universe into a small one that we could store in an array. For example, we could take the last K digits of the key for some small K (i.e the remainder when dividing by 10^K) and use it as the index into an array for the value of that key. There are only 10^K such keys, which for small K is a universe that can fit in an array.

Given such a mapping, we can store the key and value as a pair on the corresponding index in the array. This is illustrated for $K = 1$ in Figure 6.9.

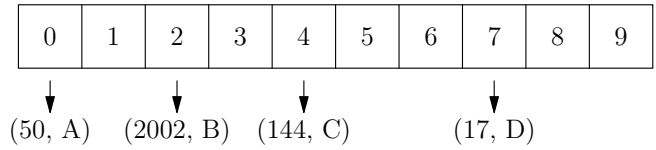


Figure 6.9: An example of where the hash table would store the values of a few keys when $K = 1$.

Transformations that take an arbitrarily large value (any integer) and maps it into a set of constant size, are called ***hash functions***[3].

[3]You might have heard about a version of this often used in cryptography, the *cryptographic hash function*, which aims to provide stronger guarantees than we care about.

Unfortunately, we are bitten by one of the fundamental limitations of mathematics – if a function maps a set to a smaller one, at least two values map to the same value. This is called a collision – two keys have the same index in the backing array. Resolving this situation is actually straightforward. Instead of using the table to store a single key-value pair, we can store a dynamic array of key-value pairs[4].

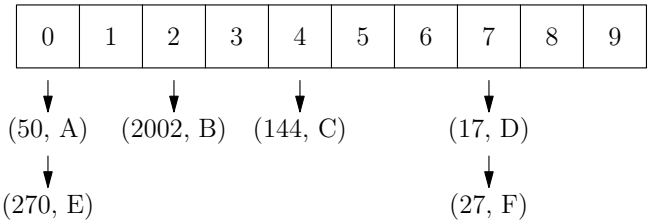


Figure 6.10: When several keys with the same hash value are stored, we save all pairs in a sub-array.

This complicates an implementation only slightly. To retrieve a value from the hash table, we look up the correct index of the array backing the table and check all key-value pairs stored in the array at that index. If we find a pair with the correct key, we return its associated value.

Setting the value of a key becomes a two-step process. First, we need to check the correct sub-array to see if it already contains a key-value pair with the same key. If it does, we update the value of the pair to the new value. If it does not, we insert the new key and value as a pair in that list.

We are not done yet. If the observant reader attempts to reason about the time complexity of the above solution, they would notice a problem. When all the keys inserted into the table have the same hash value – in the case of our simple function, the same last K digits – they map into the same position in the backing array. Searching through the sub-array to determine if it already contains the key becomes a linear time operation as our hash table degrades into a single array. Furthermore, if we pick a K that is much smaller than the number of keys, we must scan through very large arrays. For example, when $K = 1$ and we have 10^6 keys, every operation needs to scan through on average 10^5 values. This leaves us with two issues to resolve – how do we choose a reasonable hash function, and how big must our hash table be?

As for the second question, the goal is to keep the size a multiple of the number of keys. When the number of keys grows too large for the table, the size is increased and the hash of all keys recomputed with regards to the new table size. Typically, one would double the size of the hash table when it grows too dense, corresponding to how dynamic arrays were implemented. This keeps the amortized time complexity constant over resizing. As

[4]Traditionally, most introductions to the hash table use a data structure called a *linked list* to pairs that collide. We elect not to, since it's mostly a question about real-world performance, and expect you to use the implementation from STL if you use C++.

a bonus, if one starts with a table size that is a power of two, doubling it keeps the size a power of two. In the remainder of the section, we assume the size is equal to 2^N at the time of hashing.

The first question depends on the context. For the sake of programming problems, one can usually take the upper N bits of Ax as the hash, where A is a large constant odd constant, i.e:

```
(A * x) >> (64 - N)
```

If A is not odd, Ax in binary just has a few extra zeroes at the end, reducing the useful bits.

Exercise 6.14. What happens if one takes the lower N bits of the product Ax as the hash instead?

When we now move on to trying to compute the complexity of a hash table, we assume that a randomly chosen key has the same probability of being mapped to any of the possible hash values. By the resizing trick above, we can also assume that the table size is always within a constant of the number of keys.

Assume that K operations are performed on a table of some size M. The complexity of the i'th operation is exactly that of the length of the sub-array to which the key involved in the operation would map. The expected complexity of the operation is then equal to the expected length of the sub-array. Let a_j be 0 if the j'th operation inserted a key into this sub-array, and 1 otherwise (for $1 \leq j < i$). The expected length of the sub-array is then $\mathbb{E}[\sum_{j=1}^{i} a_j] = \sum_{j=1}^{i} \mathbb{E}[a_j]$ by the linearity of expectation. By the assumption that keys map randomly into hash values, $\mathbb{E}[a_j] \leq \frac{1}{M}$ so that the sum above is bounded by $\frac{j}{M} \leq \frac{K}{M}$. Since $\frac{K}{M} \leq c$ for some constant c by the dynamic resizing, the expected length is also bounded by a constant, meaning the complexity is as well.

Note that this analysis says nothing about the *worst-case* complexity of an operation (which can be linear of all keys map to the same hash value) or the expected length of the *longest* sub-array (which is $\frac{\log K}{\log \log K}$).

Universal Hashing

Certain competition forms include a stage where contestants may challenge the solutions of others for correctness, by providing a test case they believe the solution would fail. In this case, the hash function above is not good enough. Another contestant can easily generate values of x that all map to the same hash value, by generating a large number of values and evaluating your hash function on them, picking a large set of collisions from them. This also applies when using `unordered_map` from STL[5]. To resolve this, one picks a hash function from a family of functions at random at every invocation of your program, a concept called ***universal hashing***. In practice, the randomness tends come from reading the current time at a sufficiently granular level to be hard to predict.

[5] Using `map` is fine however, since it is not backed by a hash table.

The hash function we will look at is again Ax (as an unsigned 64-bit integer), but this time A is a random (odd) 64-bit integer. We claim that taking the top K bits of Ax makes a good hash function from 64-bit integers to K-bit integers, i.e.

$$h(x) = \left\lfloor \frac{Ax}{2^{64-K}} \right\rfloor.$$

To prove we are not making you use a weak hash for when you compete against the author, we provide a somewhat technical and uninteresting proof that uses some of the simpler number theoretical tools in Chapter 17.

Theorem 6.1
For any two fixed 64-bit integers x, y, their hashes $h(x)$ and $h(y)$ are equal with probability $\frac{2}{2^K}$ over all choices of the hash function parameter A).

Proof. This proof uses some basic number theoretic facts – if you are not familiar with modular inverses, you might need to work through Chapter 17.

Assume that $h(x) = h(y)$. This means that the top K bits of Ax and Ay are equal. Thus, the top K bits of $Ax - Ay = A(x - y)$ must either be all zeroes (if $Ax \geq Ay$) or all ones (if $Ax < Ay$, causing a carry bit from the top K bits).

Now, we introduce the following variables. Let z be the odd part of $x - y$ such that $x - y = z2^i$ for some i. Also, let B be the top 63 bits of A so that $A = 2B + 1$. Since A is a uniformly random odd 64-bit integer, B is uniformly random. We can now perform the rewrite $A(x - y) = (2B + 1)(x - y) = (2B + 1)z2^i = Bz2^{i+1} + z2^i$. Since B is uniformly random over 2^{63} and z is odd, $Bz \bmod 2^{63}$ is uniformly random over 2^{63} (this follows from the fact that z as an odd number is relatively prime with 2^{63}). Thus, the integer $A(x - y) = Bz2^{i+1} + z2^i$ is uniformly random in its top $63 - i$ bits and contains only zeroes in its lower i bits.

Note that $Ax = Ay + A(x - y)$. Since $A(x - y)$ has zeroes in the lower i bits and a 1 in the i'th bit, the i'th bit of Ay will change when adding $A(x - y)$ to it, so that it will differ from the i'th bit in Ax. By assumption the top K bits of Ax and Ay, are equal, which thus forces $i \leq 64 - K$.

This means that the top $63 - i \geq 63 - (64 - K) = K$ bits are uniformly random. A hash collision could only occur when they were all one or all zero, which happens with probability $\frac{2}{2^K}$. □

Exercise 6.15. Is it a problem that the hash has pairwise collisions with probability $\frac{2}{2^K}$ rather than $\frac{1}{2^K}$ with regard to hash table complexity?

Additional Exercises

Exercise 6.16. Assume that you want to implement shrinking of a dynamic array (or a hash table) where many elements were deleted so that the capacity is unnecessarily large.

This will be implemented by calling a particular function after any removal, to see if the array should be shrunk. What is the problem with the following implementation?

```
1: procedure ShrinkVector(V)
2:     while 2 · V.capacity > V.size do
3:         arr ← new T[V.capacity/2]
4:         copy the elements of V.backing to arr
5:         V.backing ← arr
6:         V.capacity ← V.capacity/2
```

Notes

For a more rigorous treatment of the basic data structures, we again refer to *Introduction to Algorithms* [11]. In particular, it goes through other techniques regarding hash tables more thoroughly, something we skipped since it is the hashing technique and general knowledge of the structure we deemed important here – an efficient implementation is something your language standard library will provide.

If you want to dive deeper into proper implementations of the algorithms in C++, *Data Structures and Algorithm Analysis in C++* [57] covers what we brought up in this chapter and a bit more.

Chapter 7

Recursion

This chapter introduces the first proper algorithmic principle in the book, that of ***recursion***. The first four chapters of the next part – brute force, greedy algorithms, dynamic programming and divide and conquer – are all based on this concept. Recursion is perhaps the first truly creatively tricky (rather than technically difficult) technique faced by the fresh programmer, so we have chosen to dedicate an entire chapter for a primer on the topic.

The remainder of this book, and computer science as a whole, strongly depends on a solid understanding of recursion. You are therefore urged to read it more carefully than the previous chapters. Even better; before you read it, make sure to have already read it![1]

7.1 Recursive Definitions

The first example of recursion that most people become acquainted with is the definition of a famous mathematical sequence, the *Fibonacci numbers*. The infinite sequence starts with the numbers 0, 1, 1, 2, 3, 5, 8, 13, ... Except for the first two, each number is computed by taking the sum of the two previous ones. A formal mathematical definition of the i'th Fibonacci number F_i can look like this:

$$F_i = \begin{cases} 0 & \text{if } i = 0 \\ 1 & \text{if } i = 1 \\ F_{i-1} + F_{i-2} & \text{for } i \geq 2. \end{cases} \tag{7.1}$$

By the definition, we would have e.g. $F_6 = F_5 + F_4 = 5 + 3 = 8$, which is true.

Exercise 7.1. Use the definition to compute the 10 first Fibonacci numbers.

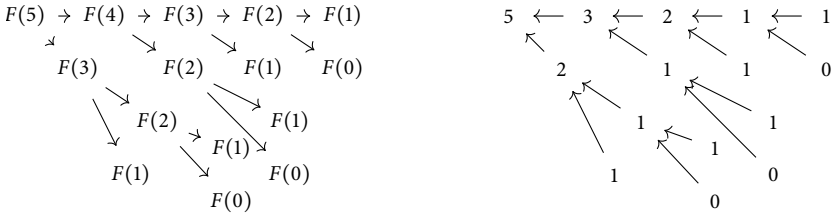
This is a so-called called ***recursive definition***, meaning that it refers back to itself – the definition of a Fibonacci number depends on the definition of (earlier) Fibonacci numbers. A program to directly implement the recursion looks very similar to the mathematical definition Eq. 7.1.

[1] This is a recursion joke that sadly isn't funny even when you know recursion.

Snippet 7.1: Computing Fibonacci Numbers

```
int F(int n) {
    if (n == 0) return 0;
    if (n == 1) return 1;
    return F(n - 1) + F(n - 2);
}
```

Note that this function, just like the recursive definition, computes its result $F(n)$ by calling itself to compute the (smaller) Fibonacci numbers $F(n-1)$ and $F(n-2)$. A knee-jerk reaction might be that such a function could never finish. After all, in order to compute a single Fibonacci number, the function calls itself, not just one, but two times! The solution is one of the key ideas of recursion, namely that there are base cases where the self-referential – recursive – computation eventually bottoms out, so that the definition does not refer back to itself forever and ever. In the case of Fibonacci, once you try computing F_0 or F_1, the definition gives us the values immediately without having to apply the recursive case. One can visualize the computation as in Figure 7.1.



(a) The recursive calls. **(b)** The return values.

Figure 7.1: A visualization of the computation of $F(5)$

Another application of the recursive principle would be computing a^n where n is non-negative integer. Since a^n is defined as the product of a n times, we can base a recursion around first computing a^{n-1} and then multiplying it with a:

$$a^n = \begin{cases} 1 & \text{if } n = 0 \\ a \cdot a^{n-1} & \text{for } n \geq 1. \end{cases} \tag{7.2}$$

The implementation is similarly straightforward.

Snippet 7.2: Recursive Exponentiation

```
int power(int a, int n) {
    if (n == 0) return 1;
    return a * power(a, n - 1);
}
```

Even though recursive definitions are the simplest examples of recursion, they often come up in practice.

Problems traditionally programmed using loops can also be solved by formulating them recursively[2]. Consider the problem of summing an array of integers. Normally, you'd use loops for such a task. However, there is nothing preventing you from using the following recursive definition. Let $A = (a_0, \ldots, a_{n-1})$ be an array of integers. If $S(k)$ is the sum of the first k elements of A, we have that

$$S(k) = \begin{cases} 0 & \text{if } k = 0 \\ a_{k-1} + S(k-1) & \text{otherwise.} \end{cases} \tag{7.3}$$

To compute the sum of the entire array, we call $S(n)$. Even though we now have to deal with a vector, the implementation is similar:

Snippet 7.3: Recursive Summing

```
// Invoked with sum(A, A.size())
int sum(const vector<int>& A, int k) {
    if (k == 0) return 0;
    return A[k - 1] + sum(A, k - 1);
}
```

Equation 7.3 is a recursive definition too: it reduces the problem of summing the entire array A to summing a smaller part of A. **This is the essence of recursion** – it was the common factor in all three examples. A recursive definition is meant to express the solution to a problem in terms of other instances of the same problem. The goal is that the new instances should be smaller than the first, in order to make progress on the problem.

Exercise 7.2. Write recursive functions to compute:

- the product of all integers in an array,
- the largest element in an array, and
- the greatest sum of two consecutive elements in an array.

[2] In fact, some programming languages do not have loops. Instead, you must formulate them recursively.

7.2 The Time Complexity of Recursive Functions

How fast is a recursive program? It is not as easy to compute as most of the algorithms we have seen so far. The work is distributed over several recursive calls, and we need to sum all of it up.

In all problems in this chapter, the time complexity of the recursive function except the recursive calls themselves is always the same. This is true of all of our examples so far – they only performed constant-time work plus some recursive calls. For functions where this is true the time complexity boils down to two factors: the number of function calls in total, and the time complexity of a single function call (excluding the recursive calls). Summing all the work is as simple as taking the product of these two things.

In the example of summing a vector of size m, there are a total of $m + 1 = \Theta(m)$ function calls. One call is made for each element, and one final call for the base case of the recursion. A single call performs only constant-time operations, so it has complexity $\Theta(1)$. The time complexity is therefore $\Theta(m) \cdot \Theta(1) = \Theta(m)$.

The number of function calls made in total can be considerably harder to compute, such as for the Fibonacci recursion.

Exercise 7.3. Write a program that uses the recursive function to compute Fibonacci numbers. Try computing all the Fibonacci numbers starting from F_{30} and upwards until the execution takes over 30 seconds. Take note of how long your program takes. What complexity does the function seem to have?

From the above exercise, it is clear that the running time is not a linear function. In fact, it turns out to be exponential. A simple lower bound is $2^{\frac{n}{2}}$ function calls, which we can prove by induction. Let $T(n)$ be the time taken to compute F_n. If $T(n) \geq 2^{\frac{n}{2}}$ for all n up to some $n' - 1$ and $n = 1$, then

$$\begin{aligned} T(n') &\geq T(n'-1) + T(n'-2) \\ &= 2^{\frac{n'-1}{2}} + 2^{\frac{n'-2}{2}} \\ &\geq 2^{\frac{n'-2}{2}} + 2^{\frac{n'-2}{2}} \\ &= 2^{1+\frac{n'-2}{2}} \\ &= 2^{\frac{n'}{2}} \end{aligned}$$

using the fact that $T(n) = T(n-1) + T(n-2) + \Theta(1)$, so the statement holds for $n = n'$ too. This lower bound is quite lax though – we can do better.

The above bound had the form of a polynomial: $(2^{\frac{1}{2}})^n$. If we substitute $x = 2^{\frac{1}{2}}$, the bound worked out because the inequality $x^{n-1} + x^{n-2} \geq x^n$ held. To find the best possible lower bound *of this form*, we should instead choose the greatest possible x such that $x^{n-1} + x^{n-2} \geq x^n$ is true. First divide the inequality by x^{n-2}, resulting in $x + 1 \geq x^2$, or $x^2 - x - 1 \leq 0$. We then solve a simple quadratic equation and find $\frac{1-\sqrt{5}}{2} \leq x \leq \frac{1+\sqrt{5}}{2}$,

attaining the slightly stronger lower bound $T(n) \geq \frac{1+\sqrt{5}}{2}^n \approx 1.618^n$. This is not quite rigorous: we ignore the term $\Theta(1)$ in our computations. When finding a lower bound, it's fine to throw away things from the recurrence since that only gives us a weaker bound. When proving the corresponding upper bound we need to take it into account though.

Exercise 7.4. Prove that $T(n) = O(1.62^n)$.

7.3 Choice

After reading the past examples, you may feel dissatisfied with recursion. Most of the problems we applied the recursive principle too is simplest solved by a basic for loop. While all recursion is based on reducing a problem instance to a smaller instance of the same problem, it perhaps makes more sense in other settings. This time, we look at problems involving choices of different kinds. Hopefully they convince you that the recursive principle can provide great power as a mode of thinking – even if using a recursive function *implementation* may still seem unnecessary as compared to a single for loop.

Stairs

Tasha the kitty loves playing with the stairs at home while her caretakers are at work. Her favorite game involves jumping up to the top of the stairs by repeatedly skipping either 1 or 2 stairs at a time. She doesn't like jumping on the exact same sequence of stairs during two different climbs.

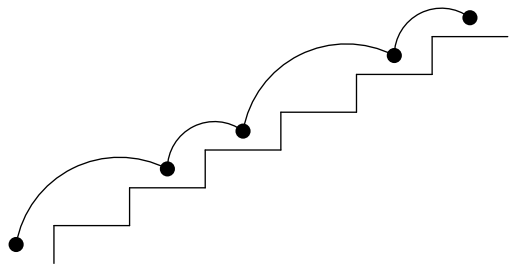


Figure 7.2: One way Tasha could climb a staircase of 6 stairs.

If the staircase has $1 \leq n \leq 20$ steps (including the top), in how many different ways can she climb the stairs?

Solution. With such a small n, computing this efficiently is not the main issue; computing it at all is. The trick lies in formulating Tasha's jumping up the stairs as a sequence of choices. After Tasha has jumped k steps, she has two choices – should her next jump be up a single stair to $k+1$, or two stairs to $k+2$? When dealt such a problem, always ask yourself: what was Tasha's last choice, just before she climbed up to the top of the stairs? Consider these two options in Figure 7.3.

If there are a total of n stairs and Tasha's last jump was a single step, then she came from step $n-1$. Similarly, if she took two steps, she came from step $n-2$. These two

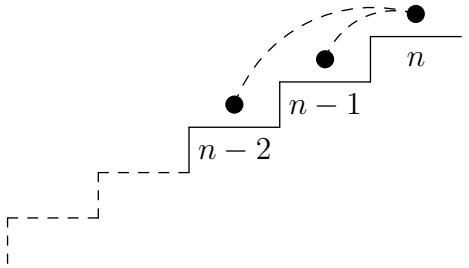


Figure 7.3: The two jumps leading to the top.

options are exhaustive – there is no other way she could have come to step n. They are also exclusive – we assumed that this was Tasha's last jump, so there is no overlap between these possibilities. This means that the number of ways Tasha can get to the n'th step must be equal to the number of ways she could get to the $(n-1)$'st step, plus the number of ways she could get to the $(n-2)$'nd step.

A recursive procedure based on this insight is then straightforward:

```
1: procedure Stairs(n)
2:     if n = 0 then
3:         return 1
4:     if n = 1 then
5:         return 1
6:     return Stairs(n − 1) + Stairs(n − 2)
```

Note the base cases we added, for the case of an empty staircase or a single stair. The time complexity of the solution is the same as that for Fibonacci, since the recursion is the same. □

We solve another problem from an early Swedish high school qualifier in the same way.

The Plank – theplank

By Håkan Strömberg. Swedish Olympiad in Informatics, School Qualifiers 2001.

You want to construct a long plank using smaller wooden pieces. There are three kinds of pieces of lengths 1, 2 and 3 meters respectively, each which you have an unlimited number of. You can glue together several of the smaller pieces to create a longer plank.

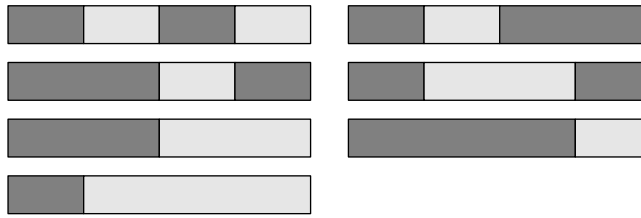


Figure 7.4: There are 7 ways to glue together a 4 meter plank.

If the plank should have length n ($1 \leq n \leq 24$) meters, in how many different ways can you glue pieces together to get a plank of the right length?

Solution. The idea here is the same as in the Stairs problem. To compute the number of all possible plank constructions, we need a recursive definition that reduces the problem into counting the number of ways one can build some smaller planks. For any given plank of length n, the rightmost piece of the plank has size either 1, 2 or 3. This means that the number of ways in which we can construct the plank is equal to the number of ways in which planks of sizes $n-1$, $n-2$ and $n-3$ can be glued together. While this isn't easier to compute directly, we can apply the same reduction recursively to these smaller planks, ending up with a very similar solution:

```
1: procedure PlankWays(n)
2:     if n < 0 then
3:         return 0
4:     if n = 0 then
5:         return 1
6:     return PlankWays(n − 1) + PlankWays(n − 2) + PlankWays(n − 3)
```

Again, we had to add a few base cases to give the recursion somewhere to stop. The two base cases we picked here may be slightly less intuitive. We say that there is a single way to construct a plank of length 0, and no ways to construct negative-length planks. □

Exercise 7.5. Prove that the time complexity of PlankWays algorithm has a lower bound of $\Omega(1.83^n)$ and an upper bound $O(1.84^n)$.

These two problems are much alike, and many other recursive problems follow this template:

- formulate the problem as a sequence of choices,
- look at what the last choice was, and
- find out if "backtracking" along that choice reduces the problem to smaller instances of the same problem.

Now that we have warmed up, we are going to look at a slightly harder recursive problem, where it is less obvious to figure out how to reduce the problem to a smaller one.

Dominoes

In how many ways can a $2 \times n$ ($1 \leq n \leq 20$) grid be tiled by n dominoes, i.e. bricks of size 1×2 or 2×1 such that no dominoes overlap?

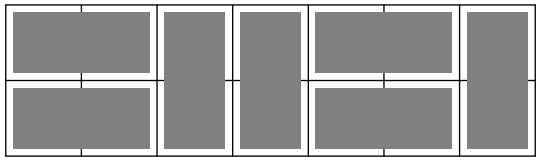


Figure 7.5: An example tiling of a 2×7 grid.

Solution. Looking at the example tiling in Figure 7.5 might help us. Let us denote the number of tilings of a $2 \times n$ grid with $S(n)$. In general, a recursion would somehow reduce the problem of computing $S(n)$ to computing smaller values of this function. By considering the rightmost domino of the example, a solution idea should form. If the rightmost tile is placed vertically, the remaining grid has size $2 \times (n-1)$, so there are $S(n-1)$ such tilings. If it is not placed vertically, the two rightmost squares must instead be occupied by two horizontal tiles. In this case, the remaining grid would have size $2 \times (n-2)$, meaning there would be $S(n-2)$ ways to complete the remainder of the tiling (see Figure 7.6).

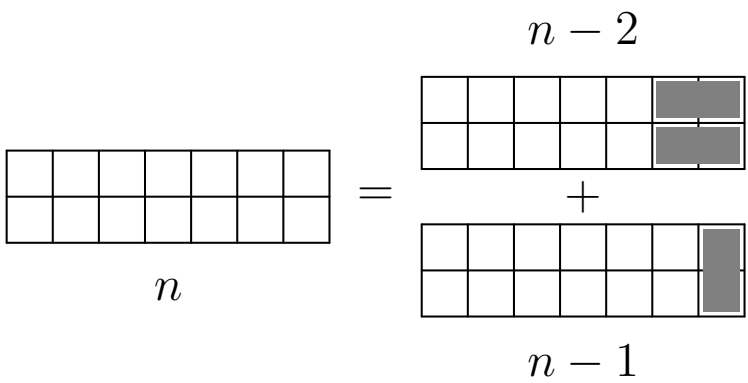


Figure 7.6: The two resulting subproblems after covering the rightmost column.

Since these are the only two options, the total number of tilings must be $S(n) = S(n-1) + S(n-2)$, and thus we get our recursive solution. Here too we got the same recursion as the one for Fibonacci, with the same time complexity. □

Exercise 7.6. 1. Write a recursion to compute the number of strings of length n consisting of only letters A and B, with no two A's next to each other.

2. Write a recursion to compute the number of subsets of $\{1, 2, \ldots, n\}$, where at least one of i, $i+1$ must be in the subset for all $1 \leq i < n$.

In the chapter on brute force we revisit this way of thinking as we use recursion to solve optimization problems rather than simply counting ways.

7.4 Multidimensional Recursion

So far, every recursive solution we produced were about a single sequence of numbers – the input was an integer n, and we computed the n'th value of the sequence through a recursive definition.

There are of course other recursions that come in all shapes and sizes. Recursing on more advanced problems can sometimes give us several recursive sequences that refer to smaller values of each other.

Varied Amusements – variedamusements

Marika and Lisa loves going to amusement parks. This time, they have their eyes set on a park with lots of exciting rides of three different types: tilt-a-whirls, roller coasters and drop towers. There are $1 \leq a \leq 10$ different tilt-a-whirls, $1 \leq b \leq 10$ roller coasters and $1 \leq c \leq 10$ drop towers. They want to ride $1 \leq n \leq 10$ different rides in sequence, but never two rides of the same type in a row. In how many ways can they choose such sequences of n rides?

Solution. On the surface, the problem is a prime candidate for the choice-strategy. There are n choices – what ride to go on each time. However, once we choose the last ride the girls took, we are faced with a problem. If we chose a roller coaster, the first $n-1$ rides may not end with a roller coaster. This is not a smaller instance of the same problem, where the last ride could be anyone we wanted. Instead, we get three different problems depending on the type of ride we choose as the last one: how many sequences of $n-1$ rides are there that does not end with *A*) a tilt-a-whirl? *B*) a roller coaster? or *C*) a drop tower?

What happens if we apply the same strategy to these three new problems? In the problem where we have to choose an $n-1$ ride sequence that does not end with a tilt-a-whirl, there are two options for the last ride. If it was a roller coaster, we have to choose the remaining $n-2$ rides such that they do not end with a roller coaster. If it was a drop tower, the remaining $n-2$ rides may not end with a drop tower. Either way, both cases reduce to a smaller problem of the other two types.

By introducing the three new problems $A(n)$, $B(n)$ and $C(n)$, defined as the number of ride sequences of length n not ending in a tilt-a-whirl, roller coaster or drop tower

respectively, we can produce recursive definitions that refer only to these recursions:

$$A(n) = b \cdot B(n-1) + c \cdot C(n-1)$$
$$B(n) = a \cdot A(n-1) + c \cdot C(n-1)$$
$$C(n) = a \cdot A(n-1) + b \cdot B(n-1)$$

with the base cases of $A(0) = B(0) = C(0) = 1$. The answer then becomes $a \cdot A(n-1) + b \cdot B(n-1) + c \cdot C(n-1)$.

When implementing the solution in C++, don't forget results from Exercises 2.29-2.30 to resolve the circular dependencies between functions calling each other. □

Exercise 7.7. Compute, with proof, the time complexity of the solution to Varied Amusements.

Problem 7.8.
Varied Amusements variedamusements *(subtask 1)*

7.5 Recursion vs. Iteration

After looking through the problems we solved in this chapter, you might wonder if recursion really is needed. When computing the Fibonacci numbers by hand, we tend to not use a method nearly as complicated as the recursive function. Instead, we write them down one by one after each other, taking the sum of the last two ones to compute the next. This approach could be implemented iteratively in code:

Snippet 7.4: Iterative Fibonacci

```
int F(int n) {
    int secondLast = 0; // stores F_(i-2)
    int last = 1; // stores F_(i-1)
    for (int i = 2; i <= n; i++) {
        // Compute F_i = F_(i-2) + F_(i-1)
        int current = last + secondLast;
        // Since i will be be increased by 1,  the old F_(i-1) becomes
        // F_(i-2), and the old F_i becomes F_(i-1)
        secondLast = last;
        last = current;
    }
    return last;
}
```

Algorithmically, recursion – in the sense of functions calling themselves – is not needed. As a programming construct it doesn't bring any additional computational powers. Recursive functions can even be simulated with a single loop and a stack, by storing the

current chain of recursive calls in the stack and processing them one at a time in the loop. This is actually what your computer does behind the scenes[3].

The reason behind the strong focus in recursion is another, namely that it is an incredibly powerful mode of thinking. To us, it is unclear how one would find a natural solution to the Dominoes problem without going in with a recursive mindset and looking for that reduction to a smaller instance. That being said, once a recursive formulation has been deduced, an iterative implementation can many times be simpler or faster. For example, try to compute F_{46} using both the recursive and iterative approach. You'll see that one of the two versions finishes, and one do not.

Solving a recursion iteratively when it has several sequences referring to each other is possible too. It is slightly more difficult since you must be careful in what order you compute values. All values that the one being computed is dependent on must be filled in beforehand.

Problem 7.9.

Varied Amusements	variedamusements	*(all subtasks)*

Additional Exercises

Exercise 7.10. There are n lines drawn in the plane, no two of which are parallel, and no three lines intersecting at the same point. What is the number of connected regions they divide the plane into?

Problem 7.11.

Odd A's, Even B's	oddaevenb
Tritiling	tritiling

Notes

Recursion as a problem solving technique is a common one both in mathematics and algorithmics. Mathematics has greater focus on finding closed forms for the recursion, while we are happy with any kind of efficient computation. There is a rich combinatorial theory behind finding such closed forms. As previously mentioned, *Concrete Mathematics* [29] is one of the "introductory" mathematics books that really excel on teaching these techniques.

Analytic Combinatorics [16] describes many interesting analytic methods to solve the kind of recursive counting problem we discussed. While it's sometimes hard to find exact solutions to analytically, those methods provide a great framework to determine the order of growth of this kind of sequences.

[3] At a low level, modern processors are basically a single execution loop with a stack for function-related memory.

Chapter 8

Graph Theory

We end the foundational part with an introduction to ***graph theory***, the study of mathematical objects known as graphs. As a mathematical area, it dates back to the early 1700s, when Euler first studied the famous *Seven Bridges of Königsberg* problem. It is one of the most well-studied areas in algorithmic problem solving. You can find a graph theory problem in almost every programming contest.

8.1 Graphs

A graph is an abstract way of representing various types of relations, such as roads between cities, friendships between people, network links between computers and so on. Put simply, they are a set of objects where certain pairs of objects are connected. Formally, graphs are defined in the following way.

> **Definition 8.1** A ***simple graph*** $G = (V, E)$ consists of a pair V of ***vertices*** and a set E of ***edges***. An edge consists of a set of vertices $\{u, v\}$ called the ***endpoints*** of the edge.

A graph lends itself naturally to a graphical representation, where vertices are represented by points in the plane with lines drawn between the two endpoints of each edge. For example, the graph given by $V = \{1, 2, 3, 4, 5\}$ and $E = \{\{1, 2\}, \{2, 3\}, \{3, 4\}, \{2, 4\}\}$ can be drawn as in Figure 8.1.

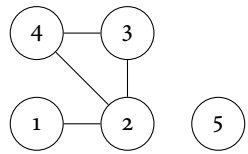


Figure 8.1: An example graph with 5 vertices and 4 edges.

Exercise 8.1. Draw the graphical representation of the graph with vertices $\{a, b, c, d\}$ and edges $\{\{a, b\}, \{b, c\}, \{c, d\}, \{a, d\}, \{b, d\}\}$.

Exercise 8.2. The graph on n vertices containing *all* possible edges is called the ***complete graph***, or K_n. How many edges does K_n have?

Trip Planning – tripplanning

Lars is planning to do a backpacking tour by train throughout $N \leq 10^6$ cities in Europe. He has a list of train lines numbered from 1 to $M \leq 10^6$ that each goes back and forth between some pair of cities, no two between the same pair. He wants to visit the cities in the order $1, 2, \ldots, N$, finally returning back to his home in city 1.

Since Lars has limited vacation days, he only has time to take exactly N direct trains during his trip. Can you determine if this is possible, and tell Lars the numbers of the train lines he should take?

Solution. There is only a single sequence of train trips that fulfills Lars' requirement, namely $(1,2), (2,3), \ldots, (N-1,N), (N,1)$. The problem thus asks if all of these train lines exist. If they do, we should print their numbers. This is a typical problem that can be modeled as a graph. In those terms, we have a graph on N vertices with its M edges given in a list. We are asked if the graph contains a certain list of edges.

A possible solution would be to keep a vector of the indices of these particular edges while we read the list of edges. If we find the edge $\{k, k+1\}$ for a given k, we store the index of the edge in the k'th position in the vector. Only if we managed to find every edge should we reply with their indexes. Otherwise, we output that there is no trip. □

We often perform operations on the set of vertices adjacent to a specific vertex, especially in later sections when searching through graphs. These sets have a special name.

Definition 8.2 For all edges $\{u, v\}$ in a graph, the vertices u and v are ***neighbors*** or ***adjacent*** to each other. The set of neighbors of a vertex v is called its ***neighborhood***. The size of the neighborhood of a vertex v is called the ***degree*** of v. It is denoted $\deg(v)$.

The degrees of the vertices in a graph fulfill many useful properties. A simple theorem that shows how one can reason about graphs concerns the sum of all degrees in a graph.

Theorem 8.1

The sum of degrees of a graph $G = (V, E)$ is even. Specifically,

$$\sum_{v \in V} \deg(v) = 2|E|.$$

Proof. Both sides of the equation count the same thing. The right-hand side counts each edge in the graph twice. For a given edge $\{u, v\}$, how many times is it counted in the left-hand side? It contributes 1 each to the degrees of u and v, so it must be counted twice as well. Since both sides count the same thing, they are equal. □

Exercise 8.3. Use Theorem 8.1 to prove that there is an even number of vertices of odd degree in a graph.

Example 8.1 In a graph G, the degrees of its vertices are 3, 5, 4, 4, 4, 6, 6 respectively. Prove that there is a sequence of adjacent vertices starting and ending with the vertices of degree 3 and 5.

Solution. Let v be the vertex of degree 3, and S the set of vertices u for which there is a sequence of adjacent vertices starting at v and ending at u. Construct a new graph H with S as vertices and as edges only those between two vertices in S. For every u in S, all neighbors of u must be in S too, so the degree of a vertex in H is the same as it was in G.

The sum of the degrees of all vertices in S is even by Theorem 8.1. Since S contains a vertex of degree 3, it must contain another vertex of odd degree (or the sum would be odd). The only vertex with an odd degree is the one of degree 5, so it must be in S too. By the definition of S, there is a sequence of neighboring vertices between the two vertices. □

Exercise 8.4. Prove that in a simple graph of at least 2 vertices, there must exist 2 vertices of the same degree.

Problem 8.5.

Railroad	railroad2
Popularity Contest	popularitycontest
Hermits	hermits

While the simple graph is able to represent many kinds of relations, we sometimes need extra information to capture all the aspects we are interested in. Consider a graph representing roads between cities, where the vertices represent the cities and the edges correspond to the roads between them. In this case, we might be interested in also capturing the lengths of the roads between all the cities: a situation depicted in Figure 8.2.

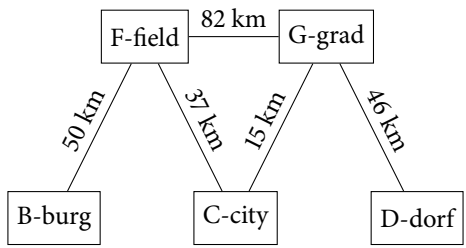


Figure 8.2: A road network on 5 cities.

We can modify our definition of an edge to include such a number, giving us a new type of graph.

Definition 8.3 A ***weighted graph*** is a graph together with a ***weight function*** $w(e)$ that associates each edge e with a real-valued ***weight***.

Weighted graphs often appear when there is a natural measure of an edge, like the distance between two cities, the latency between two computers, or the cost of a flight between two airports.

Problem 8.6.
Triangle Drama triangledrama

Finally, not all relations we model are symmetric in the way indicated by normal graphs. In many situations, we prefer if an edge could have a certain direction, going *from* a vertex *to* another vertex. For example, when modeling all the car roads in a city, some roads may be one-way, a nuance the simple graph would miss. We fix this by making edges ordered pairs rather than sets:

Definition 8.4 A ***directed graph*** is a graph (V, E) where E consists of ***directed edges***, i.e. ordered pairs (u, v) of vertices. The edge (u, v) is called an ***out-edge*** of u and ***in-edge*** of v. Similarly, the ***outdegree*** and ***indegree*** of a vertex is the number of out-edges and in-edges of the vertex.

When representing directed graphs graphically, edges (u, v) are arrows with the arrowhead pointing from u to v (Figure 8.3).

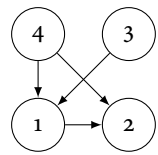


Figure 8.3: The graph given by $V = \{1, 2, 3, 4\}$ and $E = \{(1, 2), (3, 1), (4, 2), (4, 1)\}$.

Problem 8.7.
Eulerian Graphs eulerian

8.2 Representing Graphs

When dealing with graphs in algorithms, there are three common data structures used to represent them: *adjacency matrices*, *adjacency lists* and *adjacency maps*. Graphs are often given implicitly, where a problem specifies rules for how to find the edges that a given vertex is the endpoint of. This latter representation is for example common when dealing

with searches in graphs representing all the positions in a game (Section 16.3), where the rules of the game dictate how the graph looks.

Adjacency Matrices

An adjacency matrix represents a graph on n vertices as a 2D $n \times n$ matrix in the following way:

> **Definition 8.5 — Adjacency Matrices**
> For a directed, weighted graph with vertices $V = \{v_1, ..., v_n\}$ and weight function $w(e)$, the graph's ***adjacency matrix*** is defined as the $|V| \times |V|$ matrix with entries $a_{i,j} = w(v_i v_j)$.
>
> For undirected graphs, we set $a_{i,j} = a_{j,i} = w(\{v_i, v_j\})$, and for unweighted graphs $a_{i,j} = 1$.

This representation uses $\Theta(|V|^2)$ memory, and takes $\Theta(1)$ time adding, modifying and removing edges. To iterate through the neighbors of a vertex, you need $\Theta(|V|)$ time, even if the vertex has lower degree. Adjacency matrices are best to use when $|V|^2 \approx |E|$, i.e. when the graph is ***dense***.

The adjacency matrix for the directed, unweighted graph in Figure 8.3 is:

	1	2	3	4
1	0	1	0	0
2	0	0	0	0
3	1	0	0	0
4	1	1	0	0

When a graph allows ***parallel*** edges, i.e. multiple edges between the same pair of vertices, or ***self-loops***, edges going from a vertex to itself, the adjacency matrix instead stores the number of edges between each pair of vertices. If an adjacency matrix is the right structure for a weighted directed graph with parallel edges, it is often enough to store the weight of the lightest edge among those that are parallel.

Adjacency Lists

Another way to represent graphs is by storing lists of neighbors for every vertex, so-called ***adjacency lists***. This approach uses only $\Theta(|E| + |V|)$ memory. When the graph has significantly fewer edges than $|V|^2$, i.e. it is ***sparse***, this can be considerably less than what adjacency matrices use. If you use a vector to represent each list of neighbors, you also get $\Theta(1)$ addition and removal (if you know the index of the edge and ignore their order) of edges, but it takes $\Theta(|V|)$ time to determine if an edge exists in the worst case. On the upside, iterating through the neighbors of a vertex takes time proportional to the number of neighbors instead of the number of vertices in the graph. Iterating through all the

neighbors of all vertices takes time $\Theta(|E| + |V|)$ compared to $\Theta(|V|^2)$ for the adjacency matrix. As with the memory usage, this is clearly better for large, sparse graphs.

When representing weighted graphs, the list stores the edges as pairs of (*neighbor*, *weight*). For undirected graphs, both endpoints of an edge contains the other in their adjacency lists.

This representation is common in many graph search algorithms to be studied in later sections and Chapter 14.

Adjacency Maps

An ***adjacency map*** combines the upsides of both structures ($\Theta(1)$ time to check if an edge exists) and the lists (low memory usage and fast neighborhood iteration) in a single data structure. Instead of storing a fixed-size vector per vertex as for the matrices, or a vector containing only the edges, we can use a hash table per vertex to store its neighbors. The adjacent vertices are stored as keys in the table, with the weights of the corresponding edges as their values.

This has the same time and memory complexities as the adjacency lists, but it also allows for checking if an edge is present in $\Theta(1)$ time. The downsides are that hash tables have a higher constant factor than adjacency lists, and that you lose the ordering you have of your neighbors (if this is important). The adjacency map also inherits another sometimes important property from the matrix: you can remove arbitrary edges in $\Theta(1)$ time without knowing anything about its location in an adjacency list.

When a graph that is dynamically modified has a large number of vertices the adjacency map is the representation of choice.

Exercise 8.8. Given a graph, list the appropriate representation(s) under the following constraints.

1. $|V| = 1000$ and $|E| = 499500$
2. $|V| = 10000$ and $|E| = 20000$
3. $|V| = 1000$ and $|E| = 10000$

Problem 8.9.

Weak Vertices	weakvertices
Eulerian Graphs 2	eulerian2

8.3 Breadth-First Search

The most common algorithms on graphs involve *searching* them, trying to find the kind of sequences of adjacent vertices that Example 8.1 were about. In graph theory parlance, they are called *paths*.

Definition 8.6 — Paths and Distances
In a graph, let $(p_0, p_1, \ldots, p_l)$ be a sequence of **distinct** vertices where p_i and p_{i+1} are adjacent. We call the sequence of edges $p_0p_1, p_1p_2, \ldots, p_{l-1}p_l$ a ***path*** of ***length*** l.

For two vertices v and u, we call a path of minimum length between them a ***shortest path***. The ***distance*** between them, written as $d(v, u)$, is defined to be the length of a shortest path between them.

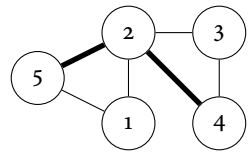


Figure 8.4

In Figure 8.4, the sequence $5 \to 1 \to 2 \to 4$ is a path of length 3. It is not a shortest path between the two vertices 4 and 5, which is the bolded path $5 \to 2 \to 4$. The distance between 4 and 5 is thus 2.

Exercise 8.10. Find the distance and a shortest path between every pair of vertices in Figure 8.4.

For unweighted graphs, we can find the shortest paths from a given vertex to *every* other vertex in the graph very efficiently.

Single-Source Shortest Path, Unweighted Edges

Given an unweighted graph and a source vertex s, compute the shortest distances $d(s, v)$ for all vertices in the graph.

Solution. For simplicity, we first consider the problem on a grid graph, where the unit squares constitute vertices, and squares that share an edge are connected. Some squares are blocked and don't have a vertex in the graph. An example can be seen in Figure 8.5.

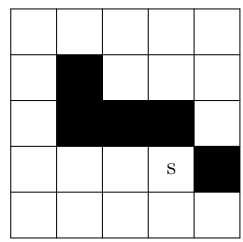


Figure 8.5: An example grid graph, with source marked s.

Initially, we know what vertices have distance 0. This is only the source vertex s itself. This seems like a reasonable starting point, since the problem is about shortest paths from

s. The next natural question is what vertices have distance 1? These are exactly those with a path consisting of a single edge from s – it's neighbors (marked in Figure 8.6).

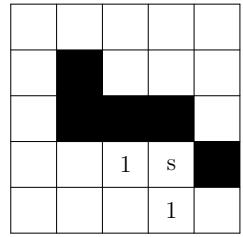


Figure 8.6: The squares with distance 1 from the source.

If a vertex v has distance 2, it must be a neighbor of a vertex u with distance 1 (except for the starting vertex). This is also a sufficient condition, since we can construct a path of length 2 by extending the path of any neighbor of distance 1 with the edge (u, v).

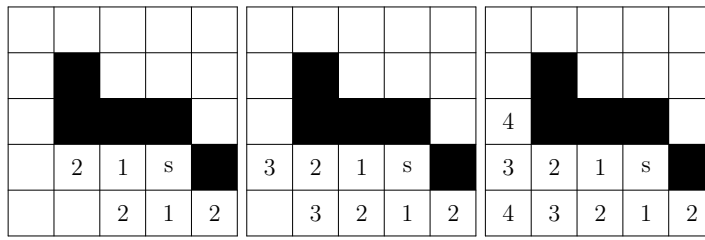
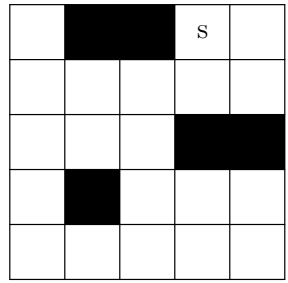


Figure 8.7: The squares with distances 2, 3 and 4.

This last line of reasoning generalizes to any particular distance, i.e., that all vertices with distance k must have a neighbor with distance $k-1$ but no neighbor with a smaller distance. The principle can be used to construct the following algorithm. Initially, we set the distance of s to 0. Then, for every $dist = 1, 2, \ldots$, we mark all vertices that have a neighbor with distance $dist - 1$ as having distance $dist$ if they did not already have a shorter distance. □

Exercise 8.11. Use the BFS algorithm to compute the distance to every square in the following grid:



This algorithm is called the ***breadth-first search***. As described so far it is slightly underspecified, and could easily be implemented in quadratic time. A simple linear-time implementation iteratively constructs lists of the vertices at each distance. Given this list for a distance d, the list for distance $d+1$ could then be constructed just as described.

```
1:  procedure BreadthFirstSearch(vertices V, vertex s)
2:      distances ← int vector of size |V| filled with ∞
3:      distances[s] ← 0
4:      dist ← 0
5:      atDist ← vector of vertices
6:      atDist.add(s)
7:      while atDist is not empty do
8:          atDistPlusOne ← vector of vertices
9:          for every from in atDist do
10:             for every neighbor v of from do
11:                 if distances[v] = ∞ then
12:                     atDistPlusOne.add(v)
13:                     distances[v] ← dist + 1
14:         dist ← dist + 1
15:         atDist ← atDistPlusOne
16:     return distances
```

Each vertex is added to *nextVertices* at most once, since it is only added if $distances[v] = \infty$ which is immediately set to something else. We then iterate through all neighbors of these vertices. The number of all neighbors is $2E$ in total, so the algorithm uses $\Theta(V+E)$ time.

Usually, the outer loop is often coded in another way. Instead of maintaining two separate vectors, we can merge them into a single queue:

```
1:  while atDist is not empty do
2:      from ← atDist.pop()
3:      for every neighbor v of from do
4:          if distances[v] = ∞ then
5:              atDist.add(v)
```

6: $distances[v] = distances[from] + 1$

Exercise 8.12. Prove that the shorter version of the BFS loop is equivalent to the longer one.

Problem 8.13.

Erdős Numbers	erdosnumbers
Horror List	horror
Erratic Ants	erraticants
Button Bashing	buttonbashing

In many problems the task is to find a shortest path in a graph where vertices and edges are given *implicitly*, with specified rules on what edges exist. To use the BFS, we must first figure out how the graph is generated.

8-puzzle

In the *8-puzzle*, 8 tiles are arranged in a 3×3 grid, with one square left empty. A move in the puzzle consists of sliding a tile into the empty square. The goal of the puzzle is to perform some moves to reach the target configuration. The target configuration has the empty square in the bottom right corner, with the numbers in order 1, 2, 3, 4, 5, 6, 7, 8 on the three lines.

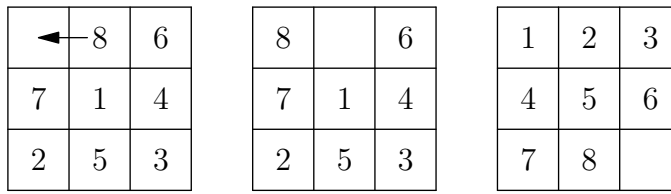


Figure 8.8: A puzzle with a valid move. The target configuration is to the right.

Given a puzzle, solve it in as few moves as possible. The moves should be given as a sequence of the digit on the tile that is slided in each move.

Solution. Finding the minimum number of moves is a typical BFS problem, characterized by a starting state (the initial puzzle), some transitions (the moves we can make), and the task of finding a short sequence of transitions to some goal state. We start solving this problem, and figure out exactly how the sequence is constructed afterwards.

The problem must first be modeled using a graph. The vertices represent the possible arrangements of the tiles in the grid, and an edge connects two states if they differ by a single move. A sequence of moves from the starting arrangement to the target configuration represents a path in this graph. The minimum number of moves required is the same as the distance between those vertices in the graph, so we can use a BFS.

In an implicit-graph BFS problem, most of the code deals with the representation of a state as a vertex, and generating the edges that a certain vertex is adjacent to. We generally

avoid computing the entire graph explicitly. Instead the edges from a vertex are generated when the vertex is visited in the breadth-first search. In the 8-puzzle, we can represent each state as a 3×3 2D-vector. The difficult part is generating all the states one move away from a certain state.

Since this is mostly a language specific programming technique, we give an example in C++ on move generation.

Snippet 8.1: Generating 8-puzzle Moves

```
typedef vector<vector<int>> Puzzle;

vector<Puzzle> edges(const Puzzle& v) {
  int emptyRow, emptyCol;
  for (int row = 0; row < 3; row++)
    for (int col = 0; col < 3; col++)
      if (v[row][col] == 0) {
        emptyRow = row;
        emptyCol = col;
      }
  vector<Puzzle> possibleMoves;
  auto makeMove = [&](int rowMove, int colMove) {
    int newRow = emptyRow + rowMove;
    int newCol = emptyCol + colMove;
    if (newRow >= 0 && newCol >= 0 && newRow < 3 && newCol < 3) {
      Puzzle newPuzzle = v;
      swap(newPuzzle[emptyRow][emptyCol], newPuzzle[newRow][newCol]);
      possibleMoves.push_back(newPuzzle);
    }
  };
  makeMove(-1, 0);
  makeMove(1, 0);
  makeMove(0, -1);
  makeMove(0, 1);
  return possibleMoves;
}
```

Note how useful the lambda language feature (Section 2.11) is. If `makeMove` is a separate function, we have to pass many extra variables to it.

With the edge generation finished, the rest is a normal BFS, slightly modified to account for the fact that the vertices are no longer numbered $0, \ldots, V-1$. Instead of vectors, we can use e.g. maps to store the list of distances.

Generating the actual sequence of moves remains. This is done through a process called *backtracking*. To support it, the BFS must store additional information. Whenever we add a vertex u to the queue through the edge $v \to u$, we keep track of the fact that u's shortest path was an extension of v's. With this change, the full BFS code could look something like this:

Snippet 8.2: 8-puzzle BFS

```
map<Puzzle, Puzzle> puzzle(Puzzle S, Puzzle target) {
  map<Puzzle, int> distances;
  map<Puzzle, Puzzle> from;
  distances[S] = 0;
  queue<Puzzle> q;
  q.push(S);
  while (!q.empty()) {
    Puzzle cur = q.front();
    q.pop();
    if (cur == target) break;
    int dist = distances[cur];
    for (auto& move : edges(cur)) {
      if (distances.find(move) != distances.end()) continue;
      distances[move] = dist + 1;
      q.push(move);
      from[move] = cur;
    }
  }
  return from;
}
```

Infinite distances to a state in `distances` are represented by the absence of the state (it would be a waste to store it for all possible states). When states are integers in a vector, we can instead use value guaranteed to exceed the maximum distance.

Given the `from` map, the move sequence can be constructed in reverse. Starting at the target move, we follow the sequence of positions on the shortest path back to the initial one by repeatedly looking up the previous position using the backtracking map. We then compare consecutive states to see what tile was moved. This approach is outlined in the next snippet.

Snippet 8.3: 8-puzzle Backtracking

```
vector<Puzzle> sequence;
sequence.push_back(target);
Puzzle at = target;
while (at != S) {
    at = from[at];
    sequence.push_back(at);
}
reverse(sequence.begin(), sequence.end());
for (int i = 0; i < sequence.size() - 1; i++) {
    Puzzle a = sequence[i];
    Puzzle b = sequence[i + 1];
    // Look at the difference between a and b...
}
```

To simplify the backtracking, the `from` mapping often stores both the previous state and the actual move to avoid having to reconstruct a move from two states. □

Competitive Tip

Sometimes, using maps with keys like 3×3 vectors in the BFS is too slow. When this happens, you can try to simplify the key type. In the 8-puzzle problem, there is a mapping from puzzles to 9-digit integers: write out the digits of the puzzle one row at a time. Using integers as map keys gives a (sometimes significantly) better constant factor – for the 8-puzzle, about 2.4 times better when the author tested it!

In some solutions we can even map the state to integers so small that we can use a vector instead. In this case, there are only $9 \cdot 8 \cdot \cdots \cdot 2 \cdot 1 = 362880$ possible states (why?), so if we manage to find a mapping like that we can avoid maps entirely. This change gave the author another factor 2 speedup for the 8-puzzle.

Problem 8.14.

Peg Solitaire	solitaire
Pebble Solitaire	pebblesolitaire2

While this kind of search problem directly uses the shortest paths found by a standard BFS as the answer, some problems require modifications of a BFS, or use the distances generated only as an intermediary result.

Shortest Cycle

Find the length of the shortest cycle in a graph on $V \leq 200$ and $E \leq \frac{V(V-1)}{2}$ edges. A ***cycle*** is a path with an extra edge between the first and last vertices.

Solution. We use a typical simplifying reduction for graphs. The shortest cycle in the entire graph is a very global property, and they are usually hard to compute directly. It is often easier to restrict what you are trying to compute to something local, like the shortest cycle that a given vertex is part of. As long as we can find that cycle in linear time, we can afford doing this for every vertex in the graph, while guaranteeing that we find the shortest cycle overall.

To find the shortest cycle containing some given vertex, a modified BFS does the trick. Consider how the vertices on a cycle are searched by the BFS.

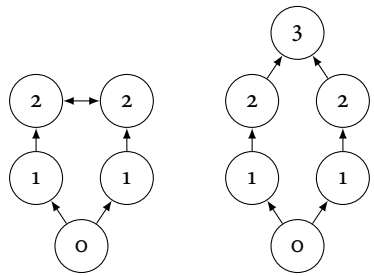
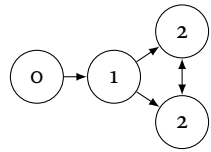


Figure 8.9: The BFS search order along a cycle.

On an even-length cycle, the vertex furthest away from the source node is the first vertex seen twice by vertices closer to the source. On an odd-length cycle, one of the two vertices furthest away is instead the vertex first seen by a vertex with the same distance to the source. Checking for when one of these events occur inside the BFS is simple, but one can also check for these two conditions after performing the BFS. □

Exercise 8.15. In *Shortest Cycle*, the proposed algorithm can detect cycles that the source vertex is not actually part of, as in the following graph.



Is this a problem for the algorithm?

Problem 8.16.

Beehives	beehives

Exercise 8.17. A variant of the shortest path problem is when there is no longer a single source, but potentially multiple. Rather than finding the distance from each vertex to the source, we want to find the distance to *any* source for each vertex in the graph. This problem can be reduced to the single source version and thus solved with a single BFS. Find this reduction.

Problem 8.18.

Wet Tiles	wettiles
Fire	fire2

Exercise 8.19. The original BFS algorithm hides a way to find shortest paths in weighted graphs where all edge weights are either 1 or 0. The idea is that if we encounter an edge (v, u) with weight 0 at some point, we can add it to the vector *atDist* rather than *atDistPlusOne*, since u has the same distance as v (and thus should be processed at the same time). How can this idea be integrated into the shortened BFS algorithm?

Problem 8.20.

Bridges	bryr
Ocean Currents	oceancurrents

8.4 Depth-First Search

In some graph searches we don't care about finding the *shortest* path to all other vertices, but are rather interested in if there is a path at all. This is the case when we want to determine ***connectivity*** in a graph.

Definition 8.7 If there exists a path with vertices u and v as endpoints, we call them ***connected***. If all vertices in a graph are connected we call the entire graph connected.

The maximal connected vertex subsets of a graph are called the ***connected components*** of the graph.

Each vertex belongs to a single component, so they form a partition of the vertices. This is a fact we use in the next problem.

Sailing Friends – sailingfriends

A common saying goes that if you like sailing, it is much better to have a friend with a boat rather than being the friend with a boat.[a]

Being mathematically inclined, you have realized that this property applies recursively – it is much better to be the *friend of a friend* with a boat rather than having a boat, and so on.

In a small coastal city, there are $N \leq 100\,000$ people, some of which already have boats. Among them, $E \leq 200\,000$ pairs of people are friends with each other. You start to wonder, what is the smallest amount of people who would have to buy a boat so that everyone in the city either owns a boat or could borrow one from a friend (possibly indirectly through a friend of a friend of a …)?

[a] Boats are very expensive and require a lot of maintenance, we have heard.

Solution. With the new terminology on connectivity, we can formulate this with graphs. All the people and their friendships form an undirected graph. Someone can borrow a boat if anyone in their connected component has a boat. Thus, the number of extra people who need to buy a boat equals the number of connected components in which nobody has a boat.

It is a bit overkill to use BFS for the task of searching through components for vertices with boats since we are not interested in finding any shortest paths. A similar alternative is to start a search from some source vertex and perform the search recursively for each neighbor, rather than searching every neighbor before *their* neighbors are searched. The algorithm is similar to the BFS, but they search the graph in a very different order. The breadth-first search grows the set of visited vertices using a wide frontier around the source vertex, while the depth-first search instead tries to first plunge deeper into the graph as long as it finds a new vertex (hence the names). An example of how a DFS can search a graph is given in Figure 8.10, where the vertices are marked with the order in which they are visited.

This results in very short code to search the component containing a given vertex and determine if it contains a boat:

```
1: procedure FindBoat(vertex at)
2:     seen[at] = true
3:     foundBoat ← hasBoat[at]
4:     for every neighbor next of at do
```

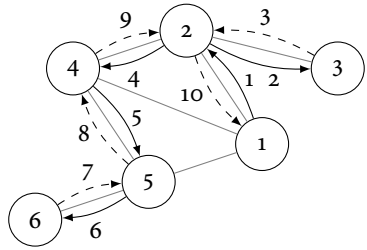


Figure 8.10: Normal arrows indicate when the DFS enters a node, while dashed arrows show when the DFS backtracks.

```
5:        if not seen[next] and FindBoat(next) then
6:            hasBoat ← true
7:    return hasBoat
```

To keep track of what vertices have been visited by the search, we use a global array *seen*. The search is done until all vertices must have been visited:

```
1: extraBoats ← 0
2: for ever person p do
3:     if not seen[p] then
4:         if not FindBoat(p) then
5:             extraBoats ← extraBoats + 1
```

□

Problem 8.21.

Reachable Roads	reachableroads
A Mazing!	amazing
Dropping Directions	droppingdirections

It might seem as if the DFS is just a weaker version of the BFS that is somewhat easier to code. This is true until Chapter 14, where we study advanced applications of the DFS where the BFS can not help us. Why not wait until then to introduce it? Due to the simplicity of coding the DFS compared to a BFS, it is usually the algorithm of choice in problems where we just want to determine connectivity, so adding it to your toolbox early will save you a lot of time until then. We show another application where some opt to use a DFS over a BFS, namely two-coloring graphs.

Breaking Bad – breakingbad

By Bjarki Ágúst Guðmundsson. RU ÁFLV 2014. CC BY 3.0. Shortened.

Walter was once a promising chemist. Now he teaches high school students chemistry, and was recently diagnosed with lung cancer. In both desperation and excitement he decides to use his chemistry skills to produce illegal drugs and make quick money for his family. He forms a partnership

with one of his old students, Jesse, who has some experience with the drug scene.

Now Walter and Jesse are preparing for their first "cook" (the process of making the drugs). They have a list of $N \leq 100\,000$ items they need for the cook, but they realized they have to be very careful when going to the store. The reason is that it may be suspicious to buy one of $M \leq 100\,000$ pairs of items, like cold medicine and battery acid, in the same trip.

They decide to divide the items among themselves, so that each of them can go one trip to the store and buy their share of items without the risk of anyone becoming suspicious. Help them find such a division, or tell them that it is not possible.

Solution. In graph terms, the problem is called two-coloring. We can construct a graph with the items as vertices and the forbidden pairs of items are edges. The problem becomes to color each vertex either red or blue (meaning that Walter or Jesse buys the item, respectively) such that no edge (a forbidden pair) connects two vertices of the same color.

To solve the problem, we should look at what constraints the color of a vertex puts on colors in general. We can always pick a random vertex and color it blue (by symmetry) to start with. If a vertex is blue, all its neighbors must be red for the graph to be two-colored. Similarly, a vertex that is painted red must have only blue neighbors. This coloring can be constructed recursively by a DFS:

```
1: procedure TwoColor(vertex at, color)
2:     if coloring[at] = uncolored then
3:         coloring[at] = color
4:     else if coloring[at] ≠ color then
5:         Answer "impossible"
6:     else
7:         return
8:     for every neighbor next of at do
9:         TwoColor(next, the opposite of color)
```

□

Exercise 8.22. Prove that a graph is two-colourable if and only if it contains no odd-length cycles.

Problem 8.23.
Hoppers hoppers

8.5 Trees

On a special kind of graph, the DFS can actually solve the single source shortest path problem. This is the case when, for any pair of vertices, there is **exactly one** path between them. The DFS paths from a certain source must then be the shortest ones. These graphs have a simple characterization.

Definition 8.8 A **connected** graph that does not contain any cycles is called a ***tree***. A graph where all connected components are trees is called a ***forest***.

Since a cycle trivially has two paths between each pair of vertices on it, the lack of them is a necessary condition for the path between any two vertices to be unique. We leave the other direction as an exercise.

Exercise 8.24. Prove that in a tree, there is exactly one path between any pair of vertices.

Exercise 8.25. A tree vertex with degree 1 is called a ***leaf***. Prove that a tree with at least 2 vertices has at least 2 leaves.

Exercise 8.26. Prove that a tree with n vertices have exactly $n - 1$ edges.

Exercise 8.27. In a graph on n vertices, a subset of $n - 1$ edges that is also a tree is called a ***spanning tree*** (it necessarily connects all vertices in the graph, i.e. it *spans* the entire graph). A connected graph always has a spanning tree. Show how to find it using the DFS.

Trees are represented using adjacency lists in most cases, since they are the most sparse (connected) graphs possible. The input formats vary quite a bit though, and may require some conversion to the adjacency list format.

Many graph problems can be solved faster and easier when dealing with trees rather than general graphs. Most of the well-known NP-complete graph problems with only known exponential-time algorithms instead takes polynomial time on trees.

Tree Diameter

In a graph, we compute the distance $d(a, b)$ for every pair of vertices. The largest such distance is called the ***diameter*** of the graph. Given a *tree*, compute its diameter.

Solution. In a general graph, the diameter is normally found by performing a BFS from each vertex to find the distances between every pair of vertices and taking the maximum of all distances. This gives a $\Theta(V(V + E))$ algorithm.

In trees, the path uniqueness property gives us a substantial speedup compared to the general case. The maximum distance is the longest overall path in the tree. While finding the longest path in general graphs is NP-complete, for trees two applications of the DFS is enough. To find one of those endpoints, consider the situation in Figure 8.11 with a diameter a-b. Any vertex f on the diameter must have one of a and b as its furthest away vertex v. Otherwise, at least one of a-f-v or b-f-v forms a path that is longer than a-b. It may be the case that one of those paths have the same length as a-b, but then v is the endpoint of another diameter (it is not necessarily unique). If we knew one endpoint a of the longest path, a single DFS is enough to find the other one – it is by definition the vertex furthest away from a.

The same is true for all other vertices too. Consider the subtree formed by a vertex on the diameter, for example f, and all vertices in one of the connected subtrees for which it

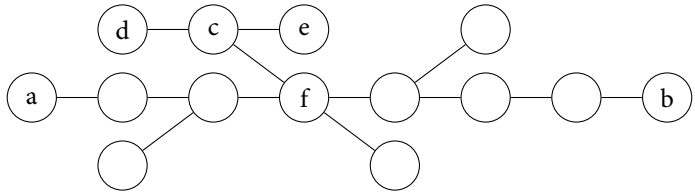


Figure 8.11: A tree with diameter a-b.

is the closest vertex on the diameter, such as d, c and e. If the longest path of some vertex d in the subtree goes through f, the other end of this path must be the furthest vertex from f, known to be a diameter endpoint. If it does not go through f, the furthest vertex from d is instead within the subtree.

Assume that this furthest vertex is e. The length of this path is bounded by $d(d,e) < d(d,f)+d(e,f)$. The distance $d(e,f) \leq d(a,f)$ where a is the diameter endpoint closest to f, or else $d(e,b) = d(e,f)+d(f,b) > d(a,f)+d(f,b) = d(a,b)$ would give a longer diameter. Then $d(d,e) < d(d,f)+d(f,a) = d(d,a)$, so that a was actually further away from d than e after all.

The conclusion of this argument is that for any vertex, the vertex furthest away from it must be an endpoint of a diameter. This can be found using a DFS too, giving us our two searches.

The DFS is even easier to implement for trees. Since there are no cycles, the only way a DFS might try to visit a vertex twice is when checking the edge that the DFS entered the vertex from. This can be fixed by adding the previous vertex as a parameter to the DFS, avoiding the need for a global array to keep track of visited vertices.

```
1: procedure FurthestVertex(vertex at, vertex last)
2:     (distance, furthest) ← (0, at)
3:     for every neighbor next of at do
4:         if next ≠ last then
5:             (nextDistance, nextFurthest) ← FurthestVertex(next, at)
6:             if nextDistance + 1 > distance then
7:                 (distance, furthest) ← (nextDistance + 1, nextFurthest)
8:     return (distance, furthest)
```

□

Problem 8.28.

Adjoin the Networks	adjoin
Krtica	krtica

We have already seen another way of representing trees when studying heaps (Section 6.4.2). Here, we formalize and generalize this representation.

Definition 8.9 A ***rooted tree*** is a tree where one vertex is selected as the ***root*** of the tree. For a vertex, the single neighbor closer to the root is called its ***parent***, while the ones further away from the root are called its ***children***.

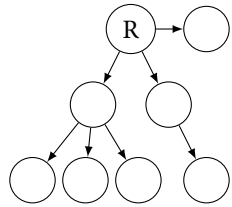


Figure 8.12: A rooted tree. Arrows points from parent to child. The root is marked *R*.

An arbitrary tree can be made rooted by selecting a vertex to be the root. Finding the parent and children of every vertex given the root is then done with a single DFS from the root. We have often talked about *subgraphs* and *subtrees* of graphs, as subsets of the vertices and edges in a graph. For rooted trees, the convention is that the subtree *of a vertex* is the subtree formed by a vertex and all its children, grand-children, grand-grand-children and so on.

Fuzzy Family Tree – fuzzyfamilytree

Your friend Sue B. Tree, of the Rootingford Trees is facing a problem. Her clan of well-known algorithmic problem solvers have a large painting of their family tree hanging on the wall, tracing the lineage of the $n \leq 100\,000$ family members all the way back to Enar E. Tree himself, known for the invention of the B-tree data structure. After some 32 generations or so, many names on the painting have faded so much that nobody knows exactly which vertex in the family tree belongs to the founder of the clan.

Sue has managed to read the names of some adjacent pairs of names for which she knows which one of them is the parent of the other. Given the layout of the family tree and the list of known parental relations, can you help Sue figure out what vertices in the tree could possibly represent Enar E. Tree? It is known that each person only have a single parent within the clan (causing cycles in family tree is grounds for immediate excommunication, even after 32 generations).

Solution. Formulated in terms of graphs, we are given a tree and want to find the vertices that may be roots in this tree, given some pairs of vertices that must be the parent and child of each other in a rooting. The straightforward $\Theta(n^2)$ solution is to try all possible rootings and then verify that each relation holds. This is too slow for such large n however.

Let us instead ask exactly what it means to have one of these parent-child constraints. Can we easily characterize necessary and sufficient conditions for a possible rooting? In Figure 8.13 we have drawn such a constraint, where p is known to be the immediate parent of a c. Since p by definition is closer to the root than c, we know that no vertex to the left

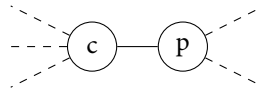


Figure 8.13

of c can be the root. The path to any such vertex from p goes through c, so p would be further away from it. Symmetrically, p is closer to any vertex to the right of p. This means that the roots that do not satisfy this particular constraint are exactly those that in the subtree of c if p was the root.

This gives us two paths forward – do we mark all valid roots for the constraint and answer with the roots valid for every constraint, or do we mark all invalid roots and instead answer with the roots that was not invalid for any constraint? The first option requires us to mark a linear number of vertices as valid for each constraint, while we are theoretically able to only mark each invalid vertex once in the other approach. The following insight asks you to prove how we would do this.

Exercise 8.29. Assume that for every constraint (p, c), we perform a DFS from c to mark roots as invalid. Prove that whenever the DFS encounters a vertex v already marked as invalid, all vertices that would be marked by recursing into v are already marked as invalid.

This means that all the depth-first searches amortized over all constraints takes only linear time. □

Problem 8.30.

Kitten on a Tree	kitten
Marbles On A Tree	marblestree

8.6 Topological Sorting

A directed graph is the data structure of choice for encoding different kinds of dependencies among objects. For example, construction projects consist of a large number of steps. Many can be done in parallel, but some steps have a strict happens-before relationship. When building a house, you can plan the interior decor as soon as the architectural plans of the house are finished, but you can never start mounting the roof before you put up the load-bearing walls! When this is modeled as a directed graph, we take the various stages in the process as vertices and draw an edge from u to v if u depends on v.

Given a dependency graph, in what order should we perform the actions if we want to complete them one at a time? This is a question of real, practical concern. In software engineering, the compilation of software in many languages requires performing interdependent actions one at a time in an order such that an action is performed only after all its dependencies. We call an ordering of this kind a ***topological ordering*** of the graph and

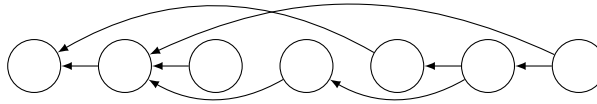


Figure 8.14: A topologically ordered directed graph.

the process of finding it ***topological sorting*** (see Figure 8.14 for an example). Intuitively, a graph should have a topological ordering as long as there are no actions that depend on each other in a cycle (that the graph must lack cycles is clearly a necessary condition, at least). These graphs – directed without cycles – are called ***directed acyclic graphs***, normally abbreviated *DAG*.

Exercise 8.31. Prove that if a directed graph does not have any cycles, it has a topological ordering.

Topological Ordering

Given a directed graph with V vertices and E edges without cycles, find a topological ordering of it.

Solution. An ordering is easy to find in theory: choose as first vertex any one without an outgoing edge. This gives us an $O(V^2)$ algorithm – find such a vertex by looping over them all, add it to the ordering and remove any edges going into it (with some careful graph representation to avoid an $O(VE)$ algorithm).

The general approach is correct, and can with small improvements be made linear-time. To avoid looping over all vertices to find one without outgoing edges, we keep a queue with vertices that currently have no outedges. When a vertex is added to the ordering we remove some edges, so new vertices may have to be added to the queue if we remove their last outgoing edge. To quickly find the set of edges to remove when a vertex is added to the topological ordering we store the reverse of all edges instead, so that each vertex has a list of its incoming edges. The final trick is to not remove any edges explicitly – the above algorithm only cares about the *number* of outgoing edges for each vertex. The final algorithm bears some resemblance with the BFS.

```
1: procedure TopologicalSort(graph V, E)
2:     outdegree ← int vector of size |V|
3:     inedges ← vector of vertex vectors of size |V|
4:     for every edge v → u in E do
5:         increment outdegree[v] by 1
6:         inedges[u].append(v)
7:     ordering ← vertex vector
8:     q ← vertex queue
9:     add all v where outdegree[v] = 0 to q
```

```
10:     while q is not empty do
11:         v ← q.pop()
12:         ordering.append(v)
13:         for u in inedges[v] do
14:             decrement outdegree[u] by 1
15:             if outdegree[u] = 0 then
16:                 q.add(u)
17:     return ordering
```

□

Problem 8.32.

Reactivity Series	reactivity
Brexit	brexit
Build Dependencies	builddeps

Exercise 8.33. A ***tournament*** is a directed graph where for every pair of vertices v and u exactly one of (v, u) or (u, v) is an edge. Prove that if a tournament has cycles, its shortest cycle has length 3.

Additional Exercises

Problem 8.34.

Lava	lava	
Pub-lic Good	pubs	
Frozen Rose-Heads	frozenrose	
Molekule	molekule	
Distinctive Character	distinctivecharacter	
Torn To Pieces	torn2pieces	
Brexit Negotiations	brexitnegotiations	
Pokémon Ice Maze	pokemon	
Conservation	conservation	
Conquest Campaign	conquestcampaign	
Mall Mania	mallmania	
Mårten's DFS	martensdfs	*(all subtasks)*
Flight Planning	flight	
Amanda Lounges	amanda	
Import Spaghetti	importspaghetti	
Landlocked	landlocked	
Who's the Boss?	whostheboss	
Through the Grapevine	grapevine	
Fountain	fontan	
On Average They're Purple	onaveragetheyrepurple	

Notes

This chapter is but a tiny peek into the area of algorithmic graph theory. The tools we picked up here are mostly for the purposes of solving easier ad-hoc graph problems and supporting us in developing other kinds of techniques. Graph theoretic algorithms are postponed to Chapters 14 and 15 when we have built up the necessary knowledge to tackle them. Solution methods for graph theory is a true potpourri of a wide range of graph theoretic algorithms, advanced data structures, a large bag of common tricks and raw mathematical problem solving.

For further resources, *Graph Theory* [13] by Reinhard Diestel is widely acknowledged as the go-to book on more advanced concepts. The book is freely available for viewing at the its home page[1]. We save reading tips into algorithmic graph theory until we revisit graph theory.

[1] `heap.link/diestel-graph-theory`

Part II

Common Techniques

Chapter 9

Brute Force

Many problems are solved by testing a large number of possibilities. For example, chess engines work by testing countless variations of moves and choosing the ones resulting in the "best" positions. This approach is called ***brute force***. Brute force algorithms exploit that computers are fast, resulting in you having to be less smart. Just as with chess engines, brute force solutions might still require some ingenuity. A brute force problem might have a simple algorithm that requires a computer to evaluate 2^{40} options, while some deeper analysis might be able to reduce this to 2^{20}. This would be a huge reduction in running time. Different approaches to brute force may be the key factor in reaching the latter case instead of the former.

This chapter deals with ***optimization problems*** and ***search problems*** that can be formulated in the following way. In them, we are given some ***search space*** of possible solution candidates S and a ***value function*** f. The goal is to find a solution x in S that maximizes $f(x)$, i.e. to optimize the function. For search problems, we typically want to find a solution that the problems defines as "valid". A lot of algorithmic problems can be formulated this way, as we see in this chapter. A famous search problem is the NP-complete *Hamiltonian cycle* problem: given a graph, find a cycle that visits every vertex exactly once. Its optimization variant the *Traveling Salesman Problem* (TSP) is perhaps even more famous: in a *weighted* graph, find the Hamiltonian cycle of least weight. TSP has many practical applications, for instance logistics companies that want to minimize the total distance traveled to deliver a set of packages. The search space for TSP is the set of all Hamiltonian cycles, with $f(x)$ equal to the weight of the cycle.

Brute force is an umbrella term for all solution steps that more-or-less naively try many options. The most obvious one is to check the entire search space S for the best solution, which is slow for large S. This and the two following chapters (on Greedy Algorithms and Dynamic Programming) develop techniques that exploit the particular structures of problems to perform smarter searches than this.

9.1 Generate and Test

Before we work on techniques to make brute force solutions smarter, we start with ways of solving optimization and problems at all. Our first method is ***generate and test***. This brute force strategy is based on *generating* solutions – naively constructing candidate solutions to a problem – and then *testing* them – evaluating the value function on them

and removing any potentially invalid solution candidates that were accidentally generated. It is applicable whenever the number of candidate solutions is quite small.

Maximum Clique – maxclique

In a graph, a subset of vertices form a ***clique*** if each pair of vertices is connected by an edge.

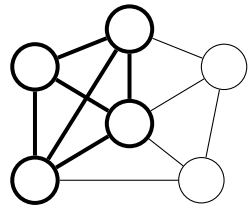


Figure 9.1: A graph with a clique of size 4 highlighted.

Given a graph with V vertices, determine the size of the largest clique.

Solution. This problem is one of the so-called NP-complete problems we mentioned in Chapter 5. Thus, a polynomial-time solution is currently out of reach. We solve the problem in exponential time for $V \leq 15$.

Is a generate and test approach suitable? To know, we must first define what our candidate solutions are. In this problem, only one object comes naturally: subsets of vertices. For every vertex subset candidate, we want to test whether it is a clique and among those choose the largest one.

In the maximum clique problem, there are only 2^V subsets of vertices – a quite small number. Given a vertex subset, we can verify whether it is a clique in $O(V^2)$ time by checking if every pair of vertices in the candidate subset has an edge between them. To perform this check in $\Theta(1)$ time, we store the graph as an adjacency matrix. This gives us a total complexity of $\Theta(2^V \cdot V^2)$ in the worst case. According to our table of approximate allowed input sizes for various complexities (p. 71), this should be fast enough for $V = 15$.

When coding a solution like this, you typically use bitwise operations to generate all the subsets of a set (p. 87). Since this is a common type of brute force, we include a C++ solution rather than pseudo code to illustrate how this is done.

Maximum Clique

```
int maxClique(int V, const vector<vector<bool>>& adj) {
    int largest = 0;
    for (int subset = 0; subset < (1 << V); subset++) {
        bool isClique = true;
        for (int i = 0; i < V; i++)
            if ((subset & (1 << i)) != 0)
                for (int j = 0; j < V; j++)
                    if (i != j && (subset & (1 << j)) != 0 && !adj[i][j])
                        isClique = false;
```

```
10          if (isClique)
11              largest = max(largest, __builtin_popcount(subset));
12      }
13      return largest;
14  }
```

□

Exercise 9.1. The maximum clique algorithm can be made to run in $\Theta(2^N \cdot N)$ by representing neighborhoods using bitsets. Adapt the above code to do this.

This kind of brute force problem is often easy to spot. There will be a very small input limit on the parameter you are to brute force over. Solutions are often subsets of some greater set, or all combinations of smaller sets.

Problem 9.2.

Maximum Clique	maxclique	*(for 2 points)*
4 thought	4thought	
Lifting Walls	walls	
Font	font	

Let us look at another example of this technique on a non-optimization problem, where solution candidates are not just subsets.

The Clock – klockan

By Pär Söderhjelm. Swedish Olympiad in Informatics 2004, School Qualifiers.

When someone asks you what time it is, most people respond "a quarter past five", "15:29" or something similar. If you want to make things a bit harder, you can answer with the angle from the minute hand to the hour hand, since this uniquely determines the time. However, most people are not used to this way of specifying the time, so it would be nice to have a program which translates this to a more common format.

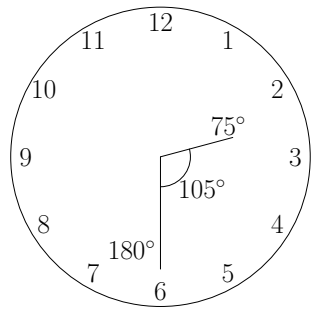


Figure 9.2: The angle between the hands at time 02:30.

We assume that our clock have no seconds hand, and only displays the time at whole minutes (i.e., both hands only move forward once a minute). The angle is determined by starting at the hour hand and measuring the number of degrees clockwise to the minute hand. To avoid decimals, this

angle is specified in tenths of a degree.

Given the angle $0 \leq A < 3600$, find the current time.

Solution. It is difficult to come up with a formula that gives the correct times as a function of the angles between the hands on a clock. Instead, we can turn the problem around. If we know what the time is, can we compute the angle between the two hands of the clock?

Assume that the time is currently h hours and m minutes. The minutes hand is then at angle $\frac{360}{60}m = 6m$ degrees clockwise from straight up. Similarly, the hour hand moves $\frac{360}{12}h = 30h$ degrees clockwise after h whole hours, with an extra $\frac{360}{12}\frac{1}{60}m = 0.5m$ degrees added due to the minute. While computing the current time directly from the angle is difficult, computing the angle from the current time is easy.

The generate and test solution is to compute the angle at each of the $60 \cdot 12 = 720$ possible times and test which one matched the given angle:

```
1: procedure Clock(A)
2:     for h ← 0 to 11 do
3:         for m ← 0 to 59 do
4:             ▷ Angles are 10'ths of degrees.
5:             hourAng ← 300h + 5m
6:             minuteAng ← 60m
7:             angBetween ← (minuteAng − hourAng + 3600) mod 3600
8:             if angBetween = A then
9:                 return h:m
```

□

Exercise 9.3. Can there be two times that produce the same angle? If yes, give an example. If no, prove that there are no two such times.

Competitive Tip

Competitions sometimes pose problems which are solvable quite fast, but where a brute force algorithm works just as well. Code the simplest correct solution that is fast enough, even if you see a faster one.

Problem 9.4.

The Clock	klockan	*(all subtasks)*
All about that base	allaboutthatbase	
Perket	perket	

9.2 Backtracking

Backtracking is a variation of the generate and test method that serves two different purposes. First, it is a way to construct candidate solutions recursively when they have a more complicated structure. For example, consider our solution to the maximum clique problem, where we generated candidate subsets using bitsets. An alternative would be to

go through each vertex one at a time, deciding whether to include it in the current clique candidate or not in a recursive fashion.

A recursive procedure of this kind does not add many lines compared to the bitset strategy, but is in practice significantly slower due to the overhead of function calls:

```
1: procedure RecursiveSubset(at, N, I)
2:     if at = N then
3:         output I
4:         return
5:     RecursiveSubset(at + 1, N, I)
6:     add at to I
7:     RecursiveSubset(at + 1, N, I)
```

The procedure keeps track of the number of vertices *at* we have taken this decision for so far, as well as the set of vertices I that we have decided to include in the subset in this recursive branch. The recursion is started with RecursiveSubset$(0, N, \{\})$.

Exercise 9.5. Compute the total number of recursive calls made by the function.

Secondly, the recursive nature of backtracking allows us to integrate the testing step directly into the generation step through ***pruning***. This means that we try to exclude invalid candidate solutions, or candidate solutions we somehow know can never beat a better one that we have already found, by testing partial solutions at each recursive step, rather than waiting until the very end. Once a partial candidate is identified as being infeasible, the backtracking can stop early. The effect this has on the recursion is illustrated in Figure 9.3.

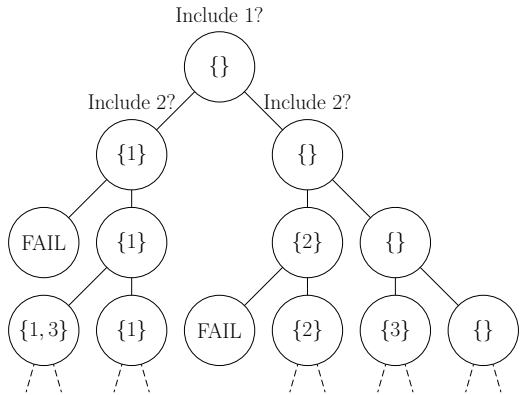


Figure 9.3: Recursive backtracking over subsets. The subsets $\{1, 2\}$ and $\{2, 3\}$ were invalid partial candidates that were pruned from the recursion.

If there are much fewer valid partial candidates than total candidates that a generate and test approach would check, this saves time. The general guideline is that backtracking

works best when we can:

- construct candidate solutions recursively,
- quickly determine whether a partial candidate solution can possibly be completed to a candidate solution, and
- the number of valid partial solutions is sufficiently small.

We look at three representative backtracking problems to build understanding of the method. In the first problem, we use backtracking mostly as a smarter means to generate all the candidate solutions, and they are so few that this is fast enough without any pruning. In the following one, we must figure out some more difficult pruning heuristics, and they unfortunately don't result in any easily proven bounds that gives us confidence in a solution. The final problem is the most difficult to solve, but we are able to prove that the pruning we end up with is good enough.

6-cliques

Given a graph with 45 vertices, compute the number of cliques with at most 6 vertices in it.

Solution. The bitset generation of subsets doesn't work here since there are 2^{45} of them – too many to test. A smarter generation strategy is to only generate subsets with at most 6 vertices and then test if they are cliques. For 45 vertices there are only 9 531 040 such subsets (in Chapter 18 you learn how to count them), so this approach would be much faster. The recursive method we showed to generate all subsets only need a slight modification:

```
1: procedure 6Subsets(at, N, I)
2:     if at = N or I has size 6 then
3:         check if I is a clique
4:         return
5:     6Subsets(at + 1, N, I)
6:     add at to I
7:     6Subsets(at + 1, N, I)
```

This solution does more recursive calls than one for each subset, so it's not obviously fast enough. To analyze it, we should first realize that I has size at most 5 except for in the final recursive call when the 6'th element is added. There are 8 145 060 subsets of size 6, to which we need to add the calls where $|I| \leq 5$. For each such I, the function is called with at most 45 different values of N. There are 1 385 980 subsets of size at most 5, so the number of extra calls is bounded by 62 369 100, giving us less than $\approx 70 \times 10^6$ recursive calls.

To avoid extra costs in verifying if a subset is actually a clique, we should integrate neighbor checking into the recursive function when a new vertex is added to I, preferably using bitsets (Exercise 9.1). □

Problem 9.6.

Class Picture	classpicture
Geppetto	geppetto
Map Colouring	mapcolouring

We move on to the kind of backtracking problem that is more of an art than problem solving. Here, the goal is to come up with smart enough constraints for when a partial solution should not be checked further. It is hard to know exactly when you should be happy with the pruning strategy you have, but testing your code on cases you think should be the worst-case for your solution is often good enough.

Semi-Magic Knight Tour – magicalmysteryknight

In chess, the knight is a piece that moves one step either horizontally or vertically, and two steps in the other direction (for example two steps up and one step to the left) for a total of 8 moves. On a normal 8×8 chess board, the knight is capable of starting anywhere on the board and through 63 moves visit every square on the board. This is called a knight tour.

A semi-magic chess board is one where each integer 1 through 64 is written in one of the squares, such that each row and column have the same sum – 260.

These two concepts can be combined to form a *semi-magic knight tour*. A knight is placed somewhere on the chess board and the integer 1 is written on that square. The knight now performs a tour on the chess board, writing in turn the integers 2, 3, . . . , 64 on the squares it visits. If these numbers form a semi-magic chess board, this was a semi-magic knight tour.

You are given a chessboard with at least 19 distinct integers between 1 and 64. The integer 1 is always present on the board. Find any semi-magic knight tour that has the given numbers on the given squares. It is guaranteed that there exists at least one tour that agrees with the given numbers.

Solution. This is a fairly standard backtrack-and-prune problem, where the main difficulty lies in knowing how to prune. The backtracking itself is done by recursively constructing all knight tours from the starting square given in the problem. If the i'th square of the knight is given by the input, we have to make sure that the $(i-1)$'th move we make connects to that square (which might allow us to fast-forward moves if the $(i+1)$'st square is also fixed, and so on).

From here on, you just let your creativity flow in finding optimizations. The first things that should be looked at are the sums of each row and column. We know what they *should* sum up to, and the more moves that are made, the harder it should be to find numbers that actually sum up to the right thing for a given row or column. This makes the sum constraints the obvious thing to attack in a partial solution.

The simplest constraint is that if a row has sum S and k squares yet to be visited, we must have $S + 64 \cdot k \geq 260$ – a square can never have a larger value than 64, so the left-hand side is an upper bound on the attainable row sum. This is a very lax bound that can be improved further. We cannot fill all the empty squares with 64, only a single one. A better upper bound would be $64 + 63 + \cdots + (64 - k + 1)$. To improve the bound even more, think

about how the knight moves. It can never visit the same row twice in a row, so the empty squares in the row can only be filled with every other number, giving us the upper bound $64 + 62 + \cdots + (64 - 2(k-1))$. A similar lower bound can be made that takes into account what integer the knight writes next on its tour.

There are several variants of the above pruning heuristic. Most of them turns out to degenerate into the above in the special case where the squares fixed in the input are the 19 first on the tour. This is, at least for almost every heuristic the author could conceive of, the worst case. Intuitively, the complexity of recursive solutions increases with the number of consecutive decisions that have to be made correctly, so for tour constructions these are typically worst cases. In this particular problem, having the 18 first and the final square be fixed can also be a bad case, if there are several possible final squares and each one requires some specific sequence of early moves.

An other heuristic that unfortunately is negated by the suggested worst case is to check how many moves are required to move from the current position of the knight to the next square fixed by the input. If this is larger than the number of moves until that fixed square, we can immediately backtrack.

A heuristic that is very powerful for the case we proposed is checking if there are *isolated* and *dead-end squares*. We call a square isolated if there are no free squares that the knight could jump into the square from, and dead-end if there is only a single such square. A partial solution is clearly not allowed to have an isolated square since the knight can never visit it. Similarly, a dead-end square must be the last one the knight visits, since it can't move away from it. This means that there can be at most one dead-end square. Checking for these two conditions turns out to slow down random test cases a lot since it is an expensive heuristic to compute, but is very helpful in the case where only the initial squares are fixed because the solution backtracks for a long time in partial candidates with such squares that can never be visited. Together with the best version of the sum heuristic, this pruning strategy – if implemented well in a fast language – is enough to get accepted[1]. □

It turns out that there are only 140 valid semi-magic knight tours. When solving the problem outside of a contest situation (the problem appeared in a 5-hour contest) it is possible to spend a few days of compute time to generate all tours ahead of time and hard-code them in your program. For some problems, this is a legit solution approach even during a contest. A brute force solution may be much easier to solve by precomputing some values that only takes maybe a minute to compute ahead of time. For instance, computing the n'th value of some simple recursive formula like the Fibonacci number F_n for a large n, say 10^9, is much faster if you precompute every millionth value. Any given n is at most a million computation steps away from the precomputed ones. This is better

[1] In the contest the problem was given in, the first heuristic was enough, but stronger problem instances were later added to the online version on Kattis.

than having to compute the 10^9'th value from the base cases.

So far, backtracking has seemed straightforward, even if we needed some smart pruning in the last problem. A popular type of backtracking in contests today are variants of NP-complete problems where some extra constraint is added that allows for faster solutions than those for the general case. In this next problem, we are given a special case of the ***independent set*** graph theory problem where the degree of every vertex is at most 4. An independent set is the dual of a clique: a subset of vertices where *no* pair of vertices in the set is adjacent.

Basin City Surveillance – basincity

By Pål G. Drange and Markus S. Dregi. NCPC 2014. CC BY-SA 3.0. Shortened.

Basin City is known for her incredibly high crime rates. The police see no option but to tighten security. They want to install exactly $k \leq 15$ traffic drones at some of the $n \leq 100\,000$ intersections to observe who's running on a red light. If a car runs a red light, the drone will chase and stop the car to give the driver an appropriate ticket. The drones are quite stupid, however, and a drone will stop before it comes to the next intersection as it might otherwise lose its way home, its home being the traffic light to which it is assigned. The drones are not able to detect the presence of other drones, so the police's R&D department found out that if a drone was placed at some intersection, then it was best not to put any drones at any of the neighboring intersections. As is usual in many cities, there are no intersections in Basin City with **more than four** other neighboring intersections.

The drones are government funded, so the police force would like to buy as many drones as they are allowed to. Being the programmer-go-to for the Basin City Police Department, they ask you to decide, for a given number of drones, whether it is possible to position exactly this number of drones.

Solution. At a first glance, it is not even obvious whether the problem is a brute force problem, or if some smarter principle should be applied. After all, 100 000 is a huge number of intersections! The problem becomes more reasonable after our first insight. If we have many intersections, all adjacent to very few other intersection, it should be easy to select k non-adjacent intersections. To formalize this insight, consider what happens when we place a drone at an intersection.

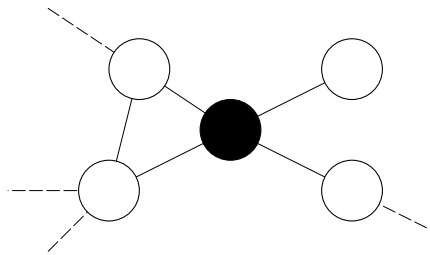


Figure 9.4: The intersections affected by placing a drone at an intersection.

By placing a drone at the intersection marked in black in Figure 9.4, at most five intersections are affected – the intersection we placed the drone at, along with its neighboring intersections. If we would remove these five intersections, we would be left with a new city where we need to place $k-1$ drones. This simple fact – which is the basis of a recursive solution to the problem – tells us that if we have $N \geq 5k-4$ intersections, we immediately know the answer is *possible*. The -4 terms comes from the fact that when placing the final drone, we no longer care about removing its neighborhood, since no further placements will take place.

We can therefore assume that the number of intersections is less than $5 \cdot 15 - 4 = 71$, i.e., $n \leq 70$. This certainly makes the problem seem much more tractable. Now, let us start developing solutions to the problem.

First of all, we can attempt to use the same algorithm as for the 6-clique problem., i.e. recursively constructing the set of our k drones by, for each intersection, testing to either place a drone there or not. Placing a drone at an intersection forbids us from placing drones at any neighboring intersection.

Unfortunately, the number of recursive options this tests is the number of subsets of at most 15 vertices of 70, a way too large number (in general the number of k-subsets of an n-set grows like $\Theta(n^k)$ for a fixed k, as we learn in Section 18.3.1). The values of n and k do suggest that an exponential complexity is in order, just not of this kind. Instead, something similar to $O(c^k)$ where c is a small constant would be a better fit. One way of achieving such a complexity would be to limit the number of intersections we must test to place a drone at before trying one that definitely works. If we could manage to test only c such intersections, we would get a complexity of $O(c^k)$.

Competitive Tip

In this problem, we tried to use the size of the parameters n and k together with the time limit to guide the kind of solution we need to design. While this works most of the time, note that this can sometimes be severely misleading – as this problem was before we realized that having 100 000 intersections was a red herring.

The trick, yet again, comes from Figure 9.4. Assume that we choose to include the black intersection in our solution, but still can not construct a solution. The only reason this case can happen is (aside from bad previous choices) that no optimal solution includes this intersection. What could possibly stop this intersection from being included in an optimal solution? At least one of its neighbors must be included in every optimal solution. Otherwise, we could just pick an optimal solution where none of the neighbors were included and add the intersection to it. Fortunately for us, this gives us just what we need to improve our algorithm – either a given intersection or one of its neighbors can be included in any optimal solution.

We have accomplished our goal of reducing the number of intersections to test for

each drone to a mere 5, which gives us a complexity of about $O(5^k)$ (possibly with an additional polynomial factor in n depending on implementation). This is still too much unless, as the jury noted, some "clever heuristics" are applied. Fortunately for us, we have two techniques left that speeds things up dramatically (even giving us a better time complexity).

The first trick is to assume that the graph we are working with is connected. In many problems the connected components of the graph are all independent of each other. This is also the case in this problem. Placing a drone on an intersection only affects its immediate neighbors and recursively their neighbors, but never a different connected component. Since components are independent, we can solve them separately by computing the maximal number of drones that can be placed in each component, until we have placed all the k drones.

How does the connectedness help us? Consider what happens when we have placed our first drone on the black intersection as in Figure 9.4. After excluding it and its adjacent intersections, the neighbors of the adjacent intersections are left with at most 3 neighbors instead. Recursing on one of those intersection results in only 4 choices, either placing a drone on the intersection itself one of its (at most 3) neighbors. In fact, after placing at least one drone in a connected component, it always has an intersection with at most 3 neighbors.

Exercise 9.7. In a connected graph with maximum degree 4, we remove a subset of the vertices. Prove that there is at least one remaining vertex of degree 3.

Taking this insight to its conclusion, we achieve a complexity of $O(4^k)$ by always branching on the intersection with the fewest neighbors.

While such an algorithm is significantly faster than $O(5^k)$, further improvements are possible. Again, let us consider under what circumstances a certain intersection is excluded from any optimal solution. We have already concluded that if this is the case, then one of its neighbors must be included in any optimal solution. Can it ever be the case that only *one* of its neighbors are included in an optimal solution, as in Figure 9.5 where we tested to place a drone at the black vertex rather than a as in Figure 9.4?

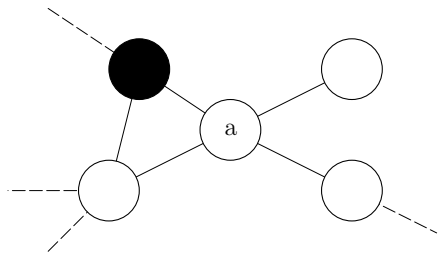


Figure 9.5: Placing a drone at a single neighbor of an intersection.

This is actually never the case. In any solution we can move the drone from the black intersection back to a if the black intersection was the only of a's neighbors with a drone. Now, we are basically done; for any intersection, there will either be an optimal solution including it, or (at least) two of its neighbors. Since an intersection has at most 4 neighbors, it has at most 6 pairs of neighbors. This means that the recursion requires at most $T(k) \leq T(k-1) + 6T(k-2)$ steps in the worst case, which is bounded by 3^k (a solution to the recurrence). A final improvement would be to combine this insight with the independence of the connected components. The second term of the time recurrence would then be a 3 instead of a 6 (as 3 neighbors make 3 pairs). Solving this recurrence gives us the complexity $O(2.31^k)$. □

Note that we could always reduce the number of vertices in the graph to $5k - 4$. As of writing, the best solution to the problem for the version with maximum degree 4 is $O(1.14^n) = O(1.91^k)$ [59], which is better than our solution but not by a very large amount. The currently best solution for the general case is instead $O(1.20^n) = O(2.49^k)$ [60], which our solution is provably better than.

Problem 9.8.

Domino	domino
Fruit Baskets	fruitbaskets
All Friends	friends

So, what is the take-away regarding backtracking? Start by finding a way to construct candidate solutions iteratively. Then, try to integrate the process of testing the validity of a solution with the iterative construction, in the hope of significantly reducing the number of candidate solutions that need evaluating. Finally, we might need to use some additional insights, such as what to branch on (which can be something complicated like the neighborhood of a vertex), deciding whether to backtrack or not (i.e. improving the testing part) or reducing the number of branches necessary (speeding up the generation part).

9.3 Parameter Fixing

Parameter fixing is one of the most widespread brute force methods since it's often used as a simplifying step in problems that are mainly about some completely other topic than brute force. It is the algorithmic equivalent of the mathematical problem solving technique known as *casework*, i.e. simplifying a problem by dividing it into sub-cases that are easier to solve. For example, a mathematical problem may become much easier to reason about if you knew the parity of some integer n in the problem, so you solve the problem for the cases where n is odd or even separately.

In algorithmic problem solving, we can use the fact that computers are able to work through a huge number of cases to greatly simplify some problems. Sometimes, we might

be able to take a parameter in a solution space and try *all* possible values. This is what we call "fixing" the parameter. The idea is that while any single choice of the parameter may be wrong, we are testing all of them and one of them is bound to be correct.

The path to a solution usually starts in the other end. Let us explain the technique with a problem.

Buying Books – kopabocker

By Pär Söderhjelm. Swedish Olympiad in Informatics 2010, Finals.

You are going to buy $N \leq 100$ books, and are currently checking the different $M \leq 15$ internet book stores for prices. Each book is sold by at least one book store, and can vary in prices between the different stores. Furthermore, each book store incur a postage fee if you order from it. Postage may vary between the various book stores, but it is always the same for a book store no matter how many books you decide to order. You may order any number of books from any number of the book stores. Compute the smallest amount of money you need to pay for all the books.

Solution. If we performed naive generate and test on this problem, we would probably get something like 15^{100} solutions, by testing every book store for each book. This is infeasible. So, why can we do better than this? There must be some hidden structure in the problem that makes testing all those possibilities unnecessary. To find this structure, we analyze a candidate solution as given by the naive generate and test method, i.e. an assignment from each book to a book store where we should purchase it from.

For the sake of example, let's assume that in this candidate solution, we purchased books from the book stores 1, 4 and 5. If we then purchased a book from store 4, but it was actually cheaper from store 1, we should have picked it from there instead. Thus there seems to be quite a bit of redundancy in this set of candidate solutions – a strong hint that we might have found some crucial insight. We *could* decide to use this fact to turn our generate and test into a backtracking algorithm, by pruning away any solution where we at some point decide to purchase a book from a book store that contains a book that we already purchased at a cheaper price. Unfortunately, this is easily defeated by giving most of the books equal prices at most of the book stores.

Let's use the insight differently. Our observation hints that making a choice for every book is not especially good, since choices elsewhere affects the optimality of an individual book choice greatly. Digging deeper, we find that we do not really have any choice in where we buy the books. During the course of the suggested backtracking, what matters is never what particular store we purchased a book from. Rather, it is only of interest what stores we have decided to use so far – this uniquely determines where we purchase every book from, since we are forced to buy the book from the cheapest store that we buy books from.

At this point, we are basically done. We have reduced the amount of information we need to know in order to easily find a solution "which book stores do we purchase from?". This parameter has much fewer possibilities – only 2^{15}. By testing each possible choice and

fixing it, we can immediately find the best candidate solution using the "cheapest store for each book" rule. Since we test every option for this parameter in the full set of candidate solutions, we must also test one that is given by a optimal solution. This results in the following pseudo code:

```
1:  procedure BuyingBooks(N, M, costs C, postages P)
2:      answer ← ∞
3:      for every subset S of bookstores do
4:          cost ← 0
5:          for every s in S do
6:              cost ← cost + P[s]
7:          for every book b do
8:              cost ← cost + min_{i in S} C[i][b]
9:          answer ← min(answer, cost)
10:     return answer
```

□

Alternatively, we could have come to the same insight by asking ourselves, can we brute force over the book stores? Whenever you have small parameters in a problem, this is a question worth asking. The other important mental heuristic to use is looking for things that would make the problem simpler if you were able to just remove them. For example, certain ordering constraints can be removed by brute forcing over all permutations of something, choices can be removed by brute forcing over subsets (like in Buying Books) and so on. In fact, we have already used the parameter fixing technique back in Section 8.3 when we found the shortest cycle in an undirected graph. There, we fixed the parameter "a vertex that lies on the shortest cycle" by trying all possibilities.

The parameter to brute force over is not always this explicit, as in the following problem which asks us to find all integer solutions to an equation in a certain interval.

Digital Root Equation

Given integers a ($|a| \leq 10\,000$), b ($1 \leq b \leq 10$), and c ($|c| \leq 10\,000$), find all integer solutions n to the equation

$$n = a \cdot dr(n)^b + c$$

where $dr(n)$ is the *digital root* digital root of n. To get the digital root of an integere, repeatedly replace it with its digit sum until it is less than 10. For example, $dr(919) = 1$: $9 + 1 + 9 = 19$, $1 + 9 = 10$, and $1 + 0 = 1$.

Solution. The only explicit object we have to brute force over is n. Unfortunately there are way too many possibilities for that to be feasible. If we bound n from the right hand side – the digital root is always between 0 and 9 – we see that it's always between $-10\,000 \cdot 9^{10} - 10\,000$ and $10\,000 \cdot 9^{10} + 10\,000$, we get roughly $7 \cdot 10^{13}$ possibilities to check. Instead, studying the equation a bit closer, we see that $dr(n)$ also varies as a function of n. This is helpful, since $dr(n)$ has far fewer options than n – there are only 10 possible digital

roots of any integer. Thus, we can solve the problem by looping over all the possibles values of $dr(n)$. This uniquely fixes our right hand side, which equals n. Given that n, we verify that $dr(n)$ has the correct value.

Whenever we have such a function, i.e. one with a large domain (like n in the problem) but a small image (like $dr(n)$ in the problem), this technique can be used by brute forcing over the image instead. This is actually what we did in the book store problem. Our function was then from a given candidate solution to the book stores used. Since the image of this function (all the possible subsets of book stores) was small, the problem could be attacked by brute forcing over the image of the function rather than the domain. □

Problem 9.9.

Nered	nered	
Multigram	multigram	
Shopping Plan	shoppingplan	
Milestones	milestones	
Candy	godis	*(all subtasks)*

9.4 Meet in the Middle

The ***meet in the middle*** (MITM) technique is a special case of the parameter fixing technique. The general theme is to first fix half of the parameter space and build some fast data structure that allows us to quickly find the best choice of parameters when we later brute force over the other half. Basically, we try to avoid the multiplicative effect of brute forcing the latter half for each choice of the first half with the additive effect of only brute forcing each half once. It is a space-time trade-off in the sense that we improve the time usage (testing half of the parameter space much faster), by paying with increased memory usage (to save the precomputed structures).

Subset Sum

Given a set of integers S, find a subset with a sum equal to T, or determine that no such subset exists.

Solution. In this classic NP-complete problem, a simple generate and test solution would have N parameters to brute force over. For each element of S, we either choose to include it in A or not – a total of two choices for each parameter. This naive attempt at solving the problem (which amounts to computing the sum of every subset) gives us a complexity of $O(2^N)$. While sufficient for e.g. $N = 20$, we can make an improvement that makes the problem tractable even for $N = 40$.

The trick to a faster solution is that the choices we make among individual parameters is highly independent. If we have decided which of the first $\frac{N}{2}$ elements to include in the subset, these choices together only put a single constraint on the remaining $\frac{N}{2}$ elements. If the sum of the elements chosen from that first half is U, the sum of the latter half must be

$T - U$ for the sum to be correct Thus, if we could quickly answer the question "can we choose the latter half of the integers such that they have a given sum?" we could solve the problem by fixing the first half of the parameters. Each constraint individually takes $O(2^{\frac{N}{2}})$ time to check if we use brute force. However, we can compute the answer for *all* such questions in one go by computing the sum of every subset of the latter half, which takes $\Theta(2^{\frac{N}{2}})$ time as well. The resulting sums and subsets can be inserted into a hash map, letting us determine if a sum can be formed using the last half of the elements in $\Theta(1)$ instead. Here is the space-time trade-off – we sacrifice an exponential amount of memory (a map of $2^{\frac{N}{2}}$ elements) to win an exponential speedup of $2^{\frac{N}{2}}$.

Initially the complexity might seem to be $\Theta(N2^{\frac{N}{2}})$ since it takes $\Theta(N)$ on average to compute the sums of all subsets of half of the elements. It's possible to reduce this to $O(1)$ by generating them in increasing order of subset size and computing sums by adding a single new element to an already computed sum, using the bitset trick `__builtin_ctz(x & ~x)` to find the index of the lowest set bit of the bitset `x`.

The pseudo code for our meet in the middle solution looks something like this:

```
1:  procedure SubsetSum(S, T)
2:      Lset ← the |S|/2 first elements of S
3:      Rset ← the elements of S not in Lset
4:      Lsums ← new map
5:      for each subset L of Lset do
6:          Lsums[the sum of L] = L
7:      for each subset R of Rset do
8:          remainder = T − the sum of R
9:          if Lsums contains the key remainder then
10:             return the union of Lsums[remainder] and R
11:     return impossible
```

□

Problem 9.10.

Closest Sums	closestsums
Celebrity Split	celebritysplit
Indoorienteering	indoorienteering
Rubik's Revenge in ... 2D!? 3D?	rubiksrevenge

To figure out if a meet in the middle solution is applicable, the constraints that the parameters of the first half put on the second half should be "simple" somehow. A meet in the middle solution is possible for the maximum clique problem as well, albeit it is slightly harder. The solution approach would ask the question: for a given clique constructed from the first $\frac{N}{2}$ vertices, what is the largest clique compatible with it (i.e., neighboring all the vertices in the first clique) that can be constructed from the $\frac{N}{2}$ latter vertices?

Answering this is relatively easy after generating all the cliques among the second half. If we have such a clique C, where all vertices neighbor the set A among the first half, the

cliques that C will be compatible with are exactly the subsets of A. To compute this quickly in $\Theta(N2^{\frac{N}{2}})$, we use a neat looping trick that has many similar uses:

```
1: procedure CliquePrecompute(N)
2:     largest ← map from vertex sets to integers
3:     for every clique C in the latter half of the vertices do
4:         compatible ← those of the first N/2 vertices adjacent to all vertices in C
5:         largest[compatible] ← max(largest[compatible], the size of C)
6:     for every subset S of the first N/2 vertices in decreasing order of size do
7:         for each subset S' with one element removed from S do
8:             largest[S'] ← max(largest[S'], largest[S])
```

The key is that the second for-loop iterates through all subsets S in decreasing order of size, which makes sure that every answer for a compatible set is propagated to all its subsets correctly.

Problem 9.11.
Maximum Clique maxclique *(for 4 points)*

Sometimes, meet in the middle is not immediately applicable. Perhaps there are some other constraints in the problem that prevents it from working. We end the chapter with an example of this in the form of a neat combination problem where we also need the parameter fixing technique.

Limited Correspondence – correspondence

By Greg Hamerly. ICPC 2012 Dress Rehearsal. CC BY-SA 3.0. Shortened.

Emil, a Polish mathematician, sent a simple puzzle by post to his British friend, Alan. Alan sent a reply saying he didn't have an infinite amount of time he could spend on such non-essential things. Emil modified his puzzle (making it a bit more restricted) and sent it back to Alan. Alan then solved the puzzle.

Here is the original puzzle Emil sent: given a sequence of $k \leq 11$ pairs of strings (a_1, b_1), (a_2, b_2), …, (a_k, b_k), find a non-empty sequence $s_1, s_2, \ldots, s_m$ such that the following is true:

$$a_{s_1} a_{s_2} \ldots a_{s_m} = b_{s_1} b_{s_2} \ldots b_{s_m}$$

where $a_{s_1} a_{s_2} \ldots$ indicates string concatenation. All the $2k$ strings contain only lowercase `a-z` and are at most 100 characters long. The modified puzzle that Emil sent added the following restriction: for all $i \neq j$, $s_i \neq s_j$.

You don't have enough time to solve Emil's original puzzle. Can you solve the modified version? First, determine if the puzzle has a solution or not. If it has, find the shortest one. If there are multiple such sequences, find the lexicographically first.

Solution. The original problem as posed by Emil[2] is called the Post correspondence problem and is an *undecidable* problem, i.e. there is no algorithm in the familiar sense that can

[2] Emil Post, an American mathematician

solve the problem in finite time.

Alan's added restriction to the problem, that if $i \neq j$, then $s_i \neq s_j$, means that each pair of strings may be used in the sequence at most once. The problem can then be solved in finite time. We can test all $k!$ permutations of the pairs, and walk through them to see if the two strings formed by the respective strings of all the pairs match. This would require around $k! \cdot k \cdot 100$ operations which is about $4 \cdot 10^{10}$ for the maximum $k = 11$ – too slow.

The big difference in this problem compared to the one where we could perform meet in the middle is that our selection of strings is highly *order dependent*. We can't arbitrarily split up our word pairs into a "first half" and a "second half" and attempt to combine them. After all, the correct solution might involve constructing a string where words from the two halves are intertwined. Thankfully, the last section showed us precisely how to deal with this obstacle. If an arbitrary choice is not good enough, we try *all* the choices. We fix the parameter that is the *subset of word pairs constituting the first half*. There are only 462 ways in which one can pick the first 5 word pairs out of a maximum eleven – a small price to pay.

Fixing parameters thus allows us to split up the words in two halves. This begs the question – for each possible permutations of the words in the first half, what constraint do they put on the second half? Let's check. Assume that a given permutation of the (currently fixed) first half of the pairs give a concatenated string of a's is the string S, and without loss of generality is shorter than the concatenation of the b's. First of all, S must be a substring of the concatenation of the b's – otherwise, concatenating the strings of the second half can't make them equal. Thus, we assume that the concatenation of the b's is ST.

Symmetrically, the concatenation of all b's in the second half is the string U, the concatenation of all the a's must be TU, to together make the string STU. Thus, the question is – can we order the words of the second part so that they create strings of the form U and TU for any U? This is precisely the kind of simple question that makes meet in the middle possible. To answer it quickly, we check the concatenations of each permutation of the 6 words in the second half. If one of them is of the form U and XU for some X, we store it in a hash map from X to the lexicographically smallest U that we found. With this map in hand, we can determine if there is a way to complete the strings formed by the first half in constant time.

In total, the cost is somewhere around about $462 \cdot (5! \cdot 5 + 6! \cdot 6) \approx 2 \cdot 10^6$ hash map operations, and $462 \cdot (5! \cdot 1000 + 6! \cdot 1200) \approx 4.5 \cdot 10^8$ individual character operations depending on the implementation. While the latter seems like a lot, solving the worst-case on the author's computer 5 times takes less than a second. □

Additional Exercises

Problem 9.12.
Key to Knowledge keytoknowledge

Holey N-Queens (Batman)	holeynqueensbatman	
Tautology	tautology	
Knights in Fen	knightsfen	
Committee Assignment	committeeassignment	
Circuit Counting	countcircuits	
Infiltration	infiltration	
Vase Collection	vase	
Maximizing your Pay	maximizingyourpay	
Boggle	boggle	
Maximum Clique	maxclique	*(all subtasks)*
Political Development	politicaldevelopment	
Folded Map	foldedmap	

Notes

A deeper dive into exact brute force solutions can be found in Exact Exponential Algorithms [19]. The book also discusses several techniques that are common within algorithm problem solving. Another neat technique with elements of brute force is *color-coding* [1] to solve certain graph theoretical problems.

A concept that often comes up in research and and occasionally in programming competitions, is that of *fixed parameter tractability*. A problem that might (as of now) only have exponential time algorithms, such as the maximum clique problem, may have a polynomial solution when fixing some parameter, such as the size of the maximum clique. Indeed, we saw that the problem of finding the largest clique of size at most 6 admitted a polynomial solution in the number of vertices. Parameterized Algorithms [12] provide a comprehensive toolbox for such problems.

We skipped several brute force search techniques that are normally discussed in algorithm text books, such as A*, IDA*, bidirectional search (which is basically meet in the middle during a BFS), and so on. Those kinds of methods are unusual nowadays in competitive programming, and as algorithmic problems solved for leisure somewhat uninteresting. Here we refer to chapters on searching and in your favorite AI fundamentals textbook, for example [43].

Chapter 10

Greedy Algorithms

In this chapter we look at another standard technique to solve some categories of search and optimization problems faster than naive brute force, by exploiting properties of *local optimality*.

From the outside, greedy algorithms look like heuristic solutions that just happen to work. Of course, there are mathematical reasons for why something that looks like a heuristic is correct. As algorithmic problem solvers, this means that greedy algorithms also require correctness proofs. As competitive programmers, we are instead happy as long as we (and the online judge!) are convinced that the idea is correct by intuition.

We are thus introduced to a third kind of algorithmic problem. Previously, we have almost exclusively worked with problems that either were mostly implementation exercises, or where we needed to come up with some fast algorithm but the correctness was obvious. Greedy problems on the other hand force us to not only see where our intuition leads us, but also to prove it correct. Problems have shown traces of this aspect (such as the tree diameter problem or the correctness of the pruning rules in *Basin City*), but greedy algorithms is the only category where provable correctness (rather than clever algorithmic or data structure tricks) is the focus of *every* problem.

10.1 Locally Optimal Choices

We start with an archetypical greedy problem, the *change-making problem*.

Change-making Problem, Denominations 1, 2, 5

Given an infinite number of coins of denominations 1, 2, 5, determine the smallest number of coins summing to $T \leq 10^{12}$.

Solution. We can use the tools from brute force to formulate a backtracking solution. A recursive function takes T and attempts to add a single coin of each type x to recursively try and form $T - x$ instead:

```
1: procedure MakeChange(T)
2:     if T = 0 then
3:         return 0
4:     answer ← 1 + MakeChange(T − 1)
5:     if T ≥ 2 then
```

```
6:        answer ← min(answer, 1 + MakeChange(T − 2))
7:    if T ≥ 5 then
8:        answer ← min(answer, 1 + MakeChange(T − 5))
9:    return answer
```

Like most backtracking solutions, this takes exponential time. It branches three times per call and each call decreases T by at most 5, so the recursion will have a depth of at least $\frac{T}{5}$. It thus has a (weak) lower bound of $\Omega(3^{\frac{T}{5}})$ which, for the T in the problem, is so large that the algorithm will never finish[1].

If we formulate a recursive formula such as those in Chapter 7:

$$\text{Change}(T) = 1 + \min \begin{cases} \text{Change}(T-1) & \text{if } T \geq 1 \\ \text{Change}(T-2) & \text{if } T \geq 2 \\ \text{Change}(T-5) & \text{if } T \geq 5 \end{cases} \tag{10.1}$$

(with base case $\text{Change}(0) = 0$) you might remember that it can be computed in linear time iteratively for increasing values of T since the recursive calls have already been computed, having smaller values of T. This is actually a sneak peek of the next chapter on dynamic programming, but unfortunately it doesn't help us here – T is too large even for a linear time complexity to be of any help.

Like all of our recursive problems so far, we can formulate change making as a sequence of choices. The choice here is, for a given T, should the next coin chosen be of value 1, 2 or 5? Which of these choices do we *think* is the best one? If you had to pay a friend T money from your infinite collection of coins, how would you do it? Intuitively (a word you will grow tired of hearing in this chapter), you try to pay with large denominations first. As long as we have the choice of paying with a 5 coin, we can try doing so. Once T goes below 5, we instead pick a 2 coin until $T < 2$, when we pay with a single one coin if we need to.

Even if this might not be optimal, the strategy should not be too bad. To confirm that we are on the right track, we start with a quick argument for why this can only be worse by a constant amount (i.e. not a constant factor, but a constant number of coins) than the optimal solution. The best solution never uses more than 5 coins of value 1 or of value 2, since they could then be replaced by 1 or 2 coins of value 5 for fewer coins in total. Thus we can never be more than 8 coins from optimality.

We can take this line of reasoning further. First, we never have to use more than a single 1 coin, since we can replace two of them with a single 2 coin. On the other hand, if we do use a 1 coin, we never want to use two 2 coins, as we could exchange one 1 coin and two 2 coins for a single 5 coin. If we don't choose a 1 coin, we can have either 0, 1 or 2 two coins. Picking a third 2 coin gives us the value 6, which only takes a single 1 and 5 coin to make.

[1] At least not before the heat death of the universe (or whatever model of ultimate fate of the universe you subscribe to), after which performing meaningful computation becomes slightly harder.

The possibilities for the number of 1 and 2 coins left are:

- no 1 or 2 coins: value 0,
- a single 1 coin: value 1,
- a single 2 coin: value 2,
- one 1 and 2 coin: value 3, or
- two 2 coins: value 4.

These values are also optimal for $T = 0 \ldots 4$, easily verified using the recursive formula. It's also exactly what choosing the largest coin at every T would produce – a very strong hint that we are on the right track. To tie everything up, one thing stands out in the list – none of the combinations exceed the value 5, so whenever $T \geq 5$ it must be optimal to use 5 coins until we reach one of those cases, i.e. exactly what our strategy does.

We have shown that no matter the value of T, the optimal choice is to always pick the largest coin that does not exceed T. The number of coins this strategy produces can be computed in constant time. □

Problem 10.1.

The Bus Card	busskortet	*(all subtasks)*
Minimal Fibonacci Sum	fibonaccisum	

A typical reaction after first seeing this problem is that the solution is obvious. *Of course* the right strategy must be to pay with the largest coin first. That's the choice that brings us closest to the goal at each time. Exercise 10.2 shows why this is a treacherous way of thinking.

Exercise 10.2. Prove that the greedy choice may fail if the coins have denominations 1, 6 and 7.

While sometimes incorrect, the sentiment "do what seems best at the time" is the core of greedy algorithms. We call those choices ***locally optimal***. They might not produce the correct solution (i.e. be the *globally optimal* choice), but at least seem like reasonable choices in the given situation. To make the point that greedy algorithms are best used when proven, we give you the following exercise.

Exercise 10.3. For the following problems and suggested greedy algorithm, give an example that proves it incorrect.

1. You have some books of three different widths 1, 2, and 3. They should be placed into a number of bookshelves, each of width L. How many bookshelves do you need?
   Greedy Algorithm: add one bookshelf at a time and fill it in the following way. First place as many books as possible of width 3 in it, then those of width 2, and finally those of width 1. Repeat this until all books are placed.

2. A postal service requires a different amount of workers every day of the week, $w_1, \ldots, w_7$, but the same amounts every week. A worker always do 3-day shifts. Shifts starting on the weekend wrap around onto the Monday and Tuesday of the following week. How many shifts do you need to schedule to ensure that there are enough workers every day?
   Greedy Algorithm: Let w_1 shifts start on Monday. For each following day i, add $\max(0, i - w_{i-1} - w_{i-2})$ shifts to that day.
3. On a rectangular grid with $H \times W$ squares (H and W are even), fill each square with one letter A, B or C such that there are a A's, b B's and c C's ($a + b + c = H \cdot W$), and no two horizontally or vertically adjacent squares have the same letter.
   Greedy Algorithm: assume that $a \geq b \geq c$. Order the squares in the following way: first those with coordinates (r, c) where $r + c$ is even, first in increasing order of r and then c. Then add the square with $r + c$ odd in the same order. Place an A on the a first of those squares, then a B on the b next squares, and then a C on the c next squares.

Proving the correctness of a locally optimal choice is sometimes very cumbersome. In the remainder of the chapter, we are going to look at a few standard greedy techniques and problems. Take note of the kind of arguments we are going to use – there are a few common types of proofs often used for greedy algorithms.

Competitive Tip

If you have the choice between a greedy algorithm and another algorithm (such as one based on brute force or dynamic programming) where both are equally fast, the non-greedy choice is better if you have a hard time proving the correctness of the greedy one.

On the other hand, if an greedy algorithm that you have some confidence in is very easy to code and you are not penalized for incorrect attempts you can save valuable contest time by skipping a proof.

Problem 10.4.

Falling Apart	fallingapart
Coloring Socks	color
Planting Trees	plantingtrees
Logland	logland

10.2 Extreme Values

So when do locally optimal choices work, and how do you find them? Naturally, to be able to discuss a *best* choice, there must be some ordering of the choices. The best choice is then one of the ***extreme values*** of that ordering. Typical extreme values are:

- the greatest or smallest element,

- the widest, shortest or leftmost interval,
- the vertex with greatest or smallest degree,
- the shortest or longest edge,

and so on. An occasionally helpful guiding principle for when to look for a greedy choice is when a problem just doesn't seem like it can be solved in any other way. This generally means that the problem looks like it must be solved in near-linear time (very common for greedy algorithms) but everything non-greedy would take more time.

Sequence – sequencereduction

By Jakub Radoszewski. Baltic Olympiad in Informatics 2007.

A sequence of $N \leq 10^8$ distinct integers $a_1, \ldots, a_N$ is to be reduced to a single integer. To do this, you can choose two adjacent integers a_i and a_{i+1} and remove the smaller one, called a *reduction operation*. This operation costs $\max(a_i, a_{i+1})$. Determine the minimum total cost of performing $N - 1$ such operations (after which the sequence has a single integer).

Solution. This problem is one of those where an experienced problem solver would immediately look for something greedy. You have so little time per operation to find the optimal choice that anything smarter than a greedy algorithm seems unlikely. We show several quite different solutions, all very natural. Which one you arrive at depends on from which direction the problem is attacked.

First, a small simplification: we add ∞ as the first and last elements of the sequence. This removes some edge cases in the solution, but does not change the answer (we would never want to perform an operation with them since they are so large anyway).

If we apply the principle of extreme values, some questions come to mind.

- What operation has the lowest cost?
- What operation removes the smallest element?
- What operation has the greatest cost?
- What operation removes the greatest element?

We start with the first question. Assume that an operation (a_i, a_{i+1}) has the lowest cost, and that $a_{i+1} > a_i$ so the cost is a_{i+1}. Since it is the lowest cost operation, we must also have $a_{i-1} > a_{i+1}$ or (a_{i-1}, a_i) would have a lower cost.

We claim that it must be optimal to perform this operation, i.e. removing a_i at the cost a_{i+1}. It is clear that a_i must be reduced with one of its two adjacent elements at some point. If we are to do it now, we should of course choose a_{i+1} since this is the smaller of the two. Could it ever be better to wait for a while before this reduction? The only thing that could change is that a_{i+1} is replaced at some point with a greater number, increasing the cost of removing a_i. On the other hand, no cost can be increased by removing a_i now, since it is never the cost in any reduction operation (as its two surrounding elements

can never decrease). This gives an immediate quadratic solution – find the lowest cost operation repeatedly and perform it.

To get to a slightly simpler solution, we can move on to the second question. In the above solution, we never actually used that the operation performed was the one with lowest cost. This assumption was only helpful in establishing a trio of elements $a_{i-1} > a_i < a_{i+1}$ with the conclusion that for three elements where this holds, we can reduce a_i with the smaller of its adjacent elements without making anything worse. Can we find such a trio faster than looking at all possible operations? Yes – the smallest a_i fulfills those two inequalities. By going through all the elements in increasing order and reducing them with their smaller neighbor we also get an optimal solution. Keeping track of how the sequence looks throughout these reductions is now a data structure question that can be solved by using sorted sets. That last step can be done in linear time, but the solution is $\Theta(N \log N)$ either way since it requires sorting. This is borderline acceptable for $N \leq 10^8$. Can we do even better?

There are two ways forward. One is through clever use of data structures. Sweep through the sequence one element at a time left-to-right and keep track of what elements we haven't removed yet (call this sequence P). If the next element in the sequence is greater than the last element of P, we know that the last element of P should be reduced either with the new element or the second last element of P, until the new element is smaller than the last element of P. Then, we add the new element to P. Once we run out of elements, we start performing reduction operations using the last two elements of P until we only have a single element left.

The other way is much cleaner, and involves looking at yet another extremal value: the greatest element. Consider all the elements to the left and right of it. It must be optimal to first reduce all of them to a single element before reducing them with the greatest element. If there were two elements a_i, a_j to the left and we reduce both of them together with the greatest element, we could lower the cost by first reducing them with each other. This gives us yet another solution – find the greatest element, perform the reduction of the intervals to the left and right of it, and then perform the final reductions. But this is not the solution we are after, for it too is $\Theta(N \log N)$ if implemented in the straightforward manner[2].

Instead, look at the second largest element, located, say, to the left of the greatest element. By the same reasoning, it too is reduced with twice, except *if it is adjacent to the greatest element* since there would be no elements in between them to be reduced with the second greatest element from the right. This is true for the third greatest element too, i.e. it is reduced with twice except if it is adjacent with one of the other two greater elements, for the same reason. The third greatest element might even be reduced with zero times if it is adjacent to both the greatest and second greatest element! In fact, if we keep this

[2]It is possible to do this in linear time, but showing that solution would turn this chapter into one on data structures. Look up *Cartesian trees* if you are interested.

reasoning going, we see that the i'th greatest element is reduced with:

- twice, if the adjacent elements are smaller,
- once, if only one adjacent element is smaller, or
- never, if both adjacent elements are greater.

This is simply the same number of times as there are smaller elements immediately adjacent to it, so this gives us a linear time solution. An alternative way of formulating this is that for every adjacent pair of elements a_i, a_{i+1}, the larger of them should be added to the cost. □

After looking at the previous problem one can almost wonder if it's possible to *not* solve it since an algorithm fell out wherever we looked.

Problem 10.5.

Fishmongers	fishmongers
Frosh Week	froshweek2
Assigning Workstations	workstations

10.3 Sorting and Exchanges

Many greedy problems involve performing some set of actions in an optimal order, or arranging a sequence in an optimal way. A standard way of attacking them is to define some ***measure*** for each action or element and assume that the correct ordering is to sort them by this measure. This is not surprising, since sorting is exactly what you get by repeatedly making locally optimal choices corresponding to the lowest measure value – remember the extreme value principle! Normally you come up with a reasonable measure by intuition, and prove its correctness by showing that any unsorted solution can be improved by swapping two out-of-order elements. When this is the case the sorted sequence must be optimal – it is the only one that can not be improved by swapping out-of-order elements.

This technique of showing that a non-greedy solution can be improved to what the greedy solution produces is a so-called ***exchange argument***. It is what we used to prove that an optimal solution to the coin change problem could be transformed to the greedy choice. Back in *Basin City* when studying brute force algorithms we used an exchange argument to show that a drone could always be placed on either a given intersection or two of its neighboring intersections – exchange arguments do not only belong in greedy problems.

Minimum Scalar Product – minimumscalar

Google Code Jam 2008, round 1A. CC BY 3.0

You are given two vectors $v_1 = (x_1, x_2, \ldots, x_n)$ and $v_2 = (y_1, y_2, \ldots, y_n)$ where $n \leq 200$. The scalar product of these vectors is a single number, calculated as $x_1y_1 + x_2y_2 + \cdots + x_ny_n$.

Suppose you are allowed to permute the coordinates of each vector as you wish. Choose two permutations such that the scalar product of your two new vectors is the smallest possible, and output that minimum scalar product.

Solution. Mathematical intuition says: big numbers multiplied by big numbers give big products, while big numbers multiplied by small numbers give small products. This suggests that the best solution is given by sorting one permutation in ascending order and the other one in descending order. To prove it correct, we use an exchange argument.

Suppose $x_1 \leq x_2 \leq \cdots \leq x_n$, and the optimal permutation of y_i is not in descending order. Then there are exists $i < j$ such that $y_i < y_j$. These two elements contribute to the scalar product with the terms $x_i y_i + x_j y_j$. What happens if we swap their places? The new contribution becomes $x_i y_j + x_j y_i$. The change to the scalar product is

$$(x_i y_j + x_j y_i) - (x_i y_i + x_j y_j) = (x_i - x_j)(y_j - y_i) \leq 0$$

since $x_i - x_j \leq 0$ while $y_j - y_i > 0$. The exchange did not increase the scalar product, so this new solution is at least as good. This means that given any solution – including the optimal – we can transform it to the greedy one without making it worse, so the greedy solution is optimal too. □

The solution is not exactly of the form that was promised. We gave a sequence of concrete changes from the optimal to the greedy solution, instead of claiming that only the greedy solution could not be improved by an exchange. This is because the new solution was not necessarily **strictly** better. To avoid this, we could have started with the assumption that the y_i was the optimal solution that required the fewest number of swaps to make it sorted. If that number was not 0, the swap would give us a solution just as good that required even fewer swaps, a contradiction.

A word of caution before we move on. Exchange arguments of this kind do not always work, even when the correct solution *is one where we sort according to some measure*. There may be several so-called *local optima* that can not be improved by such swaps. In these cases, other proof techniques must be used.

In the next problem, we are going to use a slightly weaker form of the exchange argument, where we instead prove that any solution not sorted according to our measure can be improved by swapping two *adjacent* out-of-order elements. The same thing applies: this is not always strong enough. There are problems with local optima when swapping adjacent problems, but where swapping arbitrary elements is sufficient.

Exercise 10.6. Consider the following problem: given a set of integers A, find an ordering $a_1, a_2, \ldots, a_n$ that minimizes $|a_1 - a_2| + |a_2 - a_3| + \ldots |a_{n-1} - a_n|$.

1. Prove that an optimal solution is to take the a_i in ascending order.

2. Find a set A and a solution candidate that can not be improved by swapping two adjacent out-of-order elements, but can be improved by swapping to arbitrary out-of-order elements.
3. Find a set A and a solution candidate that can not be swapping any two out-of-order elements.

The measure can be more complicated than just the element itself as in *Minimum Scalar Product*. Elements might be pairs of numbers where we have to compare their differences, products or quotients with each other. The sorting and exchange argument is not always the entire answer either, but often just provides us with an ordering of something used later in the problem.

Citations – citations

By Fredrik Hernqvist. Baltic Olympiad in Informatics 2018, practice session

Grace is going to read a certain scientific book. However, she is very careful about always reading the sources that books refer to, and also the sources of the sources, and so on. The librarians are constantly complaining about Grace's excessive borrowing of books, so she now wants to find an order in which to read the books that minimizes the total borrow time.

There are $N \leq 100\,000$ books that she eventually will need to read, the i'th takes k_i minutes to read and has a citation list with $f_i < N$ books. Before starting with book 1 Grace has already borrowed all N books.

When Grace reads a book she opens the book and reads the citation list, which takes a minute, then she reads all the books that are in the list in some order of her own choosing, thereafter she reads the actual book and returns it to the library, which takes k_i minutes.

Every book except book 1 will occur in **exactly one** citation list. Compute the minimum sum of the borrow times for all books, given that Grace reads the books in an optimal order.

Solution. The best approach is to reason about the citations of book 1. We already know for how long we must borrow book 1 – it is the sum of how long it takes to read all its citations recursively plus the time it takes to read the book itself plus 1 (for reading the citation list). This same applies to its citations as well, since the problem stipulates that once she opens a book she reads all its citations recursively, followed by the book itself.

When choosing the order to read the citations of book 1 in, we have two conflicting greedy ideas. On one hand, we want to read books with smaller total reading time earlier since each minute spent reading a book when there are B books waiting contributes B to the sum of the borrow times. On the other hand, we want to read books with many recursive citations earlier since each minute a book with C recursive citations is waiting contributes C to the sum of the borrow times. The difficulty in the problem lies in finding the balance between these two factors.

Note that these are the *only* factors in selecting the order in which the citations should be read. More formally, assume that the first book has k citations where the i'th takes b_i

total borrowing time to read (if we only read this book), t_i total reading time, and has c_i recursive citations. The total borrowing time of those books are then

$$b_1 + [t_1 \cdot c_2 + b_2] + [(t_1 + t_2) \cdot c_3 + b_3] + \cdots$$

Notice how the b_i terms are not affected by the ordering of the books, so we only care about minimizing the expression

$$[t_1 \cdot c_2] + [(t_1 + t_2) \cdot c_3] + [(t_1 + t_2 + t_3) \cdot c_4] + \cdots$$

Appealing to the exchange argument, swapping two adjacent books i and i_{i+1} improves this sum if and only if $t_i c_{i+1} - t_{i+1} c_i > 0$ (this is the difference between the two sums when the books are swapped). This is equivalent to $t_i c_{i+1} > t_{i+1} c_i$ or $\frac{t_i}{c_i} > \frac{t_{i+1}}{c_{i+1}}$, so we should sort the elements by this quotient in ascending order[3]. This also correspond with our initial intuition – the quotient is smaller the more citations and the lower total reading time a book has.

The same rule for ordering citations for book 1 of course applies to every other citation list too. What remains is to recursively compute the values b_i, t_i, c_i for each book and use the formula given to compute the total borrowing time. □

Problem 10.7.

Töflur	toflur
Swap Space	swapspace
Distributing Seats	distributingseats

10.4 Intervals

We move on two standard greedy problems, both about intervals: *scheduling* and *covering*. Variants are common in algorithmic problems, but the basic ideas are the same.

Scheduling

Scheduling is a class of problems dealing with constructing large subsets of non-overlapping intervals from some given set of intervals. The scheduling problem in its classical setting is the following.

Interval Scheduling – intervalscheduling

Given a set of half-open (being open on the right) intervals S, find a largest subset of non-overlapping intervals.

Solution. As is often the case with greedy constructions, we add intervals to the subset one at a time. After choosing an interval, we remove all the intervals it overlaps with and

[3]When implementing the sort, use a custom comparator where you compare the products instead, to avoid precision issues with doubles.

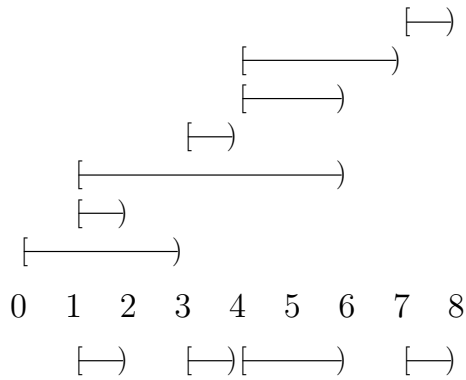


Figure 10.1: An instance of the scheduling problem, with the optimal solution at the bottom.

repeat the process until we run out of intervals. To find the interval that should be added, we now know to apply the extreme value principle. Some reasonable ideas for a locally optimal choice are for instance:

- a shortest interval,
- an interval overlapping as few other intervals as possible,
- an interval with the leftmost left endpoint, and
- an interval with the leftmost right endpoint.

In the above list, the first option – a shortest interval – might seem like it should be the least disruptive. It's not a stupid choice, but unfortunately it's not optimal due to the counterexample $[1, 3), [2, 4), [3, 5)$. It does produce a not too bad solution:

Exercise 10.8. Show that always choosing the shortest interval picks a subset at least half the size of the optimal solution.

Does this mean that the second choice is better, since it overlaps very few intervals? It's harder to disprove and is actually true for all optimal solutions smaller than 4 intervals. The idea for a counterexample is the same kind of overlapping that made the shortest interval sub-optimal, plus adding lots of intervals we would never want elsewhere to force the sub-optimal interval to be chosen. The third choice on the other hand is just bad – it could choose a single interval overlapping every other – but at least it's on the right track.

As it happens, the fourth choice is optimal. In the example instance in Figure 10.1, the strategy results in four intervals. First, the interval with the leftmost right endpoint is the interval $[1, 2)$. If we include this in the subset, the intervals $[0, 3)$ and $[1, 6)$ must be removed since they overlap $[1, 2)$. Then, the interval $[3, 4)$ would be the one with the leftmost right endpoint of the remaining intervals. This interval overlaps no other interval,

so it should obviously be included. Next, we would choose $[4, 6)$ (overlapping with $[4, 7)$), and finally $[7, 8)$. Thus, the answer would be $\{[1, 2), [3, 4), [4, 6), [7, 8)\}$.

The optimality proof is a simple exchange argument. Assume that it's not optimal to choose the interval $[l, r)$ with smallest r. Let $[l', r')$ be the interval with the smallest r' of those that were chosen in the optimal set (so that $r' > r$). Every other interval in the solution then has a left endpoint that is to the right of r'. These intervals don't overlap with $[l, r)$ either, so we can exchange $[l', r')$ with $[l, r)$ to obtain a valid solution of the same size as well.

The implementation strategy suggested earlier – finding the right interval and removing all the overlapping ones from S – is $\Theta(N^2)$ in the worst case (N is the number of intervals). If we instead walk through the intervals sorted by their right endpoints we can "lazily" remove them by only keeping track of the last added interval and throw away intervals that don't lie completely to the right of it. This approach is $\Theta(N \log N)$:

```
1: procedure Interval Scheduling(S)
2:     sort S by right endpoint
3:     rightmost ← −∞
4:     for each interval [l, r) in S do
5:         if l ≥ rightmost then
6:             add [l, r) to the answer
7:             rightmost ← r
```

□

Exercise 10.9. A variation is to instead choose K **disjoint** subsets of non-overlapping intervals, maximizing the total number of intervals chosen. The solution is similar – we should always include the interval with the leftmost right endpoint in one of the subsets if possible. The question is, which subset? Intuitively, we should choose a subset the new interval causes as little "damage" as possible. This turns out to be the subset with the rightmost right endpoint (that the interval can be placed in). **Prove that this strategy is optimal.**

Exercise 10.10. In neither of these two scheduling variants have we given any thoughts on how to break ties for intervals with the same right endpoint when sorting them. It turns out that the order does not matter.

1. Prove this for the normal scheduling problem.
2. Prove this for the scheduling variant with K subsets.

Exercise 10.11. To find the smallest K such that all the intervals can be divided into K disjoint non-overlapping subsets, we could binary search over K (see Section 12.3) and use the previous algorithm. A simpler solution is possible: do the same as in the previous solution, but add a new subset whenever you can't place the next interval in one of the existing subsets. The answer is then the number of intervals used. **Prove that this strategy is optimal.**

Problem 10.12.

Entertainment Box	entertainmentbox
Disastrous Downtime	downtime

Interval Covering

Interval covering is the cousin of scheduling where we instead want to fill up an interval completely.

Interval Covering

Given a set of half-open (being open on the right) intervals S, and an interval $[L, R)$, find the smallest subset of intervals such that their union includes the entire interval $[L, R)$, or determine that this is impossible.

Solution. Some interval must cover the point L. Of all the intervals we could choose that covers L there is only one natural greedy choice – the one that extends as far as possible to the right. A formal proof that this is optimal uses an exchange argument similar to the one we did for scheduling.

To implement this method faster than $\Theta(N^2)$ we use a strategy similar to how we solved the scheduling problem. First, sort the intervals by increasing leftmost endpoint. We can then find all intervals overlapping L by iterating through the list until we find a leftmost endpoint to the right of L. If the rightmost endpoint among these is X, that's the interval we choose. The problem is then again an interval covering problem, but for the interval $[X, R)$. The remaining intervals are already sorted, so we can just repeat the same procedure (but for intervals overlapping the point X) until the whole interval is covered.

In total, this too is $\Theta(N \log N)$ with a slightly more complicated implementation:

```
1:  procedure IntervalCover(S, L, R)
2:      sort S by left endpoint
3:      i ← 0
4:      while L < R do
5:          bestI ← −1
6:          bestR ← −∞
7:          while i ≠ S.size and S[i].left ≤ L do
8:              if S[i].right > bestR then
9:                  bestR ← S[i].right
10:                 bestI ← i
11:             i ← i + 1
12:         if bestI = −1 then
13:             return impossible
14:         add S[bestI] to the cover
15:         L ← bestR
```

□

Exercise 10.13. What changes must be made to the algorithm above if the intervals in S are closed intervals and the interval to cover is instead $[L, R]$?

Problem 10.14.

Interval Cover	intervalcover
Watering Grass	grass
Keyboards in Concert	keyboardconcert

10.5 Constructions

Constructive problems ask us to produce some kind of configuration that satisfy constraints given in the problem. They are often mathematical objects such as colourings of a grid, graphs or sequences. There are as many ways to solve construction problems as there are problem solving techniques, but greedy (and as we see in Chapter 12, recursive) constructions deserve special attention since they are quite common.

Bonbons – bonbons

SANDVIK Challenge 2020

Ylva loves bonbons, probably more than anything else on this planet. She loves them so much that she made a large plate of $R \cdot C$ ($R, C \leq 1000$ with R, C even) bonbons for her fikaraster[a]. To serve them she will place them on a wooden tray with R rows of C bonbons each.

The bonbons are of three different kinds, of which she has a, b and c bonbons respectively ($a + b + c = R \cdot C$). When filling the tray with bonbons, no bonbons of the same kind may be immediately horizontally or vertically adjacent.

Can you help Ylva place her bonbons on the tray, or determine that it is impossible?

[a] "Fikarast" is a Swedish word, meaning to take a break from work while enjoying coffee and pastries together with your colleagues.

Solution. *Bonbons* is a tricky problem where contestants came up with all kinds of incorrect greedy solutions during the contest it was part of. It is very easy to generate ideas but surprisingly hard to both find counterexamples against the bad ones and prove the correctness of the right ones.

Whenever constructions can be impossible to create, we first need to find out under what conditions it's impossible. Most of the time this is as simple as trying out a greedy construction and seeing if it fails. Only when we need to understand the impossibility constraints to come up with a greedy algorithm are we forced to figure this out beforehand. In this case, an obvious necessary condition is that there can't be more than $\frac{RC}{2}$ bonbons of any kind. There would then have to be more than $\frac{C}{2}$ bonbons on some row, but then two bonbons would be next to each other.

The first step towards a solution is a natural decomposition of the grid squares into those with an even or odd coordinate sum respectively, forming two chequered patterns. Bonbons placed in one of those patterns do not affect each other, so this gives us a lot of

freedom in deciding what bonbons to place in a single pattern. When filling the first of these patterns, it is probably a good idea to place as many bonbons of the most common kind close to each other. This leaves many "gaps" in the other pattern in which any bonbon of the other two types could be placed. Following this, we should probably try to place as many bonbons of the second most common kind and putting the leftovers into the beginning of the second pattern. It should be possible to completely use up all those bonbons, since the most common kind should have left more gaps than there are bonbons of the second kind. If we manage to do this, all the remaining free spots must be in the second pattern, since the first two together must fill the entire first pattern. This means that the last kind can be freely placed, finishing up the construction.

While this all sounds good and intuitive, it is also slightly incorrect with a trap many contestants fell into. A counterexample is given by $R = 2$, $C = 6$, $a = 5$, $b = 4$, $c = 3$. This greedy choice fails after placing the fourth bonbon of type B. It can still be placed, just not in that particular place:

```
ABABAB          ABABA.
.A.A.B          BA.A.B
```

Figure 10.2: The incorrect greedy construction, and a possible alternative.

Is this the end of this greedy approach? Luckily not – it turns out there is always *somewhere* to place the remaining B bonbons once they overflow into the second pattern, just not always in that particular order. The proof is a bit of math. First, assume that $a \geq b \geq c$. Initially, the first pattern gets $\frac{RC}{2} - a$ B's. This means that at most $2(\frac{RC}{2} - a) = RC - 2a$ squares in the second pattern are adjacent to a B, leaving $\frac{RC}{2} - (RC - 2a)$ spots left. For this to be enough, we must have that

$$b - (\frac{RC}{2} - a) \leq \frac{RC}{2} - (RC - 2a).$$

We simplify a few times:

$$b \leq \frac{RC}{2} - (RC - 2a) + (\frac{RC}{2} - a) = RC - (RC - 2a) - a = a$$

which is true by assumption, so the construction works as long as try to use all available squares in the second pattern to add the extra B's. □

In the problem, we used some kind of principle of minimal disruption: the A bonbons were initially placed in such a way that they disrupted each other – and the placement options for other bonbons – as little as possible to produce as much freedom as possible for future placements. This is a very common idea in greedy construction problems (which Scheduling actually is an instance of too). It also guides us towards a greedy strategy in the next problem, which requires quite some effort to prove correct.

Box of Mirrors – boxofmirrors

By Mārtiņš Opmanis. Baltic Olympiad in Informatics 2001

On an $N \times M$ grid, each square is either empty or contains a mirror placed diagonally from the top-right corner of the square to the bottom-left corner. From the four sides of the grid, a laser beam is fired right into the center of each row and column for a total of $2(N + M)$ beams. When a beam hits a mirror, it is reflected and changes direction by 90 degrees.

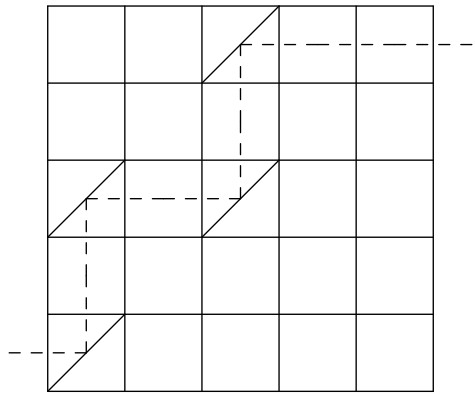


Figure 10.3: An example of how a beam travels when hitting mirrors.

For each beam, you record from what square it exits the grid. Given this information, find a possible placement of mirrors in the grid.

Solution. Knowing where to start in this problem might be the hardest part. There is no obvious way to place mirrors greedily, so we have to make one up. The right idea is to fire all the beams from the left and from below in some smart order, and place mirrors greedily in a way that makes the beans exit in the right place. Note that beams fired from these two sides travel only upwards and rightwards, so they all exit through the other two sides.

The right order turns out to be processing the rows top to bottom, followed by the columns left-to-right. This makes some sense. The first beam we place can only utilize a single mirror, and the position it gets is forced entirely forced. On the other hand, if we start with the beam on the bottom-most row, we have a lot of freedom in where to place mirrors. It then becomes hard to know where they should be placed in order to minimally disrupt the mirrors we must place for other beams.

Once we trace the beam along the second row, things are not as straightforward. If the beam needs to be routed upwards to reach its target, there are a lot of ways to do so. We can look for guidance in the extreme value principle. Some possible choices on where to route the beam upwards would be for example as early as possible or as late as possible. Both of them are terrible. It might be the case that they result in placing a mirror in the first or last column. If the beams on those columns are supposed to pass straight through

the box to the other side, we have just obstructed that goal.

Can we avoid this problem? Yes – we can, without disrupting any of those column beams, try to redirect the beam upwards *in those columns that already have a mirror*. If there are several such choices, we should pick the earliest one. The earlier we start forcing the beam down, the longer we are able to, which gives the beam more freedom on what row to exit on. Note, of course, that we only add these mirrors if a row is supposed to either *exit at a higher row than it's current position*, or if it's supposed to exit vertically anywhere.

Aside from these mirrors, for each vertically exiting beam we also need to add a final mirror forcing it upwards when it arrives horizontally in its target column. Highly surprisingly, these are actually all the mirrors that need to be added. For an example of the algorithm in action, see Figure 10.4.

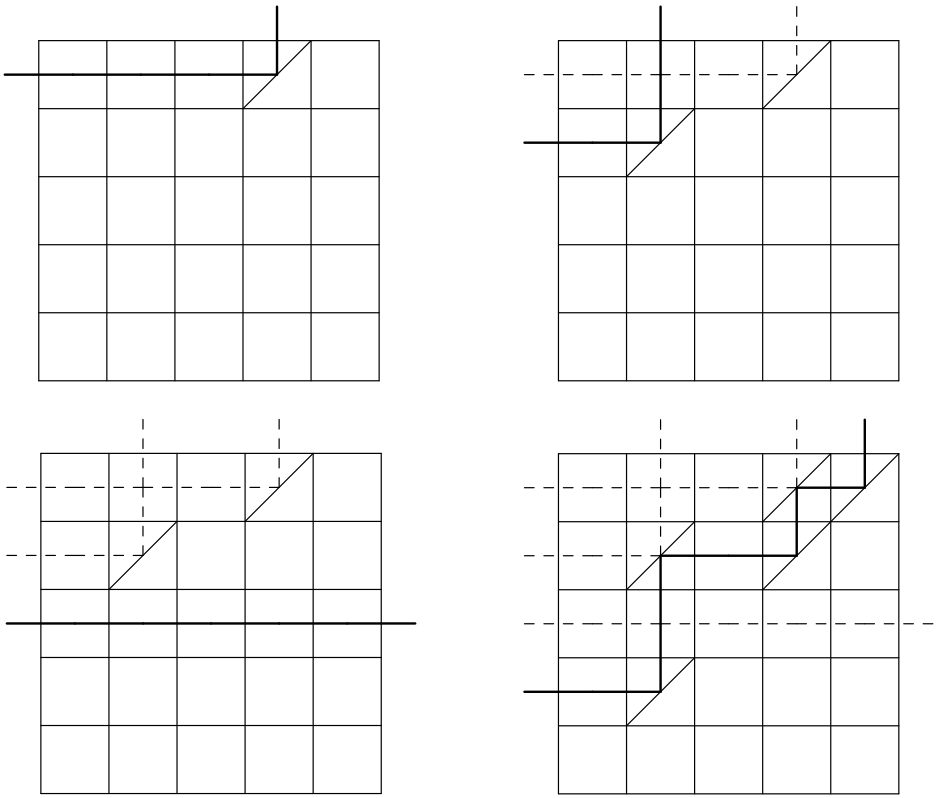


Figure 10.4: An example of how the mirror placing algorithm works.

Exercise 10.15. Prove that when a given beam is being traced through the grid, the mirrors

it causes to be placed never disrupt the path of one of the earlier traced beams.

The statement that we need no more mirrors raises a few big questions.

- When a beam exiting vertically reaches its target column, can there already be a mirror in the way on its path upwards?
- Can a beam accidentally exit through the top because there were no mirror to stop it?
- Don't we need any mirrors to make beams exiting horizontally leave at the right row?

The first question is straightforward to answer. Assume that we are tracing a beam that wants to exit upwards somewhere, but there is a mirror in the way. We only place mirrors in a column for two reasons: either there is already a mirror there, or a beam passed through the column horizontally but needed to exit in that column. Thus, a beam can only be blocked in the column if another beam already wanted to exit in the column – a contradiction.

We can now prove that all beams that are to exit through the top does so correctly. By construction, all of those beams exit upwards – we always add a mirror forcing it upwards when the beam passes through the correct column, and we now know the beam won't be blocked. A beam can also never exit in the wrong column. Assume to the contrary that there are beams leaving through the wrong column, and consider the first one, leaving through column c. The beam actually meant to leave through c must have exited through a column $c' > c$ instead, but then it passed column c – and by the previous question, would have been able to exit through the column. This answers the second question – no two beams can exit through the same hole, so no extra beam can accidentally exit vertically.

The last question is the trickiest one. We now know that all horizontally exiting beams leave through a hole to the right, but need to prove that they can not exit through the wrong row. It is a good exercise to prove that this can't happen. □

Exercise 10.16. Prove that all beams exit through the correct row after the above algorithm has placed all mirrors.

Exercise 10.17. We said that rows must be processed top-to-bottom followed by columns left-to-right. It might seem more natural to process columns right-to-left, by the same reasoning for why starting with rows bottom-to-top was bad. Is the column order necessary?

1. Find an example where processing columns right-to-left fails.
2. What step in the proof required that columns were processed in this order?

Interestingly enough, the official solution booklet from the contest in which the problem appeared did not even mention this last possibility, i.e. that horizontal beams exit

through the incorrect rows. Perhaps they thought it was obvious, but we think it's the most difficult part of the correctness proof.

Problem 10.18.

Espresso Bucks	espressobucks
Hard Drive	harddrive

Additional Exercises

Problem 10.19.

Intergalactic Bidding	intergalacticbidding
Left and Right	leftandright
Marathon	marathon
Saving the Universe	savinguniverse
Messages from Outer Space	messages
Hyacinth	hyacinth
Classrooms	classrooms
Inflation	inflation
Matchsticks	matchsticks
Poplava	poplava
Cu Chi Tunnels	cuchitunnels
Cake	cake
Canvas Line	canvasline
Boiling Vegetables	vegetables
Wireless is the New Fiber	newfiber

Notes

Determining whether coins of denominations D can even be used to construct an amount T is an NP-complete problem in the general case [31]. It possible to determine what cases can be solved using the greedy algorithm described in polynomial time though [9]. Such a set of denominations is called a *canonical coin system*.

Introduction to Algorithms [11] also treats the scheduling problem in its chapter in greedy algorithms. It brings up the connection between greedy problems and a concept known as *matroids*, which is well worth studying.

In this chapter, we only studied greedy algorithms that give an optimal solution. For many NP-complete problems, there are greedy algorithm that gives good approximations, producing solutions always within a some small factor of the optimal one.

Chapter 11

Dynamic Programming

It is time to tie up some loose ends when it comes to search and optimization problems. So far, we have seen problems where we could formulate recursions (Chapter 7), where we needed to exhaustively try all recursive choices to solve them (Chapter 9) and just now problems where we just guessed (and hopefully proved) what choice was correct (Chapter 10).

Sometimes, we mentioned in passing that recursions could be solved in linear time. In other problems we had to resort to the exponential complexity of backtracking. This chapter explains when the specific approach we have used to avoid backtracking (without resorting to greedy algorithms) work. It's one of the most important paradigms in algorithmic problem solving – *dynamic programming*.

We begin with a familiar example, the change-making problem with a different set of denominations, and follow up with a *lot* of standard problems and techniques.

11.1 Making Change Revisited

In the previous chapter, we solved the change-making problem with denominations 1, 2, and 5 greedily. The best choice of coin for a given T was always the greatest one that didn't exceed T. After getting your hopes up, Exercise 10.2 asked you to prove that the case with coins worth 1, 6 and 7 could not be solved in the same greedy fashion.

So, how is this variant solved? Well, you already know how to solve the 1, 2, 5 case non-greedily in linear time. Formulate the recursion

$$\text{Change}(T) = 1 + \min \begin{cases} \text{Change}(T-1) & \text{if } T \geq 1 \\ \text{Change}(T-6) & \text{if } T \geq 6 \\ \text{Change}(T-7) & \text{if } T \geq 7 \end{cases} \tag{11.1}$$

with base case $\text{Change}(0) = 0$ and solve it iteratively in the way that we showed in Chapter 7. The code was pretty simple, storing the answers in an array as we go along:

```
1: procedure ChangeMaking(integer T)
2:     answers ← new array of size T + 1
3:     ▷ base case
4:     answers[0] ← 0
```

```
5:    for i = 1 → T do
6:        answers[i] ← 1 + answers[i − 1]
7:        if i ≥ 6 then
8:            answers[i] ← min(answers[i], 1 + answers[i − 6])
9:        if i ≥ 7 then
10:           answers[i] ← min(answers[i], 1 + answers[i − 7])
11:   return answers[T]
```

Problem 11.1.

The Gourmet gourmeten *(all subtasks)*

What makes the iterative computation able to speed up this particular recursion from exponential to linear, but we can't do the same for e.g. the max clique problem? The theoretical answer is that recursions allow a space-time trade-off where we can reduce the time complexity to *the number of subproblems* we recursively solve, multiplied by the time it takes to solve a single subproblem. For change-making, we need to solve up to $T + 1$ subproblems in total, and each subproblem is a constant amount of work. In the backtracking problems, there were simply an exponential number of subproblems. For max clique, we had to go through all vertex subsets to see if they were cliques. To count the number of subproblems, look for how many different values the parameters to the recursive function can take. The space trade-off is that we must save the answers for each subproblem.

This property distinguishing the recursions that can be sped up using the iterative method from those that can't is in the theory called ***overlapping subproblems***. It is a fancy term that means "during backtracking, you call your recursive procedure with the same parameters many times". Generating subsets does not revisit the same subproblem, so the technique does not help. Change-making revisits the same subproblem an exponential number of times, so the technique gives us an exponential speedup.

Exercise 11.2. For the following recursive functions, determine whether their subproblems overlap (i.e. will be called multiple times with the same arguments).

```
1: procedure Fibonacci(integer i)
2:     if i < 2 then
3:         return i
4:     return Fibonacci(i − 1) + Fibonacci(i − 2)
```

```
2: procedure DFS(vertex v, array seen)
2:     seen[v] ← true
3:     for all neighbors u to v do
4:         if not seen[u] then
5:             DFS(u)
```

```
3: procedure Power(real b, integer e)
```

```
2:    if e = 1 then
3:        return b
4:    returnPower(b, ⌊e/2⌋) · Power(b, ⌈e/2⌉)
```

This is essentially all that ***dynamic programming***, or *DP* is. You take a problem that admits a self-recursive solution (i.e. it can be solved by reducing it to smaller instances of itself) and compute each subproblem exactly once. If a subproblem is indirectly used many times during a recursion, you win time.

As an intuitive mental model, this explanation is not very good. It doesn't tell you when you should go for a recursive solution because you can apply dynamic programming, or what recursion you should choose to make it work. The remainder of the chapter describes a range of ways in which you can think about recursion to find DP solutions.

11.2 Paths in a DAG

The standard problem for dynamic programming problems is about paths in a directed, acyclic graphlongest directed acyclic graph, a DAG. In Section 8.6 we learned how to topologically order the vertices of a DAG, so that every edge points from a later vertex to an earlier vertex. While this problem is important in its own right, it's also valuable to study in order to understand DP. As we soon see, the topological ordering carries importance for DP.

Longest Path in a DAG – longestpathinadag

Given a directed, acyclic graph, find a longest path in it.

Solution. This problem is best approached with the framework for recursive choices that we developed in Chapter 7. Consider the longest path $p_1 \to p_2 \to \cdots \to p_l$ in the graph. The path must start at a vertex, in this case p_1. Now channel your inner Tasha the kitty (page 99) and ask: what is the first edge on this longest path? This can be any one of those that point out from p_1. We should pick the edge $p_1 \to v$ such that the longest path starting at v is as long as possible. Any path that starts at v can be extended with one more edge $p_1 \to v$, so the longest path starting at v plus this edge is the longest path starting at p_1. This gives us the recursion we are after:

```
1: procedure LongestPath(vertex v)
2:     length ← 0
3:     for each out-edge v → u do
4:         length = max(length, LongestPath(u) + 1)
5:     return length
```

The length of the overall longest path is then $\max_v \text{LongestPath}(v)$.

To avoid computing LongestPath(v) for the same v an exponential number of times, we use the iteration trick of computing it for each v in…increasing order? Hmm, this recursion is clearly more complex than the previous ones. In the recursions seen so far we could find a simple ordering of the parameters to ensure that the recursive subproblems had already been computed when we reached a subproblem.

We somehow need to find an ordering of the vertices such that whenever there is an edge $v \to u$, the answer for u must have been before that of v. A-ha! That's exactly what a topological ordering guarantees us. We thus need to first find this ordering before we can iteratively compute the recursion:

```
1: procedure IterativeLongestPath(vertices V)
2:     answers ← new array of size V.size
3:     order ← TopologicalOrder(V)
4:     for v in order do
5:         for each out-edge v → u do
6:             answers[v] = max(answers[v], answers[u] + 1)
7:     return the maximum of length
```

The above pseudo code is $\Theta(V + E)$ for V vertices and E edges – the innermost loop does constant-time work and loops once over every single edge.

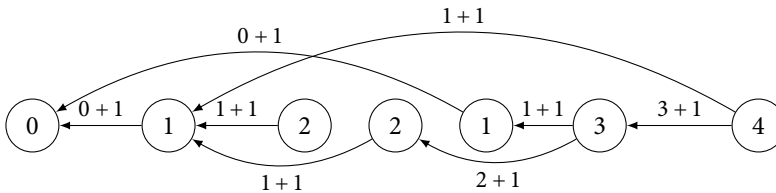


Figure 11.1: The length of the longest path starting at each vertex and how it is computed.

We have cheated a bit so far. The problem asked for *the longest path*, but we have so far only computed its length. The reconstruction works just like the BFS reconstruction for shortest path does. For each v, make sure to also store the edge $v \to u$ where u had the longest path in another vector *next*. Given this, the reconstruction is easy:

```
1: procedure LongestPathReconstruction(answers, next)
2:     i ← the index where answers is greatest
3:     while answers[i] ≠ 0 do
4:         add i → next[i] to the path
5:         i = next[i]
```

□

Exercise 11.3. The path can be reconstructed using only *answers* and the input graph. How?

Memoization and Memory

Do we really need to find a topological ordering to recurse on the paths of a DAG? Depending on how experienced of a programmer you are, the answer *no* might be surprising. The recursive function can be adapted slightly to avoid the exponential blow-up using ***memoization*** or ***top-down*** dynamic programming (in contrast to ***bottom-up***, which the iterative computation is called). The idea is that since the recursive function returns the same value every time it's called, we can store the return value after the first call and return it immediately for subsequent call. In software engineering, storing the results of computations for later reuse is called *caching*. When applied directly to the return value of a function in this manner it's called memoization. Memoizing the function for the longest path in a DAG looks like this:

```
1: memo ← new array of size V filled with −1
2: procedure LongestPath(v)
3:     if memo[v] ≠ −1 then
4:         return memo[v]
5:     length ← 0
6:     for each out-edge v → u do
7:         length = max(length, LongestPath(u) + 1)
8:     return memo[v] = length
```

The return values for the function is stored in *memo*, which is initially filled with a sentinel value −1 that allows us to distinguish between values for which the function has been computed and those not yet called. When the answer is always non-negative, −1 is the standard choice.

The benefit of memoization is twofold. First, we don't have to find the topological ordering of the function calls in the recursion. As we just saw, this ordering can need an explicit topological sorting to construct. Secondly, with memoization the recursion only calls the subproblems that can affect the answer we are interested in rather than all possible ones. In some problems, we can win constant factors due to this. In extreme cases an entire parameter can turn out to be unnecessary, such as in the Ferry Loading problem in Section 11.3. When that happens, you normally need to realize it before coding though, in order to reduce the number of states to something that fits in memory.

Problems on DAG paths are not uncommon in contests, and thanks to memoization become easy to code. Typically, the solution – in fact, the solutions for most memoized DP problems – looks very similar to the one we just showed you.

Problem 11.4.

BAAS	baas
Safe Passage	safepassage

Memoization is not without downsides. There are a lot of function calls in recursive

solutions, and they carry some overhead. This problem is not as bad in C++ as in many other languages, but it is still noticeable. When the number of states in your DP solution is running a bit high, you might want to consider coding it iteratively. The performance gain from avoiding recursion is normally greater than the benefit of not computing a few unnecessary subproblems.

In top-down DP, the memory usage is for the most part clear and unavoidable. If a DP has N states, the top-down solution uses $\Omega(N)$ memory to store all the states. For a bottom-up solution, the situation is quite different. If we choose the order in which the subproblems are computed well, we seldom need to store the answers to all of them all the time. Consider e.g. the change-making problem again, which had the following recursion:

$$\text{Change}(T) = 1 + \min \begin{cases} \text{Change}(T-1) & \text{if } T \geq 1 \\ \text{Change}(T-6) & \text{if } T \geq 6 \\ \text{Change}(T-7) & \text{if } T \geq 7 \end{cases} \tag{11.2}$$

If we compute $\text{Change}(T)$ for values of T in the order $0, 1, 2, 3, \ldots$, once the answer for $\text{Change}(k)$ has been computed, the answers for $k-7, k-8, \ldots$ are never used again. During the entire process, only the answers to the 7 last subproblems are needed. This $\Theta(1)$ memory usage is pretty neat compared to the $\Theta(K)$ usage needed to compute $\text{Change}(K)$ otherwise. In the worst case, such as when recursion is over a general directed graph, this doesn't help.

Competitive Tip

Generally, memory limits are very generous nowadays, somewhat diminishing the art of optimizing memory in DP solutions. It can still be a good exercise to think about improving the memory complexity of the solutions we look at, for the few cases where these limits are still relevant.

11.3 Standard Techniques

By now, you might have realized that a lot of what we discussed in these 3 last chapters have very much in common with the mental model of recursion from Chapter 7. The reason for this is very simple – that chapter was written to be the common ground for the recursive search techniques to make them easier to talk about. Almost all the recursive problems described in that chapter are susceptible to dynamic programming to get a polynomial-time solution. In this section we look at some more complex recursions, but the basic principles remain the same.

More Choices

We start with a variation of the classical *stock trading* DP problem.

Short Sell – shortsell

LiU Coding Challenge 2018

Simone is trading the cryptocurrency CryptoKattis, used to improve your Kattis ranklist score without solving problems. She is very perceptive of online judge trends and noticed that many coders are switching to the new online judge Doggo. She thinks that Kattis is falling out of fashion which will cause CryptoKattis to decline in value. Careful market research allows her to estimate the prices of a CryptoKattis in dollars during the next $N \leq 100\,000$ days denoted $P_1, \ldots, P_N$. She intends to use this data to perform a *short sell*. One day she will borrow 100 CryptoKattis from a bank and sell it for dollars. At a later day, she will purchase 100 CryptoKattis and repay her loan to the bank. Every day between and including these two days, she must pay K dollars in interest. What's the maximum profit she can make by choosing these two days optimally?

Solution. On a given day, Simone has three choices: lend the CryptoKattis and sell them, do nothing, or buy CryptoKattis to repay her loan. The correct question for a recursive solution is: if she only performs the first two actions, what is the most cash $C(i)$ she can have on hand at the end of the i'th day? The profit she can make by selling on day i would then be $C(i) - 100P_i$, and to solve the problem we'd take the maximum over all i. There are three cases when computing $C(i)$. If she has not bought CryptoKattis yet, the answer is 0. If she has already bought she must pay interest today, so she loses K money from the previous day's maximum cash, i.e. $C(i-1) - K$. This can also be the day she does the short sell, giving her $100P_i - K$ money. Together, these cases gives us the recursion

$$\mathrm{C}(i) = \max \begin{cases} 0 \\ 100P_i - K \\ C(i-1) - K \end{cases} \tag{11.3}$$

DP solutions for recursions in only a single parameter are predominantly written iteratively. This DP only depends on the answer for $i-1$, so we don't need an array to store past results.

```
1: procedure ShortSell(N, P, K)
2:     C ← 0
3:     profit ← 0
4:     for each price p in P do
5:         C ← max(0, 100p − K, C − K)
6:         profit ← max(profit, C − 100p)
7:     return profit
```

□

Exercise 11.5. Adapt the solution to Short Sell to the case where Simone is allowed to perform several short sells, as long as they don't overlap.

Problem 11.6.

The Stock Market borsen *(all subtasks)*

Radio Commercials	commercials	
Going to School	skolvagen	*(all subtasks)*

The following problem was one of the examples we gave on incorrect greedy algorithms. We return to it now, as perhaps our first "real" dynamic programming problem that recurses on more than one parameter.

Bookshelves – bokhyllor

By Arash Rouhani. Swedish Olympiad in Informatics 2012, School Qualifiers.

You are buying bookshelves to fit all your books. Books come in three sizes; a small book has width 1, a medium book width 2 and a large book width 3. Each bookshelf has the same width $L \leq 20$. Given the amount of books you have (at most 20 of each size), compute the number of bookshelves needed if the books are placed optimally on the shelves.

Solution. The first step in solving dynamic programming problems is to reformulate it in the recursive "sequence of choices" form we are used to. In this problem, no choices are given to us – we must define them ourselves.

How should you think when formulating recursive choices that are supposed to work well with dynamic programming? The core idea is that after you recursively try some number of choices, you arrive at a situation where the important thing is **not what choices you made**, only their consequences. For example, in the change-making problem, when solving a recursive subproblem T' it's completely irrelevant which coins we picked to get there. The optimal solution for the remaining T' money is the same no matter if we arrived there after recursively using 1000 coins of value 5 or 5000 coins of value 1. Not all problems can be formulated in this way. For example, say that you try to find the longest path starting at some vertex in a general graph using recursion. Is it enough to formulate a recursive function LongestPath(v) that recursively tries what vertex next to visit? No! The choices that the recursive function can make depends on previous choices, so this does not work.

In this problem, we propose the following construction method. Fill each bookshelf that we buy one at a time. At any given time, we have four choices. We might either place a book on the current shelf (one choice per book type) or we can buy a new bookshelf. During this construction, we need to keep track of only four things: how many books we have yet to place of each of the three types and how much space we have left on the current shelf. This results in the following recursion:

```
1: procedure Bookshelves(s, m, l, widthLeft)
2:     if s = m = l = 0 then
3:         return 0                          ▷ base case – we have no more books to place
4:     answer ← ∞
5:     if s > 0 and widthLeft ≥ 1 then
6:         answer ← min(answer, Bookshelves(s − 1, m, l, widthLeft − 1))
```

```
7:  if m > 0 and widthLeft ≥ 2 then
8:      answer ← min(answer, Bookshelves(s, m − 1, l, widthLeft − 2))
9:  if l > 0 and widthLeft ≥ 3 then
10:     answer ← min(answer, Bookshelves(s, m, l − 1, widthLeft − 3))
11: if widthLeft < L then
12:     answer ← min(answer, Bookshelves(s, m, l, L) + 1)
13: return answer
```

The function is invoked with Bookshelves$(s, m, l, 0)$ since we in the beginning need to buy a first shelf. We have not added the memoization here, and will typically not do so in this chapter to avoid cluttering – coding that is up to you.

Exercise 11.7. Why is the check for *widthLeft* $< L$ needed on line 11?

To find the time complexity of the solution, we compute the number of valid parameter combinations, called the *states* of the recursion. The values s, m, l only decrease and are at least 0, while *widthLeft* is between 0 and L, so there are $O(s \cdot m \cdot l \cdot L)$ states. The function itself takes constant time, so the number of states is also the total time complexity. □

Problem 11.8.

Dance Dance Revolution	dansmatta
Nikola	nikola

Note that the recursive construction method we used can construct *every* possible placement of books on the shelves! When we impose a construction method we must ensure that it covers the optimal possibility too! In difficult problems it's sometimes the choice of recursive construction that makes the solution fast enough, for example by not testing certain states that can never be optimal. Different ways of looking at a problem can give you a variety of recursions and parameters representing your subproblems. This next example demonstrates the importance of choosing what parameters to include with care.

Ferry Loading – lastafarjan

By Oskar Werkelin Ahlin. Swedish Olympiad in Informatics 2013, Online Qualifiers.

A ferry is to be loaded with $N \leq 200$ cars of different lengths waiting to board in a long line. The ferry consists of four lanes, each of the same length $L \leq 60$. When the next car in the line enters the ferry, it picks one of the lanes and parks behind the last car in that line. There must be a safety margin of 1 meter between any two parked cars.

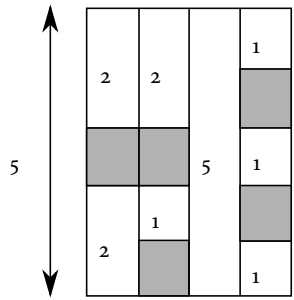


Figure 11.2: An optimal placement on a ferry of length 5 meters of cars with lengths 2, 1, 2, 5, 1, 1, 2, 1, 1, 2 meters. Only the first 8 cars could fit on the ferry.

How many cars can park on the ferry if they choose the lanes optimally?

Solution. As a simplification, increase the initial length of the ferry by 1 to accommodate an imaginary safety margin for the last car in a lane in case it is completely filled, and increment the lengths of each car by 1 so that we don't have to care about the safety margin at all.

The problem is already given in the sequence of choices form that we know how to translate to a recursion. We have an ordered list of cars and each one has 4 choices – one for each lane it could go into. If a car of length m chooses a lane, the remaining length of the chosen lane is reduced by this amount. After the first c cars have parked on the ferry, the only thing that has changed are the remaining lengths of the ferry lanes.

This suggests a DP solution with nL^4 states, each state representing the number of cars so far placed and the lengths of the four lanes:

```
1:  procedure Ferry(car, lanes)
2:      if car = N then
3:          return 0                          ▷ base case – we have no more cars to place
4:      answer ← 0
5:      for i from 0 to 3 do
6:          if lanes[i] ≥ carLengths[car] then
7:              newLanes ← lanes
8:              subtracting carLengths[car] from newLanes[i]
9:              answer ← max(answer, 1 + Ferry(car + 1, newLanes))
10:     return answer
```

Unfortunately, memoizing this procedure would not be sufficient. The number of states is $200 \cdot 60^4 \approx 2.6 \cdot 10^9$, which requires gigabytes of memory.

To improve the solution, we think about what DP actually is. Dynamic programming is all about taking a list of choices we made and compressing it to only the information that can affect future choices – like ignoring what lane each car went into, instead only keeping

track of what cars remain and how much space is left. This removes information that is redundant for the recursion. Our suggested solution still has some lingering redundancy though.

In Figure 11.2 from the problem statement, we have an example assignment of the cars $2, 1, 2, 5, 1, 1, 2, 1$. These must use a total of $3 + 2 + 3 + 6 + 2 + 2 + 3 + 2 = 23$ meters of space on the ferry. Let $U(c)$ be the total length of the first c cars. This function is invertible – two different c always give different $U(c)$. Let u_1, u_2, u_3, u_4 be the total length of how all the cars that have parked in each lane so far, so that $U(c) = u_1 + u_2 + u_3 + u_4$. The four terms on the right are parameters in our memoization together with c. The left one isn't, but it is uniquely determined from c, which is a parameter.

This means that for some fixed values of *lanes*, there's only a single possible value that *car* can have. This means that we don't need to include it as a parameter *in the memoization* – it's still fine to have in the recursion, since we use it. This simplification leaves us with $60^4 \approx 13\,000\,000$ states, well within reason. □

Problem 11.9.
Buying Coke coke

Interval DP

A common DP category is recursing over intervals of a sequence. Within an interval, we often want to perform some action on a element within the interval and recursively solve a problem on the two subintervals we get after splitting the interval at that element. First, we show a problem that contains most of the elements an interval DP problem can have.

Zuma – zuma

By Goran Žužić. Croatian Olympiad in Informatics 2009/2010, round 5

One day Mirko stumbled upon a sequence of $N \leq 100$ colored marbles. He noticed that if he touches K ($K \leq 5$) or more consecutive marbles of the same color he could wish them to magically vanish, although he doesn't have to do that immediately. When marbles vanish, the marbles to the left and right of them will move towards each other and close up the hole formed. Note that Mirko needs to touch marbles to make them vanish – there are no chain reactions.

Fortunately, Mirko brought an inexhaustible supply of marbles from home, so he can insert a marble of any color anywhere in the sequence (even at the beginning or end). Find the smallest number of marbles he must insert into the sequence to make all of the marbles vanish.

Solution. This task is mainly an exercise in finding the correct subproblem and method of constructing a solution. The key idea is to study marble 1 in the sequence and ask what happens to it. We could insert some marbles of the same color right at the start and remove it. If that's not the optimal solution, there must be another marble of the same color later on in the sequence that, after removing all the intermediate marbles, is removed in the same group as the marble 1.

If that marble is the i'th one, we must clear the entire interval of marbles $[2, i-1]$ before 1 and i can touch (as we assumed should happen). Solving the interval $[2, i-1]$ is *independent* of what we do in the rest of the sequence. We don't know what i is the correct one to test, so we loop over all choices and take the best one. This is how most interval DP solutions work: within an interval, split it into two by looping over all possible "breaking points".

After deciding to merge with the i'th marble, we still have to solve the interval $[i, N]$, now with that extra marble 1. Furthermore, there might be some marble $j > i$ that is *also* supposed to be part of the group used to vanish marble 1. Then we must clear the interval $[i+1, j-1]$, and solve the remainder of the interval $[j, N]$, where we now have **two** extra marbles of the same color as j.

Note that there are only three things that varies in our process: the two endpoints of the interval and the number of marbles we accumulate immediately to the left of the interval. It seems like the correct subproblem is: "how many marbles $M(l, r, s)$ must be added to clear the interval $[l, r]$ if we have s marbles of the same color as the l'th immediately to the left of it?" The first step of the recursion is then the expression $\min_{l<i<r} M(l+1, i-1, 0) + M(i, r, s+1)$ where the i'th and l'th marbles must have matching color.

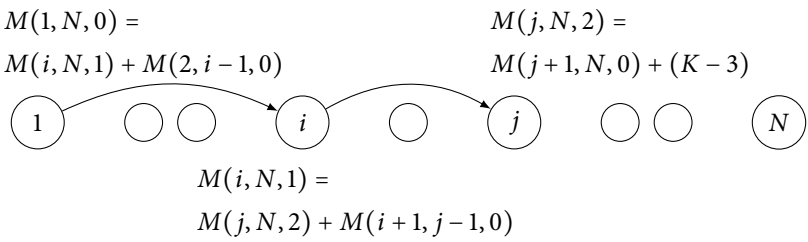


Figure 11.3: A visualization of the Zuma recursion. Marbles 1, i and j are to be merged. The rest of the interval is split up into three independent intervals that are solved recursively.

The other part of this process is vanishing marbles. At some point when solving a subproblem $M(l, r, s)$, the right answer might be that the l'th marble should not be merged with anything in $[l+1, r]$. We must then insert $K-1-s$ extra marbles, to get a group of K. This is the second part: $M(l+1, r, 0) + K - 1 - s$. Note that if $s > K-1$ this would be negative. This means that we never need to increase S beyond $K-1$, so we cap it at that in our recursion.

The DP has N^2K states taking $\Theta(N)$ time, for a complexity of $\Theta(N^3K)$. □

Sometimes we do DP on circular intervals. This is done slightly differently.

Running Routes – runningroutes

By Nathan Mytelka. 2019 ICPC North American Qualifier Contest. CC BY-SA 3. Shortened.

Polygonal School wants to increase enrollment, but they are unsure if their gym can support having more students. The gym floor is a regular n-sided polygon, affectionately called P. The coach has drawn several running paths on the floor. Each path is a straight line segment connecting two distinct vertices of P. During gym class, the coach assigns each student a different running path. The coach does not want students to collide, so each student's path must not intersect any other student's path, even at their endpoints.

Given all the paths, find the size of the largest non-intersecting subset.

Solution. If we solved the problem on a line, there would be little difference between this problem and Zuma. The relevant subproblem would be counting the maximum number of paths $R(l, r)$ within the interval $[l, r]$. For such a subproblem, there are two possibilities. Either l is the first endpoint of a path and some $l < i \leq r$ is the second, whereupon all remaining paths must lie either in $[l+1, i-1]$ or $[i+1, r]$ to avoid crossing the path $l \leftrightarrow i$ – note the two disjoint subproblems that arose – or it is not. In that case, all intervals lie in $[l+1, r]$. The recursion is thus $R(l, r) = \max(R(l+1, r), \max_{l<i\leq r}\{R(l+1, i-1)+R(i+1, r)\}$ where $l \leftrightarrow i$ must be a valid running path.

Solving the problem on a circle is not very different. The vertices of P can still be numbered 1 to n. The primary conceptual difference is that an interval may continue past n back to 1, giving us a right endpoint that is *smaller* than the left endpoint. For example, the interval $[n, 1]$ would represent only the two vertices n and 1, while the interval $[1, n]$ would represent the entire circle. With that in mind, the only change that needs to be made to the recursion is that the interval $l < i \leq r$ should be every vertex from l to r going clockwise. After this change, the answer is computed as the subproblem $R(1, n)$. □

This particular circular interval DP is among the simpler in that it's unusually easy to reduce to the line case. Sometimes that's harder, but the idea of thinking in intervals $[l, r]$ going clockwise from l to r is typically the right way.

Problem 11.10.
Arranging Hat arranginghat

Subset DP

Earlier in the chapter, we mentioned the longest path problem in general graphs. It was clear that we couldn't write a dynamic programming solution to it using only the location of the last vertex added to a path like we could for DAGs. The naive backtracking solution would take something like $\Theta(n!)$ time. There would be n choices for where to start, $n-1$ neighbors for each of them to visit, then $n-2$ choices for the third vertex, etc.

DP can't help us to do better than exponentially, but it can drastically reduce the time compared to that backtracking using the ***subset DP*** technique. It refers to doing dynamic programming where one of the parameters is a subset of some largest set.

Subset DP has two main use cases. For backtracking over permutations constructed one element at a time, we may be able to modify the recursion into only caring about *which* elements have been added to the permutation so far, rather than in what order. The result would be a reduction of the number of recursive calls from $n!$ to 2^n. That's what happens in the next problem, where we solve a variant of the NP-complete *traveling salesman problem*, a close cousin to the longest path problem, after which we examine the second use case.

Amusement Park – tivoli

By Arash Rouhani. Swedish Olympiad in Informatics 2012, Online Qualifiers.

Lisa has just arrived at an amusement park, and wants to visit each of the $N \leq 15$ attractions exactly once. For each attraction, there are two identical facilities at different locations in the park. Given the coordinates in the XY-plane of all the facilities, determine which facility Lisa should choose for each attraction to minimize the total distance she must walk. Lisa starts at the entrance at $(0, 0)$ and must return there after visiting every attraction.

Solution. Consider a partial walk constructed by naive backtracking, where we have visited attractions $s_1, \ldots, s_k$ and currently stand at the i'th facility of type j at (x, y). When deciding where to go next, it's completely irrelevant in what order the s_i were visited. We only care about what the set $S = \{s_1, \ldots, s_k\}$ is, since we do not need to visit any of the attractions in S again. A good DP state thus seems to be (S, i, j). Note that i, j only have at most 30 possibilities – two for each attraction. We also need to add an extra possible for the starting position. Since we have at most 15 kinds of attractions, the set S of visited attractions has 2^{15} possibilities. This gives us $31 \cdot 2^{15} \approx 10^6$ states. Each state can be computed in $\Theta(N)$ time, by looping over what attraction to visit next. All in all, we get a complexity of $\Theta(N^2 2^N)$.

```
1: procedure AmusementPark(i, j, S)
2:     if S contains every attraction then
3:         return dist((x_{i,j}, y_{i,j}), (0, 0))          ▷ tour is done – go back to the entrance
4:     answer ← ∞
5:     for every attraction s not yet in S do
6:         answer ← AmusementPark(s, 0, S ∪ {s}) + dist((x_{i,j}, y_{i,j}), (x_{s,0}, y_{s,0}))
7:         answer ← AmusementPark(s, 1, S ∪ {s}) + dist((x_{i,j}, y_{i,j}), (x_{s,1}, y_{s,1}))
8:     return answer
```

To code DP over subsets we generally use bitsets to represent the subset, since these map very cleanly to integers (and therefore indices into a memoization array). Revisit Section 6.5 if you don't remember how this works.. □

Subsets also appear naturally as a parameter in the backtracking recursion we base the DP on.

MeTube – dutub

By Arash Rouhani. Swedish Olympiad in Informatics 2018, School Qualifiers.

You should really be asleep by now, but you've kept watching *one more video* on MeTube all night... There are $C \leq 10$ different video categories (such as algorithm tutorials, funny cat videos and dishwasher repairs) that you are interested in. Each of the $N \leq 30$ videos on MeTube can belong to one or more categories (like a cat repairing a dishwasher as an example of a brute force algorithm). Before going to bed, you want to watch have watched videos in each category.

Given the videos, their categories and the lengths of each video, compute the minimum time you need to watch at least one video from each category.

Solution. Like in most cases where solution candidates are subsets of something, we apply a recursive solution where each video in turn either is included or excluded from the solution. After making this choice for the first k videos, the only state we need to keep track of is what categories the videos picked so far belonged to. This is a subset of size 2^C, which together with k gives $N2^C$ states, each of which takes constant time to process. □

Problem 11.11.

Programming Team Selection	programmingteamselection	
Paths	paths	*(all subtasks)*

Digit DP

Digit DP is a class of problems where we count the number with some certain properties, up to some given limit. These properties are characterized by having the classical properties of DP problems, i.e. being easily computable if we would construct the numbers digit-by-digit by remembering very little information about what those numbers actually were.

Palindrome-Free Numbers – palindromefree

By Antti Laaksonen. Baltic Olympiad in Informatics 2013

A string is a palindrome if it remains the same when it is read backwards. A number is palindrome-free if it does not contain a palindrome with a length greater than 1 as a substring. For example, 16276 is palindrome-free whereas 17276 contains the palindrome 727. The number 10102 is not valid either, since it has 010 as a substring (even though 010 is not a number itself).

Calculate the number of palindrome-free numbers between two given integers $0 \leq a \leq b \leq 10^{18}$.

Solution. A common simplification when solving counting problems on an interval $[a, b]$ is to solve the problem for $[0, a-1]$ and $[0, b]$ instead. The answer is then difference between the answer for the second interval minus the answer for the first one. These problems are much easier when counting the numbers in an interval starting at 0 instead.

Now comes an essential observation to turn the problem into a standard application of digit DP. Palindromes as general objects are very unwieldy in our situation. An iterative construction of numbers has to check digits far back in the number since any of them could be the edge of a palindrome. Fortunately, it turns out that any palindrome must

contain a rather short palindromic subsequence, namely one of length 2 (for even-length palindromes), or length 3 (for odd-length palindromes). Consequently, we only need to care about the last two digits when constructing the answer recursively. When adding a digit to a partially constructed number, it may not be equal to either of the last two digits.

Before arriving at the general solution, we solve the problem when the upper limit is $999\ldots999$ – exactly n digits of 9. Counting how many numbers we can construct of this type is then a straightforward recursion, where we construct the number one digit at a time (starting with the largest one), keeping track of the last two digits. We show C++ code for this since problem the implementation can be a bit tricky.

Snippet 11.1: Palindrome-free numbers, only 9's

```
long long palFree(int at, int len, int b1, int b2) {
    // We are done constructing the number
    if (at == len)
        return 1;
    long long answer = 0;
    for (int digit = 0; digit < 10; digit++) {
        // This digit would create a palindrome, so skip it
        if (d == b2 || d == b1)
            continue;
        // If the number has a leading zero and we add a new leading
        // zero, we make sure that the new "previous digit" is a
        // leading zero instead to avoid the palindrome check.
        if (b2 == -1 && d == 0) {
            answer += sol(at + 1, len, -1, -1);
        } else {
            answer += sol(at + 1, len, b2, d);
        }
    }
    return answer;
}

// We start the construction with an empty number with leading zeroes.
palindromeFree(0, N, -1, -1);
```

We fix the length of all numbers to have length n, by giving shorter numbers leading zeroes. Since leading zeroes in a number are not subject to the palindrome restriction, they must be treated differently. We represent them by the special digit -1 instead, resulting in 11 possible "digits". Once this function is memoized, it will have $n \cdot 2 \cdot 11 \cdot 11$ different states. Each state takes constant time to compute, so time-wise this is not an issue at all since n is bounded by 19 in the problem.

Once a solution has been formulated for this simple upper limit, extending it to a general upper limit is much easier. We first save the upper limit as a sequence of digits L. Then we need to differentiate between two cases in our recursive function. Either the *at* digits we have added so far are equal to the *at* first digits of the upper limit or they already form a smaller prefix than the upper limit. In the first case, we can't add a digit larger than

the next digit of the upper limit, or we the number would exceed the upper limit. In the other case the number we are making is lower than that of the upper limit no matter what digits we add.

These changes result in our final solution. Pay careful attention to it. While slightly tricky to get right the first time, most digit DP solutions follow this template.

Snippet 11.2: Palindrome-free numbers, general

```
vector<int> L;

long long palFree(int at, int len, bool equalToLimit, int b1, int b2) {
    // We are done constructing the number
    if (at == len)
        return 1;
    long long answer = 0;
    int maxDigit = 10;
    if (equalToLimit) {
        maxDigit = L[at];
    }
    for (int digit = 0; digit < equalToLimit; digit++) {
        // This digit would create a palindrome, so skip it
        if (d == b2 || d == b1)
            continue;

        bool newEqualToLimit = equalToLimit && digit == L[at];

        // If the number has a leading zero and we add a new leading
        // zero, we make sure that the new "previous digit" is a
        // leading zero instead to avoid the palindrome check.
        if (b2 == -1 && d == 0) {
            answer += sol(at + 1, len, equalToLimit, -1, -1);
        } else {
            answer += sol(at + 1, len, equalToLimit, b2, d);
        }
    }
    return answer;
}

// We start the construction with an empty number with leading zeroes.
palindromeFree(0, N, true, -1, -1);
```

□

Problem 11.12.

V	v
Hill Numbers	hillnumbers
Digit Sum	digitsum

Tree DP

Moving on to the last DP technique in this chapter, solutions become more abstract. When doing ***tree DP*** – dynamic programming on trees – subproblems and their parameters start

to lose their status as "just an optimized backtracking algorithm". This is in part because it's harder to visualize backtracking algorithms on tree problems as compared to problems on sequences, so the solutions seem less natural.

To do DP on trees, we start by rooting it arbitrarily and then solve subproblems for each subtree instead. The hope is that it's easy to compute the answer for the subtree of a vertex using the answers of it's children's' subtrees. That's often where the DP ends, memoizing a recursion performed on subtrees. Sometimes, we also use a *second* DP solution in order to combine the subtree answers if the problem is complicated enough. Just as for digit DP, most tree DP problems are very similar and tend to follow the structure, so make sure to internalize the following solution!

Fire Exits – fireexits

A museum has $N \leq 2000$ rooms connected by $N - 1$ corridors such that one can get from any room to every other room. The museum has no fire exits, but local regulations require that exits should be placed in a subset of rooms such that it's possible to get to an exit passing through no more than D corridors. Building exits in different rooms have different costs. Given those costs, compute the minimum cost to construct the required exits.

Solution. The rooms and corridors of the museum form a tree, which we start by arbitrarily rooting. As always in DP problems, we try to find choices to recurse on. Here, we have two choices for the root r of the tree – we either build or don't build a fire exit at it. In the first case we get a subproblem for each child v_i: computing the minimum building cost for that subtree if we already have a fire exit at distance 1 from v_i somewhere above v_i in the tree. For the purposes of the DP, we generalize the subproblem to letting the fire exit be at some arbitrary distance d from v_i instead. It will become clear why later on.

The other case is the tricky aspects of tree DP. Without an exit in r, an exit built somewhere in the subtree of e.g. v_1 could be the closest exit to vertices in the subtree v_2 in an optimal solution. Only two things matter for this spillover between subtrees though; what subtree has the fire exit closest to r, and the exit's distance to r. This yields a second subproblem for each v_i: computing the minimum building cost for a subtree if we promise to build an exit in the subtree not more than some known distance d away from v_i.

The answer to the full problem can be expressed as a subproblem of either type. For the first type, having an imaginary fire exit at distance $D + 1$ from the root for the first kind, and for the second promising to build a fire exit at most distance D from the root for the second kind. We also only have $O(N^2)$ states (d is always capped by D which is capped by N), so if we can solve the subproblems in constant time we're fine. However, we must first check that, for a given v and d, the subproblems can be solved recursively using only the answers to both questions for each child of v. Note that the two recursions are allowed to call each other! This multi-DP trick is a neat one that seldom is strictly necessary but can help to simplify some DP solutions.

We start with a slower solution that we then optimize. Call the first subproblem

$\text{HasUp}(v, d)$ and the second one $\text{NeedUp}(v, d)$. Let's compute $\text{HasUp}(v, d)$ first, i.e. there is already a fire exit d steps away from a vertex v further up in the tree. If we build a exit at v for cost c_v, we compute the total cost of the subtree as

$$c_v + \sum_{\text{child } u} \text{HasUp}(u, 1)$$

since each child u now has a fire exit at distance 1. The hard case is if v doesn't get an exit, and the exits may spill over between the child subtrees. There are two subcases here, depending on if the closest exit to v is the one d steps upwards in the tree, or in a subtree. In the first case, if $d \le D$, v needs no new exit and the closest exit to the children from further up is at distance $d + 1$, i.e. the cost is $\sum_{\text{child } u} \text{HasUp}(u, d + 1)$.

In the second case, loop over the smallest distance $d' < D$ from one of the children to an exit, as well as what child c has the exit in its subtree. That exit has distance $d' + 2$ to all the other children, so the total cost can be computed as

$$\text{NeedUp}(c, d') + \sum_{\text{child c'} \neq c} \text{HasUp}(c', d' + 2).$$

The recursion for $\text{NeedUp}(v, d)$ is similar. We either build an exit at v for the same cost as in the HasUp case, or one of the children must have built a exit that is at most distance $d - 1$ away from v. This last case can be computed in the same way too: loop over what the actual minimum distance $d' < d$ of one of the children is, and compute the cost using the same formula as in the first case.

```
1:  procedure HasUp(v, d)
2:      answer ← c_v + Σ_{child u} HasUp(u, 1)
3:      if d ≤ D then
4:          answer ← min(answer, Σ_{child u} HasUp(u, d + 1))
5:      for 0 ≤ d' < D do
6:          for child c do
7:              answer ← min(answer, NeedUp(c, d') + Σ_{c'≠c} HasUp(c', d' + 2))
8:      return answer
9:  procedure NeedUp(v, d)
10:     answer ← c_v + Σ_{child u} HasUp(u, 1)
11:     for 0 ≤ d' < d do
12:         for child c do
13:             answer ← min(answer, NeedUp(c, d') + Σ_{c'≠c} HasUp(c', d' + 2))
14:     return answer
```

These functions have three nested loops: one for d', one for c, and then there's a sum over all children to v. In the worst case, that could be $O(N^3)$ time. With N^2 subproblems we'd be looking at an upper bound of $O(N^5)$ time, way too much for $N = 2000$. We'll now start our journey to $O(1)$ per state!

The first factor N we win for free. When looping over all children in tree problems, remember that *on average* a vertex only has $O(1)$ children (there are N vertices and $N-1$ children in total).

In HasUp we win a second linear factor since the expression computed by the nested loop is actually independent of d. Thus we can compute it only once for each v, amortizing it out to $O(N)$ per state.

The same thing does not work as-is in NeedUp, since the nested loop actually depends on d. We throw in yet another useful DP trick. The only difference between NeedUp$(v, d-1)$ and NeedUp(v, d) is that the loop in the latter case includes NeedUp$(c, d-1)$ + $\sum_{c' \neq c}$ HasUp$(c', d+1)$, so the function can be simplified to

```
1: procedure NeedUp(v, d)
2:     answer ← c_v + Σ_{child u} HasUp(u, 1)
3:     answer ← min(answer, NeedUp(v, d − 1))
4:     for child c do
5:         answer ← min(answer, NeedUp(c, d − 1) + Σ_{c'≠c} HasUp(c', d + 1))
6:     return answer
```

Now, only a single factor N remains, which is the one hidden in the linear-time computation of the expression

$$\text{NeedUp}(c, d-1) + \sum_{c' \neq c} \text{HasUp}(c', d+2)).$$

We need one final trick which might be the most common one when working on this sort of thee problems. When the loop changes from a child c_a to a child c_b, the sum only changes with two terms: HasUp$(c_a, d+1)$ − HasUp$(c_b, d+1)$. The sum can thus be updated in constant time for each loop iteration.

After these optimizations, only computations that take linear time in the amount of children remain, which as stated is amortized $O(1)$ over all the vertices, so the total complexity is $\Theta(N^2)$. □

Exercise 11.13. In HasUp we have no explicit base cases. Is this a problem?

Exercise 11.14. In HasUp it may be the case that the function is invoked with $d = D+1$ in the recursive calls on lines 4 and 7. What does this subproblem represent? Can the function be called with even higher d?

Problem 11.15.
Chicken Joggers joggers

This solution was pretty hard and introduced a lot of tricks. In fact, that was most of our tree DP toolbox. Read it through a few times until you feel that you fully understand each step, and you will see that most tree DP problems become easy. The highlights were: amortized time complexity over all children, winning linear factors by updating sums

within loops that change with $O(1)$ factors, and noticing large amounts of overlap between subproblems (that NeedUp(v, d) and NeedUp$(v, d-1)$ computed almost the same thing) in order to avoid loops in the recursion. These techniques are general to all tree problems of this nature, not only the ones where we need DP.

11.4 Standard Problems

In your DP toolbox, you also need to master the following standard problems. The border between standard problems and techniques is blurry. The first problem, *Knapsack*, has so many common ubiquitous variations that it might be viewed as a technique instead. Similarly, traveling salesman – considered here as just an application of the bitset DP – could be considered a standard problem. The practical difference in the book is we look at the "normal" formulation of the standard problems rather than applied examples.

Knapsack

We start with the "traditional" knapsack problem and then briefly mention a few variations.

Knapsack – knapsack

You have a knapsack with an integer capacity C, and n different objects, the i'th with an integer weight w_i and value v_i. Select a subset of the items with maximal value, such that the sum of their weights does not exceed the capacity of the knapsack.

Solution. The common DP solution to this is $\Theta(nC)$, but depending on the constraints other solutions are also possible[1]. For very large C you can do a 2^n brute-force search, trying all subsets of the items, or with a little more effort do meet in the middle to reduce it to $2^{\frac{n}{2}}$.

Our approach is the normal one for DP on choosing subsets: we view the subset as a sequence of choices, where we either include an item or not. In this particular problem, the resulting DP state becomes very small. After including a few items, we are left only with the remaining items and a smaller knapsack to solve the problem for.

Letting $K(c, i)$ be the maximum value using at most weight c and the i first items, we get the recursion

$$K(c, i) = \max \begin{cases} K(c, i-1) \\ K(c - w_i, i-1) + v_i \quad \text{if } w_i \le c. \end{cases}$$

Translating this recursion into a bottom-up solution gives a compact algorithm:

```
1: procedure KNAPSACK(C, n, V, W)
2:     best ← new (n + 1) × (C + 1) array filled with −∞
3:     best[0][0] = 0
```

[1] In fact, $O(C \max w_i)$ can be done, but that's complex.

```
4:    for i from 1 to n − 1 do
5:        best[i] ← best[i − 1]
6:        for j from 0 to C do
7:            if W[i − 1] ≤ j then
8:                best[i][j] ← max(best[i][j], best[i − 1][j − W[i − 1]] + V[i − 1])
9:    return best
```

Exercise 11.16. How can you do the above DP in only $O(C)$ memory?

This computes the maximum values, but doesn't explicitly construct the subset. The reconstruction is very similar to the longest path in a DAG reconstruction but now has two parameters, which makes it look more complicated.

```
1: procedure KnapsackConstruct(C, n, V, W)
2:     best ← Knapsack(C, n, V, W)
3:     bestCap ← C
4:     for i from C to 0 do
5:         if best[n][i] > best[n][bestCap] then
6:             bestCap ← i
7:     for i from n to 1 do
8:         if W[i − 1] ≤ bestCap then
9:             newVal ← best[i − 1][bestCap − W[i − 1]] + V[i − 1]
10:            if newVal = best[i][bestCap] then
11:                add item i to the answer
12:                bestCap ← bestCap − W[i − 1]
```

□

Problem 11.17.

Exact Change	exactchange2
Canonical Coin Systems	canonical

The most common variation is when we are allowed to use each item an unlimited number of times. We formulate our DP solution in a similar way. The subproblem is the same, i.e. the maximum value obtainable by filling a knapsack of weight c using items $i, i-1, \ldots, 0$. In such a situation, we have two choices. We can either use i, leaving us with the same set of items but using at most $c - w_i$, or decide that we have finished using item i and look at the set of items $i-1, \ldots, 0$. The recursion is only changed by two characters:

$$K(c, i) = K(c, i-1) + K(c - w_i, i).$$

In Section 9.4 we briefly looked at a meet in the middle solution to the subset sum problem. This can also be viewed as a knapsack variant, where we ignore all the values and instead only focus on whether we *can* fill the knapsack with a certain weight or not. The knapsack DP can be adapted for subset sum with only minor changes, and conversely, the subset sum MITM can easily solve knapsack.

Problem 11.18.

Walrus Weights	walrusweights
Restaurant Orders	orders
Muzicari	muzicari

Longest Common Subsequences and Substrings

Another well-known class of DP problems is finding ***longest common subsequences and substrings*** of two sequences (see page 401 for a reminder of the terms).

Longest Common Subsequence

Given two sequences $A = \langle a_1, a_2, ..., a_n \rangle$ and $B = \langle b_1, b_2, ..., b_m \rangle$, find a longest sequence $c_1, ..., c_k$ that is a subsequence of both A and B.

Solution. When dealing with DP problems on pairs of sequences, a natural subproblem is *prefixes* of A and B. Here, some case analysis on the last letters of the strings A and B is enough for a solution. If the last letter of A is not part of a longest common subsequence, we can simply ignore it, and solve the resulting subproblem where the last letter of A is removed. The same applies to B. The remaining case is that both the last letter of A and the last letter of B are part of a longest common subsequence. In this case they can correspond to the same letter in a common subsequence, so that the remainder of the subsequence is the longest one we get after removing these two from the end of A and B. This yields a recursive formulation, which takes $\Theta(|A||B|)$ to evaluate since each state needs $\Theta(1)$ time:

$$\text{lcs}(n, m) = \max \begin{cases} 0 & \text{if } n = 0 \text{ or } m = 0 \\ \text{lcs}(n-1, m) & \text{if } n > 0 \\ \text{lcs}(n, m-1) & \text{if } m > 0 \\ \text{lcs}(n-1, m-1) + 1 & \text{if } a_n = b_m \end{cases}$$

□

Problem 11.19.

Prince and Princess	princeandprincess
Knight Search	knightsearch

Finding the longest common **substring** is not very different. The subproblem is instead finding the longest common substring that ends exactly at positions n and m in the strings, rather than the longest common substring of the prefixes. Then the answer for the cases where the last letters doesn't match is 0, so the recursion becomes

$$\text{lcs}(n, m) = \begin{cases} \text{lcs}(n-1, m-1) + 1 & \text{if } a_n = b_m \\ 0 & \text{otherwise} \end{cases}$$

and we find the answer by taking the max of all $\text{lcs}(n, m)$ results.

Longest Increasing Subsequence

Finding the ***longest increasing subsequence*** (LIS) is just as easy as finding a longest common subsequence. The LIS of a sequence A is actually the longest common sequence of A and A but sorted. This reduction gives you a simple way to compute the LIS in quadratic time.

Fortunately[2] the LIS can be found even faster than $\Theta(N^2)$ that the common subsequence reduction results in.

Longest Increasing Subsequence – longincsubseq

Given a sequence of n integers $s_1, \ldots, s_n$, find a longest increasing subsequence $a_1, \ldots, a_k$, i.e. where $a_i < a_{i+1}$.

Proof. The correct subproblem is finding the LIS of some prefix $[0, i]$ that ends with the element i. Let's denote the length of it $\mathrm{LIS}(i)$. A recursion that takes linear time per i to evaluate is simple. Among the smaller elements to the left we find the one with the longest increasing subsequence and extend it:

$$\mathrm{LIS}(i) = 1 + \max_{0 \le j < i \text{ and } s_i > s_j} \mathrm{LIS}(j).$$

To speed it up we do something very clever. We compute the values of $\mathrm{LIS}(i)$ in the order $i = 0, 1, \ldots$. At the same time, we keep an array where $B[x]$ contains the *smallest* value s_i of the sequence such that $\mathrm{LIS}(i) = x$, so far. With this array, we can rewrite the recursion in the following way:

$$\mathrm{LIS}(i) = 1 + \max_{s_i > B[x]} x.$$

Keeping the array updated is easy, after computing $\mathrm{LIS}(i)$ we set $B[\mathrm{LIS}(i)] = s_i$. This approach is still quadratic in the worst case.

To proceed, we must realize that B is a sorted array. If there is a LIS of length n that ends with an element x, we can just throw away the last element to get a LIS of length $n-1$ that ends with an element strictly smaller than x. This gives us the inequality $B[n-1] < B[n]$, so B is indeed sorted. Evaluating $\max_{s_i > B[x]} x$ is now a matter of finding the greatest x in B such that $B[x] < s_i$ in a sorted array B. This is easily done with `lower_bound` in C++ which you might remember from page 46. This function takes only logarithmic time, so that the full solution is $\Theta(N \log N)$.

To reconstruct the sequence, we need some extra bookkeeping. That code is very subtle, so we include it in C++. The implementation is slightly different from the description. Instead of $\mathrm{LIS}(i) = 1 + \max_{s_i > B[x]} x$ the equivalent computation $\mathrm{LIS}(i) = \min_{s_i \le B[x]} x$ (where this is taken to be $|B|$ if there is no such B) is used. Since B is sorted and contains no duplicates, these expressions are equivalent, but the latter is easier to compute with `lower_bound`.

[2] Or unfortunately, since this means more for you to remember!

Snippet 11.3: Longest increasing subsequence with reconstruction

```
vector<int> lis(const vector<int>& S) {
  if (S.empty()) return vector<int>();
  vi prev(S.size());
  vector<pair<int, int>> B;
  for (int i = 0; i < (int)S.size(); i++) {
    // The smallest x where B[x] >= S[i]
    int x = lower_bound(B.begin(), B.end(), make_pair(S[i], 0)) - B.begin();
    if (x == B.size()) {
      B.emplace_back();
    }
    B[x] = {S[i], i};
    prev[i] = x == 0 ? 0 : B[x - 1].second;
  }
  int len = B.size();
  int at = B.back().second;
  vi ans(len);
  while (len--) {
    ans[len] = at;
    at = prev[at];
  }
  return ans;
}
```

The meaning of `prev[i]` is to store the index of the element that precedes s_i in the LIS that ends at s_i, which is what enables the backtracking. To find it, we store not only sequence values in B, but also the index of the value as a pair. □

Exercise 11.20. How would you modify the solution to find the longest *non-decreasing* subsequence (meaning that it's enough that $a_i \leq a_{i+1}$)?

Problem 11.21.

Alphabet	alphabet
Panda Chess	pandachess
Train Sorting	trainsorting
Manhattan Mornings	manhattanmornings

Set Cover

We end the chapter with an application of subset DP on another classical NP-complete problem.

Set Cover

You are given a collection of subsets $S_1, S_2, ..., S_k$ of some larger set S of size n. Find a minimum number of subsets $S_{a_1}, S_{a_2}, ..., S_{a_l}$ such that

$$\bigcup_{i=1}^{l} S_{a_i} = S$$

i.e., *cover* the set S by taking the union of as few of the subsets S_i as possible.

Solution. For small k and large n, we can solve the problem in $\Theta(n2^k)$ by testing each of the 2^k covers. In the case where we have a small n but k can be large, this becomes intractable. Let us instead apply the principle of dynamic programming. In a brute force approach, we would perform k choices. For each subset, we would try including it or excluding it. After deciding which of the first m subsets to include, what information is relevant? If we consider what the goal of the problem is – covering S – it would make sense to record what elements have been included so far. This little trick leaves us with a DP of $\Theta(k2^n)$ states, one for each subset of S we might have reached, plus counting how many of the subsets we have tried to use so far. Computing a state takes $\Theta(n)$ time, by constructing the union of the current cover and the set we might potentially add. The recursion thus looks like:

$$\text{cover}(C,k) = \begin{cases} 0 & \text{if } C = S \\ \min(\text{cover}(C,k+1), \text{cover}(C \cup S_k, k+1)) & \text{else} \end{cases}$$

This is a fairly standard DP solution. The interesting case occurs when n is small, but k is *really* large, say, $k = \Theta(2^n)$. In this case, our previous complexity $\Theta(nk2^n)$ turns into $\Theta(n4^n)$. That's too slow for anything but very small n. To avoid this, we must rethink our DP a bit.

The second term of the recursive case of $\text{cover}(C,k)$, i.e. $\text{cover}(C \cup S_k, k+1)$, actually degenerates to $\text{cover}(C, k+1)$ if $S_k \subseteq C$. When k is large, this means many states are essentially useless. In fact, at most n of our k choices will actually result in us adding something, since we can only add a new element at most n times.

We have been in a similar situation before when solving the backtracking problem *Basin City Surveillance* in Section 9.2. We were plagued with having many choices at each state, where a large number of them would fail. Our solution was to limit our choices to a set where we *knew* an optimal solution would be found.

Applying the same change to our set cover solution, we should instead do DP over our current cover, and only try including sets which are not subsets of the current cover. So, does this help? How many subsets are there, for a given cover C, which are not its subsets? If the size of C is m, there are 2^m subsets of C, meaning $2^n - 2^m$ subsets can add a new element to our cover.

To find out how much time this needs, we use two facts. First of all, there are $\binom{n}{m}$ subsets of size m of a size n set. Secondly, the sum $\sum_{m=0}^{n} \binom{n}{m} 2^m = 3^m$. If you are not familiar with this notation or this fact, you probably want to take a look at Section 18.3.1 on binomial coefficients.

So, summing over all possible extending subsets for each possible partial C, we get:

$$\sum_{m=0}^{n} \binom{n}{m} (2^n - 2^m) = 2^n \cdot 2^n - 3^n = 4^n - 3^n$$

Closer, but no cigar. Intuitively, we still have a large number of redundant choices. If our cover contains, say, $n-1$ elements, there are 2^{n-1} sets which can extend it, but they all extend it in the same way – adding the last element. This sounds wasteful, and avoiding it probably the key to getting an asymptotic speedup.

It seems that we are missing some key function which, given a set A, can answer the question: "is there a subset S_i, that could extend our cover with some subset $A \subseteq S$?". If we had such a function, computing all possible extensions of a cover of size m would instead take time 2^{n-m} – the number of possible extensions to the cover. Last time we managed to extend a cover in time $2^n - 2^m$, but this is exponentially better!

The sum results in something different this time:

$$\begin{aligned}\sum_{m=0}^{n} \binom{n}{m} 2^{n-m} &= \sum_{m=0}^{n} \binom{n}{n-m} 2^{n-m} \\ &= \sum_{m=0}^{n} \binom{n}{m} 2^{m} \\ &= 3^n\end{aligned}$$

It turns out our exponential speedup in extending a cover translated into an exponential speedup of the entire DP.

We are not done yet – this entire algorithm depended on the existence of the magical "can we extend a cover with a subset A?" function. Sometimes, this function may be fast to compute. For example, if $S = \{1, 2, ..., n\}$ and the family S_i consists of all sets whose sum is less than n, an extension is possible if and only if *its* sum is also less than n. In the general case, our S_i are not this nice. Naively, one might think that in the general case, an answer to this query would take $\Theta(nk)$ time to compute, by checking if A is a subset of each of our k sets. Yet again, the same clever trick comes to the rescue.

If we have a set S_i of size m available for use in our cover. just how many possible extensions could this subset provide? Well, S_i itself only have 2^m subsets. Thus, if we for each S_i mark for each of its subsets that this is a possible extension to a cover, precomputation only takes 3^n time (by the same sum as above).

Since both steps are $O(3^n)$, this is also our final complexity. □

Exercise 11.22. This last step can be done in time $\Theta(k + n2^n)$. How?

Problem 11.23.

Square Fields (Hard)	squarefieldshard
Map Colouring	mapcolouring

Additional Exercises

Problem 11.24.

Selling Spatulas	sellingspatulas

Spiderman's Workout	spiderman	
Narrow Art Gallery	narrowartgallery	
Cheating a Boolean Tree	cheatingbooleantree	
Welcome to Code Jam	welcomehard	
Bus Planning	busplanning	
Maximizing Winnings	maximizingwinnings	
Nine Packs	ninepacks	
Hiding Chickens	hidingchickens	
The Uxuhul Voting System	uxuhulvoting	
Nested Dolls	nesteddolls	
Aspen Avenue	aspenavenue	
Presidential Elections	presidentialelections	
Constrained Freedom of Choice	constrainedfreedomofchoice	
Tight words	tight	
Springoalla	springoalla	*(all subtasks)*
Balanced Diet	balanceddiet	

Notes

Dynamic programming is one of the most useful algorithmic techniques to know. It appears in almost every contest at least once in, as you noticed, a large number of shapes and forms. There are many more DP techniques which we did not go through here, mostly related to different ways that a DP solution can be optimized.

The term itself is often attributed to Richard Bellman, one of the authors who the *Bellman–Ford* algorithm for shortest paths in weighted graphs is named after, which is very much a dynamic programming algorithm.

While many NP-complete problems such as TSP, knapsack, subset sum, set cover and so on have simple DP solutions that are fast enough for contest problems, they are seldom the ones of best proven time complexity or fastest in practice (which are two very different things).

Chapter 12

Divide and Conquer

A recursive algorithm solves a problem by reducing it to smaller instances of the same problem, hoping that their solutions can be used to solve the original instance. Most earlier examples shaved off a small bit of the instance before recursing on it. For example, a problem might have asked us to make n choices in the optimal way, and we reduced the problem to making $n - 1$ optimal choices instead. Furthermore, in dynamic programming the subproblems overlapped – solving two different subproblems required solving several common subproblems.

In this chapter, the recursive solutions take another approach. Instances are split into subproblems with a lot less (or no) overlap, *dividing* the problem. Predominantly, the solutions to these different parts are then combined to a solution to the original instance, *conquering* it.

12.1 Recursive Constructions

Recursive constructions constitute a large class of divide and conquer problems. The goal is to construct something, such as a tiling of a grid, cycles in a graph and so on. Divide and conquer algorithms reduce the construction of the whole object to constructing smaller disjoint parts that can be combined into the final construction. Such constructions are often by-products of mathematical induction proofs for the construction's existence. In the following problems, it's not initially clear whether the objects we are asked to construct even exist.

Grid Tiling – gridtiling

In a square grid of side length 2^n ($n \leq 8$), one unit square is blocked. Your task is to cover the remaining $4^n - 1$ squares with *triominos*, L-shaped tiles consisting of three connected squares. The triominos can be rotated by any multiple of 90° (Figure 12.1).

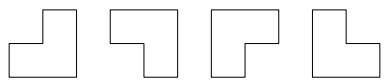


Figure 12.1: The four rotations of a triomino.

The triominos may not overlap each other, nor cover anything outside the grid.

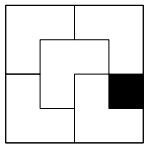


Figure 12.2: A possible tiling for $n = 2$.

Solution. Grid constructions are a good target for divide and conquer algorithms. The first step is to find out what smaller instances should be solved recursively. For this case, the side length 2^n hints at what those are. Aside from the property that $2^n \cdot 2^n - 1$ is evenly divisible by 3 (a necessary condition for a tiling to be possible), the fact that 2^n can repeatedly be split in half makes it natural to divide the grid into its 4 quadrants, as in Figure 12.3.

Exercise 12.1. Prove that $4^n - 1$ is divisible by 3.

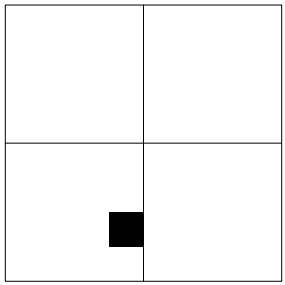


Figure 12.3: Splitting the $n = 3$ case into its four quadrants.

Each quadrant of a $2^n \times 2^n$ grid has the size $2^{n-1} \times 2^{n-1}$, the grid size for the case $n - 1$. We cannot recursively tile these four smaller grids immediately though. The crux lies in that the new grids lack the single black square that the original problem has. Indeed, a grid without a black square can not be tiled using triominos since 4^n is not divisible by 3.

Figure 12.4: A solution to the $n = 1$ case.

The solution lies in the trivial tiling of the $n = 1$ case, which actually reduces the problem to four instances of the $n = 0$ case (see Figure 12.4) – a 1×1 grid each containing only a single black square. In the solution, a black square was introduced in each of the three other quadrants by a triomino in the center, the only place where a triomino even can cover three quadrants at once. The same construction works on a grid of any size (Figure 12.5). After this modification, we can apply the the divide and conquer principle.

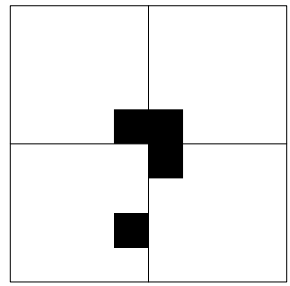


Figure 12.5: Placing a triomino in the corners of the quadrants without a black square.

Split the grid into its four quadrants, each of which now contains one black square. The quadrants can now be solved using the same procedure recursively. At some point, the recursion bottoms out at a 1×1 case, which needs no triomino to be tiled.

A recursive algorithm performing this tiling does only constant-time work, except for four recursive calls to itself. Each recursive call places exactly one triomino on the grid (except for base case $N = 0$). There are $\frac{4^n-1}{3}$ tiles to be placed, so the time complexity is $\Theta(4^n)$. This is asymptotically optimal, since this is also the size of the output. □

Problem 12.2. *Color Tiling* colortiling

We saw two key ideas here: looking at small cases to seek out a conquer strategy, and finding natural ways of dividing the structure we're constructing. Keep these ideas in mind, since they are often the right path to a solution.

While most divide and conquer problems are about constructing very tangible objects that you can draw on a piece of paper (like paths on a grid, tilings or colorings), the technique pops up in more abstract contexts too, as the next problem demonstrates.

Divisible Subset – divisiblesubset

Let $n = 2^k$, where $0 \le k \le 15$. Given $2n - 1$ integers, find a subset of size exactly n with a sum that's divisible by n.

Solution. A lot of progress can be made on problems by solving a few small cases by hand. This applies especially to construction problems, where the solutions for small inputs can hint at a generalizable pattern, or give you a clever insight into how to solve larger instances.

The $n = 1$ case here is uninteresting since it's trivially true. For $n = 2$, we get an insight that might not seem valuable but is key to the problem. The question that's asked is then, given $2 \cdot 2 - 1 = 3$ numbers, are there two numbers whose sum is even? Among any three numbers, there must be two that are either both even, or both odd. Both of these cases yield a pair with an even sum. This construction surprisingly generalizes to larger

instances. A recursion follows after attempting to find a way to combine solutions to the smaller instance.

We lay some ground work for a reduction of the case $2n$ to n. First, assume that we could solve the problem for a given n. The larger instance then contains $2(2n-1) = 4n-1$ numbers, of which we seek $2n$ numbers whose sum is a multiple of $2n$. This situation is essentially the same as for the case $n = 2$, except everything is scaled up by n. Can we scale our solution up as well?

The relevant question becomes: if we have three **sets of n numbers** whose respective sums are all **multiples of n**, can we find **two sets of n numbers** whose total sum is **divisible by $2n$**? Yes we can, by the same argument as for $n = 2$. Given three subsets with sums an, bn, cn, finding two whose sum is divisible by $2n$ is the same as finding two numbers among a, b, c whose sum is even – exactly the case $n = 2$.

A beautiful generalization indeed, but we still have some remnants of wishful thinking to take care of. The construction assumes that, given $4n-1$ numbers, we can find three sets of n numbers whose sums are divisible by n. We have now come to the recursive aspect of the problem. Recursively, we can pick any $2n-1$ of our $4n-1$ numbers to get our first subset. The subset uses up n of our $4n-1$ numbers, leaving us with only $3n-1$ numbers. We keep going, and pick any $2n-1$ of these numbers and recursively get a second subset. After this, $2n-1$ numbers are left, exactly what's needed to construct our third subset. The division is thus into *four* parts; three subsets of n numbers, and one set of $n-1$ which we throw away.

The complexity of the algorithm is more difficult to analyze than usual. The division and combination steps can be done in linear time, but makes 3 recursive calls with $\frac{n}{2}$. As a result, the complexity obeys the recurrence $T(n) = 3T(\frac{n}{2}) + \Theta(n)$. To find a closed form for this, consider the sum of $\Theta(n)$ at each of the $\log_2 n$ levels $T(\frac{n}{2^i})$ of the recurrence. At the first, there's a single one with cost $\Theta(n)$, then there's 3 with the cost $\Theta(\frac{n}{2})$, then 3^2 with cost $\Theta(\frac{n}{4})$ and so on. In total, that's

$$\sum_{i=0}^{\log_2 n} 3^i \cdot \Theta(\frac{n}{2^i}) = \Theta(n \cdot \sum_{i=0}^{\log_2 n} (\frac{3}{2})^i).$$

That inner sum equals

$$\frac{(\frac{3}{2})^{\log_2(n)+1} - 1}{\frac{3}{2} - 1} = \Theta((\frac{3}{2})^{\log_2 n}) = \Theta(n^{\log_2 \frac{3}{2}}) = \Theta(n^{\log_2 3 - 1}).$$

Multiplying this with n gives the final complexity $\Theta(n^{\log_2 3})$. □

It is typically easier to do the "divide" part of a divide and conquer solution first, but in this problem we did it the other way around – coming up with the division required us to solve the combination part first by generalizing the case $n = 2$. As always, the order in which you figure things out about a problem always trumps any formualic approaches.

Exercise 12.3. Another way to reduce the problem is to construct $n - 1$ pairs of integers with an even sum, throw away the last integer and scale the problem down by 2. What is the complexity then?

Problem 12.4.

Knight Tour	knighttour
Hamiltonian Hypercube	hypercube

12.2 Sequences

Another object that naturally invites divide and conquer solutions is sequences. The divide part is typically straightforward – sequences can be very naturally split in half – but the combination of the two halves can range from trivial to extremely difficult. The main principle is the same though: the problem we're solving must make sense to solve on disjoint parts of the original instance before investigating if divide and conquer is the correct way forward.

Inversions – inversions

Given an integer sequence $a_1, \ldots, a_n$ ($n \leq 200\,000$), we call the pair $1 \leq i < j \leq n$ an *inversion* if $a_i > a_j$. Compute the number of inversions in the sequence.

Solution. The problem gives us a clear invitation to try divide and conquer. Splitting the sequence in half gives us two contiguous subsequences, and the number of inversions *within* those sequences is closely related to the final answer. By recursively counting the number of inversions where both elements a_i, a_j lie in either half, we only need to add the number of inversions where a_i is in the left half and a_j is in the right half.

The last counting step can be done in $\Theta(n \log n)$ time. First, note that for a fixed a_i in the first half, the elements a_j in the second half it makes an inversion with are exactly those smaller than a_j. If the second half is sorted, those elements are a prefix of the half. Furthermore, if the first half is sorted too, this prefix never decreases if we gradually increase a_i. By iterating through the a_i in increasing order, it's thus enough to keep track of the largest a_j smaller than the current a_i, and check whether the prefix can be extended whenever we check a new a_i. [1] The idea is illustrated in Figure 12.6. The counting step itself only takes linear time, the $\Theta(n \log n)$ comes from the sorting.

We don't yet know of any sorting algorithms faster than $\Theta(n \log n)$, but perhaps we don't need one. During the recursion, we're sorting the same contiguous subsequences several times. For example, we're separately sorting both halves of the array **and** the array in its entirety. It's not unreasonable that the first results – the sorted halves of the arrays – can help sort the entire array. In fact, we can do this in linear time so that the entire combination step runs in linear time, for a final time complexity recurrence of $T(n) = 2T(\frac{n}{2}) + \Theta(n)$.

[1] This technique is called *two pointers*. We investigate it further in Chapter 13.

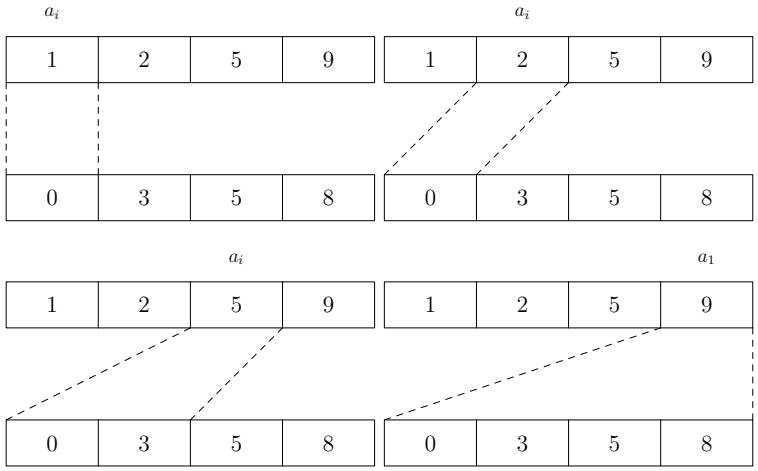


Figure 12.6: The prefix of the right that a_i from the left half creates inversions with. In total, $1+1+2+4$ inversions are made from these halves.

Exercise 12.5. Devise a linear time algorithm that, given two sorted arrays, combines them into a single sorted array.

To compute the complexity, we use the approach from *Divisible Subset*. By expanding the recurrence, we get that

$$T(n) = \sum_{i=0}^{\log_2 n} 2^i \Theta\left(\frac{n}{2^i}\right) = \Theta(n \log n).$$

□

Since we accidentally sort the entire array in the above solution, we also got $\Theta(n \log n)$ sorting algorithm. This sorting algorithm is called ***merge sort***.

Problem 12.6. *Heavy Subarrays* heavysubarrays

An issue with divide and conquer algorithms is that unless we split the sequence relatively evenly, the recursion can quickly degrade to quadratic time: consider $T(n) = T(1) + T(n-1) + \Theta(n)$ as the worst case for when the left half always has 1 element and the right $n-1$. Sometimes an uneven split can't be avoided. We must then reduce the amount of work done in the recursive function.

Non-Boring Sequences – nonboringsequences

By Adam Polak. Central European Regional Contest 2012.

A sequence is called non-boring if every contiguous subsequence contains a unique element, i.e. an element such that no other element of that subsequence has the same value. Given a sequence of

$n \leq 200\,000$ integers, decide whether it is non-boring.

Solution. The problem practically screams divide and conquer. By definition, the full sequence must contain a unique element a if it's boring. Every contiguous subsequence that goes through that element (at least) contain the unique element a, so it's enough to verify that all contiguous subsequences strictly to the left or right of a also contain a unique element. This is equivalent to the contiguous subsequence containing *all* elements to the left (or right) of a being non-boring.

This suggests the following recursive algorithm: find a unique element a, divide the sequence into two parts around a, and recursively check whether those parts are non-boring. Naively it takes linear time to find a, and if we're unlucky the unique element is always at one of the ends, so we get the $T(n) = T(1) + T(n-1) + \Theta(n)$ recurrence and a quadratic time complexity. In this case it's hard to avoid an uneven split since we can't control where a is. That means we must attack the other part of the recurrence – the $\Theta(n)$ complexity.

To find a unique element you **must** check each element at least once, so it might seem impossible to improve. The catch is that over all the recursive calls, we don't look for a unique element in a new sequence ever time, but rather connected subsequences of the original sequence. It's possible that we could pay a bit of precomputation to avoid a full linear scan to find a unique element. More specifically, assume that we had a way to tell in constant time whether a given element is unique within an interval. Then we'd only need to scan the sequence until we find a unique element, for the time complexity $T(n) = T(k) + T(n-k) + \Theta(k)$, where k is the first unique element we find.

Exercise 12.7. Devise a way to answer queries of the type "is the k'th element unique within an interval?" in constant time, given linear time preprocessing.

Sadly, if $k = n - 1$ (i.e. the unique element is at the end), the complexity recurrence is the same. We rescue the solution by scanning the sequence from *both* ends simultaneously, one element from each end at a time. This way, we only need to check $2k$ elements if the unique element is k elements away from an end, for the time complexity $T(n) = T(k) + T(n-k) + \Theta(\min(k, n-k))$. The worst case is when $k = n - k = \frac{n}{2}$ each time, resulting in $T(n) = \Theta(n \log n)$. □

A similar trick is when the partition of the array is done at a *random* point. Intuitively, a random point is on average close to the middle, so the split should be relatively even. It can be proven that the expected time complexity in this case is also $\Theta(n \log n)$.

Problem 12.8. *Bottle Caps* bottlecaps

12.3 Binary Search

The ***binary search*** is the most ubiquitous application of the divide and conquer technique. We've already used it once when solving Exercise 1.6, early on in the book. To illustrate the technique, we solve a simple problem: given a real number L and a **non-decreasing** function $f : \mathbb{R} \to \mathbb{R}$, find the greatest x such that $f(x) \leq L$. We must also know some reals lo and hi, such that $f(lo) \leq L < f(hi)$.

The trick is as simple as it is powerful. Consider the number $mid = \frac{lo+hi}{2}$. If $f(mid) \leq L$, then we know that the answer must lie somewhere in the interval $[mid, hi)$. The case $L < f(mid)$ instead gives us a better upper bound on the answer: it must be in $[lo, mid)$.

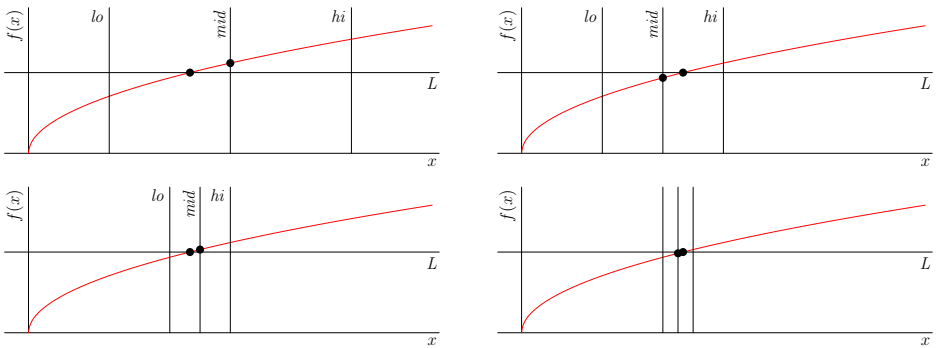


Figure 12.7: Three iterations of binary search.

By computing $f(mid)$ we halved the interval where the answer is. This step can be repeated until we get close enough to x. The example pseudo code iterates until it knows x within $prec$ of the exact answer.

```
1: procedure BinarySearch(lo, hi, L, prec)
2:     while hi − lo > prec do
3:         mid ← (lo + hi)/2
4:         if f(mid) < L then
5:             hi ← mid
6:         else
7:             lo ← mid
8:     return lo
```

Competitive Tip

Remember that the double-precision floating point type only has a precision of about 10^{15}. If the limits in your binary search are on the order of 10^x, using a binary search precision of something smaller than 10^{x-15} may cause an infinite loop. As an example, the following causes a binary search with precision 10^{-7} to fail.

```
double f(double x) { return 0; }
binarySearch(1e12, nextafter(1e12, 1e100), 0, 1e-7);
```

This happens because the difference between *lo* and the next possible `double` is larger than your precision (that's what the `nextafter` function generates). An alternative to handle precision issues is to binary search a fixed number of iterations:

```
double binarySearch(double lo, double hi, double lim) {
  for (int i = 0; i < 60; i++) {
    double mid = (lo + hi) / 2;
    if (lim < f(mid)) hi = mid;
    else lo = mid;
  }
  return lo;
}
```

The complexity of binary search depends on how good of an approximation is required. Originally, the interval we are searching in has length $hi - lo$. After halving the interval c times, it has size $\frac{hi-lo}{2^c}$. If we binary search until our interval has some size p, this means we must choose c such that $\frac{hi-lo}{2^c} \leq p$, which gives the bound $\log_2 \frac{hi-lo}{p} \leq c$. For example, if we have an interval of size 10^9 and seek precision 10^{-7}, this would require $\log_2 10^{16} = 54$ iterations of binary search.

Problem 12.9.

Suspension Bridges	suspensionbridges
Traveling Monk	monk

A classical application of binary search is to find the position of an element L in a sorted array A of length n. Applying binary search to this is straightforward, but it must be adapted into its discrete variant, i.e. the domain of $f(x)$ and the bounds *lo* and *hi* are integers. At first, we know nothing about location of the element. It's position can be any of $[0, n)$. Consider the middle index, $mid = \lfloor \frac{n}{2} \rfloor$, and compare $A[mid]$ to L. Since A is sorted, this leaves us with two cases:

- $L < A[mid]$, in which case, since the array is sorted, any occurrence of L must be strictly the left of *mid*, or
- $A[mid] \leq L$, and by the same reasoning, L can only lies to the right of *mid*.

We must repeat the search in one of the smaller intervals $[0, mid)$ or $[mid, n)$. At some point the interval consists of a single element $[i, i+1)$. Then either $A[i] = L$, or L is not present in the array.

```
1: procedure Search(array A, target L)
2:     lo ← 0, hi ← |A|
3:     while hi − lo > 1 do
4:         mid ← ⌊(lo + hi)/2⌋
5:         if x < A[mid] then
6:             hi = mid
```

```
7:          else
8:              lo = mid
9:      if x = A[lo] then
10:         return lo
11:     else
12:         return −1
```

Competitive Tip

When binary searching over discrete domains, care must be taken. Many bugs have been caused by improper binary searches.[a]

The most common class of bugs is related to the endpoints of your interval (i.e. whether they are inclusive or exclusive). Be explicit about this, and take care that each part of your binary search (termination condition, midpoint selection, endpoint updates) use the same interval endpoints.

[a] In fact, for many years the binary search in the standard Java run-time had a bug: `http://bugs.java.com/bugdatabase/view_bug.do?bug_id=6412541`

Exercise 12.10. Adapt the above binary search algorithm to find the smallest element greater than L (this is the `upper_bound` STL function.

Exercise 12.11. Adapt the above binary search algorithm to find the greatest element that's less than or equal to L (this is the `lower_bound` STL function).

Exercise 12.12. Given two sorted lists of lengths a and b, find the k'th smallest integer of their union in $\Theta(\log(a+b))$ time.

Problem 12.13.

Guess the Number	guess
Room Painting	roompainting
Out of Sorts	outofsorts

The discrete binary search can of course be adapted to any functions over discrete domains, not only to find elements in an array. It can also be slightly generalised – it's not always necessary (or possible) to perform the split into intervals to recurse into right at the middle.

Batmanacci – batmanacci

By Tómas Ken Magnússon and Bjarki Ágúst Guðmundsson. RU ÁFLV 2016. CC BY-SA 3.0. Shortened.

Let us define the Batmanacci sequence in the following manner:

$$s_1 = \mathtt{N}$$

$$s_2 = \mathtt{A}$$

$$s_n = s_{n-2} + s_{n-1}$$

where + is string concatenation. Now we get the sequence `N`, `A`, `NA`, `ANA`, `NAANA`, ... Given $n \leq 10^5$ and $k \leq 10^{18}$ what is the K'th letter in the N'th string in the Batmanacci sequence?

Solution. Since $s_n = s_{n-2} + s_{n-1}$, the character we're looking for must lie in one of two other strings that are also a Batmanacci string: s_{n-2} or s_{n-1}. Which one to search in depends on whether the length of s_{n-2} is smaller than k or not. To compute the lengths l_i of all strings s_i, note that they obey the recurrence $l_i = l_{i-2} + l_{i-1}$, i.e. they are just the Fibonacci sequence.

If $k \leq l_{i-2}$ the k'th character is in s_{n-2}, so we search for it in that string. Otherwise it's the $(k - l_{n-2})$'th character in s_{n-1}. At each step n is reduced by 1 or 2, so the complexity is $\Theta(n)$ after precomputing all l_i. □

Binary Search over the Answer

Many optimization problems can be formulated as a decision problem as well. For example, finding the longest path in a graph could be formulated as the decision problem "is there a path of at least length l in the graph?" To solve the optimization problem, binary search can be used over the decision problem to find the largest l such that there exists a path of that length. Sometimes a problem becomes significantly easier as a decision problem[2]. This technique is often referred to as ***binary search over the answer***.

Heating Up – heatingup

By Alexander Dietsch and Bjarki Ágúst Guðmundsson. NWERC 2021. CC BY-SA 3. Shortened.

Jonas just entered a chilli-eating contest. He has a pizza consisting of n slices, each containing chilli peppers. Initially slices i and $i + 1$ ($1 \leq i < n$), as well as 1 and n, are adjacent on the plate. Only one slice can be eaten at a time, and must be finished before a new slice is started. Jonas can pick any slice to eat first, but after that he is only allowed to eat slices that have at most one remaining adjacent slice. The spiciness of each slice is measured in Scoville Heat Units (SHU) between 0 and 10^{13}. Jonas has a certain spiciness tolerance, also measured in SHU, which corresponds to the spiciness of the spiciest slice that Jonas can tolerate eating. Additionally, after eating a slice of k SHU, his tolerance immediately increases by k. Determine the minimum initial spiciness tolerance necessary for Jonas to eat the entire pizza.

Solution. Whenever a problem asks you to determine the minimum or maximum of something, your first thought should be whether a binary search helps. The problem has the key property that if a tolerance of t SHU is enough, so is $t + 1$. Thanks to this, we can binary search for the lowest tolerance that is enough and instead only solve the decision problem: if the tolerance is t SHU, can Jonas eat the entire pizza?

This is easy to do in quadratic time. For each slice that requires at most t SHU to eat, simulate how much Jonas can eat if he starts with that slice in linear time. To improve this,

[2] This is unfortunately not the case for the longest path problem – the corresponding decision problem is also hard to answer.

note that if Jonas starts eating slice i and can eat another slice j during the simulation, j don't need to be tested as the starting slice. If he can eat the pizza by starting at j, he can do so starting at i too since his tolerance is higher once he reaches j compared to starting there. A possible algorithm would then be to go through each slice in order, perform the simulation, and then try the next slice that has never been eaten as the starting slice.

There's nothing immediately suggesting that algorithm wouldn't be quadratic. Theoretically, there could be a case such that if we start at slice i, we can eat all slices $1 \leq j \leq i$, but not slice $i+1$, nullifying our optimization entirely. Sometimes when we run into a suspected problem like this, it's a good idea to explicitly construct a test case that would trigger the worst-case complexity. Either we get a case to analyze further in the hope of improving our algorithm, or, as in this case, we encounter reasons for why a case like that can't actually exist.

The naive way of constructing a case would be to let slices have increasing tolerance requirements, such that a slice requires a higher tolerance than eating all previous slices would give us, i.e. $t_i = 1 + \sum_{i=j}^{i-1} t_j$. That construction immediately runs into an obstable – it would force the t_i to grow exponentially, meaning there could only be a logarithmic number of such slices. Based on this we might suspect that *any* case runs into a similar issue, and we would be right.

Assume that a specific slice k is eaten A times when starting at slices to the right of i. For each of those starting slices, let $a_1 < a_2 < \cdots < a_A$ be the rightmost slice that could be *not* eaten. When eating the starting slice that had a rightmost boundary of a_i, all the slices a_j with $j < i$ must be eaten too, since they lie between k and a_i (and by assumption we eat slice k). As a result $t_{a_i} > \sum_{j=1}^{i-1} t_{a_j}$, meaning they grow exponentially. Thus, each slice can be eaten only $O(\log(\max t_i))$ times over all tested starting points. As a consequence, the whole simulation is $O(n \log(\max t_i))$, which we need to run $O(\log(\max t_i))$ times. □

Problem 12.14.

Big Boxes	bigboxes
Finn the Giant	jattenfinn
Distributing Ballot Boxes	ballotboxes
Free Weights	freeweights
Ljutnja	ljutnja

The degree to which the binary search is necessary varies from problem to problem. It can often by a simplifying device for a problem where the optimization problem can be solved directly but requires more intricate implementation compared to the decision problem. One common use case where the binary search is essential is when maximizing averages.

Great GDP – greatgdp

By Olav Røthe Bakken. Bergen Open 2019. CC BY-SA 3.0. Shortened.

In your homeland Treetopia, there is exactly one way of travelling between any pair of the $N \leq 100\,000$ cites. A delegation from Cyclostan is coming to visit Treetopia. In order to impress the delegation, you want to take them to parts of the country such that the GDP per capita is maximized across the visited cities. The trip can include visiting cities on several branches in the country, and it is not possible to travel through a city without visiting it. The one and only airport in Treetopia is in the capital Treetopolis, and this is where the delegation from Cyclostan will arrive.

Solution. The problem with averages is that they are not easily additive. For an average $S = \frac{1}{n}\sum_{i=1}^{n} a_i$, the difference when adding a_{n+1} to it depends on the length of S: $\frac{a_{n+1}-S}{n+1}$. As such, optimization problems of a recursive nature often need to know not only the best average obtainable for a subproblem, but the best average *for each possible length*. Binary search helps us by reducing the optimization to a decision problem. Determining if it's possible to obtain an average *at least* S is equivalent to

$$\frac{1}{n}\sum a_i \geq S \Leftrightarrow \left(\sum a_i\right) - nS \geq 0 \Leftrightarrow \sum(a_i - S) \geq 0.$$

Suddenly the length of the sequence disappeared, only implicitly present in the number of terms in the final sum.

In the original problem, the GDP per capita equals $\frac{\sum g_i}{\sum p_i}$ where g_i is the GDP of the i'th city, an p_i its population. A binary search over the GDP per capita S turns the optimization part into the decision problem $\sum(g_i - p_i \cdot S) \geq 0$. The problem is then to find a connected subtree containing the airport such that this holds. Maximizing this value is much easier. Root the tree in the capital vertex v_c, and let $T(v)$ be the maximum value $\sum(g_i - p_i \cdot S)$ in the subtree with v as root assuming that v is included, so that the answer is $T(c_v)$. For each child c_i of v it can either be included (contributing $T(c_i)$ to the value) or not (contributing 0), so that

$$T(v) = g_v - p_v \cdot S + \sum \max(0, T(c_i)).$$

Computing this formula naively is amortized $\Theta(n)$, plus a logarithmic factor for the binary search. □

Problem 12.15. *Prosjek* prosjek2

12.4 Centroids

So far we've seen structures where dividing an instance into smaller ones comes naturally. Sequences can be split at half, grids into quadrants, and so on. We now turn our attention to trees, for which the correct divide and conquer strategy might be less obvious. We show three problems on this topic; a warmup showing a plethora of tree ideas, an interactive

one where we learn about the *centroid* through a guest appearance from the area of *streaming algorithms*, and finally an IOI problem that popularized the *centroid decomposition* technique.

Network – network

Baltic Olympiad in Informatics 2015

In Byteland, all the $n \leq 100\,000$ computers in the country are connected by $n - 1$ direct links between pairs of computers, such that there's a sequence of links between any pair of computers. A disadvantage of this setup is that if any single link is severed, the network is partitioned (i.e. there will be two sets of computers that aren't connected). Given the existing links, determine a *smallest* set of links to add to the network, such that network won't be partitioned by the breakdown of any single link.

Solution. First, we need to understand what kind of network we are to construct. If a given link $\{u, v\}$ goes down, there must be another path between u and v. This is equivalent to $\{u, v\}$ lying on a cycle. A graph where this holds, i.e. that each edge lies on a cycle, is called ***two-connected***.

One common way forward for problems of this kind is to look for a reasonable lower bound on the number of links that must be added, and prove that it coincides with the upper bound as well through giving an explicit construction. If we study the example in Figure 12.8, a lower bound pops up. Any vertex v that is a leaf in the graph must have a new link added to it, or it is impossible for the single edge $\{v, u\}$ adjacent to v to lie on a cycle.

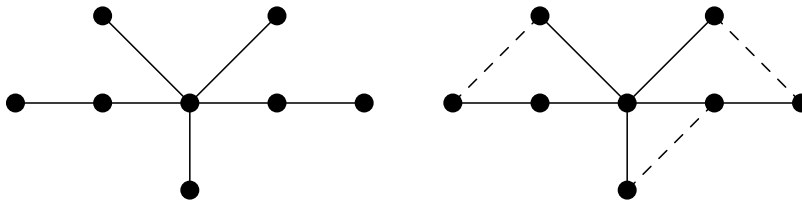


Figure 12.8: An example network and a possible solution with 3 new links.

A new link can be adjacent to at most two leaves, so if the graph has l leaves, at least $\lceil \frac{l}{2} \rceil$ new links need to be added. For more evidence in favor of this lower bound, note that any optimal solution can be transformed to only adding edges between leaves.

Exercise 12.16. Prove that there's an optimal solution where each link is added between a pair of leaves.

There are several possible valid constructions that achieve this lower bound. A natural recursive idea is to look at what happens to the graph when an edge is added between two leaves. We claim that after adding the link $\{u, v\}$, the path between them can be

merged into a single vertex, with all edges adjacent to the path instead connected to the new vertex. Each edge on the path is already on a cycle after the new link is added, so the merge maintains that all edges remaining in the graph need to be placed on a cycle. If it's always possible to find two leaves where this operation reduces $\lceil \frac{l}{2} \rceil$ by 1, this can be repeated $\lceil \frac{l}{2} \rceil$ times until the entire graph is only a single vertex, after which you're done. This would give an immediate $O(n^3)$ algorithm: for each pair of leaves, check whether compressing the path decreases $\lceil \frac{l}{2} \rceil$ (a linear time operation). Since we know that *each* leaf must be chosen at some point, we can directly improve this to $O(n^2)$ by fixing one of the leaves in the pair.

This algorithm is provably correct. Let's focus on determining when merging the path between a pair of leaves does **not** work out. Merging the path between u and v always removes those two leaves, so if $\lceil \frac{l}{2} \rceil$ doesn't decrease, another leaf must have appeared. The only possible case is that the vertex created by merging the path became a new leaf. That vertex is only a leaf if all other vertices on the path had degree 2, except one which had degree 3. When that happens, there must be some other leaf that lies on a path through the vertex of degree 3. If the path to that leaf is *also* of this form, there can only be one more leave in the whole tree it must be that there's only one more leaf in the whole tree, a case that's easy to solve directly.

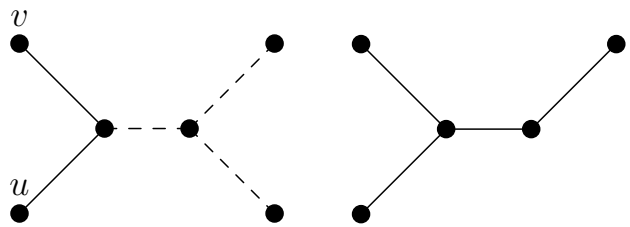


Figure 12.9: To the left, a pair of leaves that creates a new leaf if merged. To the right, the only kind of graph where this is true for each pair of leaves.

The fact that for each leaf u, there's only a single leaf v that is not acceptable to add a path between actually lets us improve this enough. By performing a DFS from u and keeping track of how many vertices of degree 3 and ≤ 4 seen so far, we can find another leaf v and make sure it's on a good path in time linear to the distance between u and v. This is not strictly true – if v is the "bad" leaf for u, we have to pay the cost for the DFS to v as well, but this happens with a sufficiently small probability for the average cost to be low enough. With careful coding the path merging can be done in time amortized $O(n \log n)$ (or better, as we learn in the next chapter), by always merging the path into the vertex with the highest degree and some other tricks.

We ended up with a long-winded solution from first principles that has a very icky final step. In the real contest, this is not the approach anyone who got full points took. If you were experienced enough to successfully work through all this reasoning and implement

the last part in $O(n \log n)$ time, you also probably know the following fact about trees: there exists a vertex such that no subtree of that vertex has a strict majority of the leaves, that we call the ***leaf centroid***. Given a leaf centroid c, you can divide all the leaves into pairs where the leafs are from different subtrees. By connecting each pair with a new link, you are guaranteed to make the tree two-connected. The vertices in each path lie in different subtrees of c, so the path must go through c and cover all the edges from each leaf up to c – but each edge lies on a path from c to a leaf.

The hard part is to find a leaf centroid. A greedy algorithm to find it comes out naturally from trying to prove its existence. □

Exercise 12.17. Prove that any tree has a leaf centroid.

The reason we didn't immediately jump to the easy-to-code linear solution is because the first solution contained ways to think about trees that are valuable in other situations, and the "nice" solution is much harder to intuitively derive without prior experience.

The ending to this problem serves as a nice segue into the next problem, where we need to find a similar central vertex.

Meeting Point – motesplatsen

By Joakim Blikstad. Swedish IOI Selection 2021.

$N \leq 25\,000$ friends (where N is odd) live on a tree, in one of the N vertices each. The friends want to designate one of the vertices as their meeting point. The optimal meeting point v should have the property that it minimizes the sum of the distances from each vertex to v. The distance between two vertices is defined as the number of edges on the path between them. Unfortunately, the friends can't recall exactly how the tree looks. The only know, for any three friends living in vertices a, b, and c, at what vertex v they meet when hanging out together, i.e. the vertex minimizing the sum of distances from v to a, b and c. Up to 500 000 times, you can ask for the meeting point of three friends. Find the optimal meeting point for all the friends.

Solution. Before anything else, you should solve the variant where you are given the entire graph, to better understand the problem. A modification of the leaf centroid algorithm finds the optimal meeting point in that case. Given a vertex v, if more than half of the vertices in the tree lie on the other side of an edge $\{v, u\}$, the sum of all distances decreases by moving to u. Consequently, there must be no such subtree for the optimal meeting point. A vertex where this holds is called a ***centroid***. Can there be more than one centroid? Yes – but never more than two, if and only if there's an edge where exactly half of the vertices lie on each side. Since the problem guarantees an odd N this cannot happen.

Exercise 12.18. Prove that a graph with an odd number of vertices has exactly one centroid.

We have identified the vertex we're after, and a linear time algorithm to find it when the graph is known. The problem might require us to find the centroid in another way than through the standard procedure, but we should first try to see if it works. There are two parts of the algorithm that need to be expressed through the questions from the problem:

finding the neighbour that's the root of the largest subtree, and determining the size of that subtree. If we can do this, we're able to traverse into the majority subtree all the way to the centroid. The primitive operation to build upon pops up if you think about what the answers are to the questions where you fix the vertex a in the question and vary b and c.

Exercise 12.19. Given vertices a, b, and c, devise a way to determine whether b and c are in different subtrees of a with a single question.

Thanks to Exercise 12.19, we can count the size of a subtree: pick a vertex in it, and count the number of vertices that are in the same subtree, one at a time. To find the neighbour that's the root of the largest subtree, we need to make one more observation by playing around with queries. For simplicity, we first introduce a useful concept for working with rooted trees.

> **Definition 12.1 — Lowest common ancestor**
> In a rooted tree, for two vertices u, v we call the common ancestor of u and v that is furthest away from the root their ***lowest common ancestor*** (LCA), denoted by $\text{lca}(u, v)$.

There's always at least a common ancestor since the root of the tree is the ancestor of all vertices, so the term is well-defined. Also, since a vertex is considered to be its own ancestor, it's possible that $\text{lca}(\text{u}, \text{v}) = u$.

The LCA is tightly related to the optimal meeting point of three vertices, and has a definition that's much easier to reason with going ahead.

Exercise 12.20. Prove that the optimal meeting point of a, b and c equals $\text{lca}(b, c)$ (if we consider the tree to be rooted in a).

From now on, we'll use $\text{lca}(b, c)$ whenever we mean the optimal meeting point of a, b, c whenever a root a is clear from context. Now, a simple follow-up tells us how to find the root of the largest subtree.

Exercise 12.21. Prove that $\text{lca}(b, c)$ is always closer to a than the furthest of b and c is whenever $b \neq c$.

A consequence of this is that in the largest subtree of a, its root v is the only vertex for which $\text{lca}(v, u) = v$ for any u in the subtree.

We can now design a solution to the first subtask where $N \leq 99$. Pick a random initial vertex a. Exercise 12.19 lets us partition all other vertices into their subtrees to find the largest one. By Exercise 12.21 we can also find the root of that subtree. We can then move to that vertex and set it as our new a. After at most 49 ($\approx \frac{N}{2}$) we must have found the centroid.

Let's count the number of queries this takes. For each a, the partitioning of all vertices can be done in $\frac{98 \cdot 97}{2}$ queries by asking all pairs of questions for b and c. The same answers are then used to find the root of the largest subtree of a. We never do this more than 49 times, for a total of 232 897 queries, well under 500 000.

We give the solution to the second subtask ($N \leq 999$) as a short aside. Asking all pairs of queries for a fixed a is actually enough to reconstruct the entire tree. This requires $\frac{998 \cdot 997}{2} = 497\,503$ queries. In some interactive problems where you're not given the underlying graph explicitly, performing a full reconstruction using the provided queries is actually the right solution (at least for a valuable subtask!), so it's a good idea to keep in mind.

Exercise 12.22. Given a vertex a, prove that the meeting points for a fixed a and all pairs b and c are enough to reconstruct the entire tree.

For a full solution, we need a different approach. The idea of moving towards the centroid one step at a time is unfortunately doomed to fail. In the worst case the initial vertex has distance 12 499 to the centroid. This leaves us with only 40 queries per step to find the largest subtree. Instead, we need a small shift in perspective. After finding the largest subtree of a vertex v, we know that no vertex in another subtree could possibly be the centroid. Therefore, we can *compress* all those subtrees into v as kind of a "super vertex" that represents a group of vertices. We never have to ask any specific questions about those vertices; it's enough to know how many vertices v represents at all times.

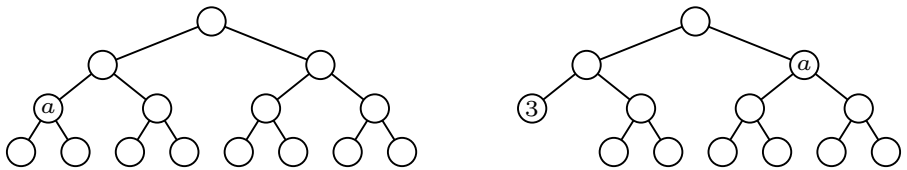


Figure 12.10: Compressing all smaller subtrees into a super vertex.

To avoid the 12 499 iterations, we must now realize that it's not actually important to move closer to the centroid all the time. Since the compression step removes at least one vertex (and hopefully more) from the tree at each point, a possible strategy is to just randomly choose a vertex that have not yet been merged into a super-vertex.

Figure 12.11: Performing the super vertex compression at randomly chosen vertices. Only the unlabeled vertices (i.e. those not yet merged into a super vertex) are chosen.

Picking a random vertex every step fails on a common graph counterexample, the *star graph*, which consists of a single vertex connected to $N - 1$ leaves. The star graph is a good sanity check for tree algorithms, since it can sometimes bring out quadratic behaviour from algorithms that seems to take linear time on random trees you generate yourself.

That's exactly what happens here. Since a leaf have no additional subtrees, no other vertices are merged into a super vertex if a leaf is picked. More often than not, we would pick at least half of the leaves.

Exercise 12.23. Prove that if a star graph has k leaves, the algorithm would pick at least $\frac{k}{2}$ leaves before picking the centroid with probability $\geq \frac{1}{2}$.

A good vertex to pick would be one where the largest subtree is small – that's when the most vertices are eliminated. A small largest subtree corresponds to being as close as possible to the centroid, since the largest subtree size decreases when moving towards the centroid. This provides motivation for choosing not a random vertex, but the *optimal meeting point* of three random vertices. Intuitively, that vertex should be on average closer to the centroid than a random vertex. This method works for the star graph case, lending further credence to it. We prove that a constant proportion of vertices are merged into a super vertex in each iteration of this kind.

Exercise 12.24. Given any three vertices x, y and z, prove that their optimal meeting point is one of $\text{lca}(x, y)$, $\text{lca}(x, z)$ or $\text{lca}(y, z)$.

This characterization us useful, because on average the subtree of the LCA of two randomly chosen vertices is big:

Exercise 12.25. Prove that if a and b are randomly chosen vertices in an N-vertex rooted tree, the probability that the subtree of $\text{lca}(a, b)$ has at most qN vertices is q (where $0 \leq q \leq 1$).

Choosing three random vertices a, b, c, the optimal meeting point is one of their pairwise LCA's. Since the probability of the subtree of a random LCA having at most qN vertices is bounded by q, the probability that *any* of the three pairwise LCA's having at most qN vertices is bounded by $3q$. This follows by the so-called *union bound* from probability theory – that for random events $a_1, \ldots, a_n$, the probability $P(\text{at least one event } a_i \text{ happens})$ is bounded by $P(a_1) + \cdots + P(a_n)$. To find the worst case, we should try to assign as much probability to the subtree being small as possible. Letting $p(x)$ be the probability that the subtree has size at most xN that means we want $p(x) = 3x$ for $0 \leq x \leq \frac{N}{3}$, so that the subtree size is uniformly distributed between 0 and $\frac{N}{3}$. The expected value of the subtree size is then $\frac{N}{6}$. As such we should on average need $\log_6(12499) \leq 6$ rounds before having eliminated all but one vertex. Sometimes we'll need a few more if we're unlucky, but with high probability we'll not need many more.

One final obstacle remains. We need a quadratic number of queries to find the largest subtree, but can only afford a linear amount. It's not strictly necessary to find the largest subtree though. When not at the centroid, the tree we are looking for always has a *strict majority* of all the vertices in the tree, so it's enough that we find the majority subtree, determine that one doesn't exist. The right idea comes from the solution to a famous

puzzle. In a sequence where one element occurs more than half of the time (called the *majority element*), find it in linear time **and constant extra memory**.

Many majority-related problems are based on the following fact:

Exercise 12.26. Prove that if a sequence A has a majority element, removing two *distinct* elements from a sequence does not change its majority element.

This lets us formulate a simple linear-time algorithm. Iterate through the sequence A, and keep a list M of remaining elements. Whenever you get a new element a, check M to see if it contains an element different from a, and, if so, throw away both of them. Otherwise, add a to M. After iterating through all of A, M contains all elements of A after throwing away pairs of distinct elements, so if A had a majority element, M has the same majority element. Furthermore, M never contains two distinct elements; when the latter of the elements were added, it would have been paired up with the first and thrown away. Thus, M contains some occurrences of the majority element of A.

This algorithm uses more than constant extra memory though, since the list M can in the worst case contain all of A. M only contains identical elements though, so we can replace it with two simple variables instead: the element that M contains, and its count:

```
1:  procedure Majority(list A)
2:      count ← 0
3:      el ← ?
4:      for a in A do
5:          if count = 0 then
6:              el = a
7:          if a = el then
8:              count ← count + 1
9:          else
10:             count ← count − 1
11:     return el
```

This is called the *Boyer-Moore majority vote algorithm*.

Exercise 12.27. Prove that if A contains an element with frequency *exactly* one half, Majority does not necessarily return it.

Exercise 12.28. Adapt the Majority algorithm to find a vertex in the majority subtree.

There's no need to check after the fact that the subtree we found actually contains more than half of the vertices. The only time it doesn't is when the vertex we picked as root is the centroid, but in that case it's fine to compress any subtrees since none of them contrains the centroid.

The number of queries is expected to be at most $28 \cdot 12\,499 < 500\,000$. Since the number of remaining vertices quickly decrease, the number of queries we need is even smaller. □

A long solution with many different steps, insights and detours.[3] Make sure to write your own implementation to make sure you understood the solution to the (many) exercises in the solution!

Problem 12.29.

We end the chapter with a divide and conquer classic that shows a famous centroid technique: ***centroid decomposition***.

Race – race

By Martin Fixman. International Olympiad in Informatics 2011. Shortened.

During IOI 2011, Pattaya City will host a race: the International Olympiad in Racing (IOR). As the host, we have to find the best possible course for the race. Around Pattaya there are $N \leq 200\,000$ cities connected by $N - 1$ bidirectional highways of integer lengths, such that there's exactly one sequence of highways connecting any pair of cities. The IOR regulations require the course to be a path whose total length is exactly $K \leq 1\,000\,000$, starting and ending in different cities. No highway may be used twice on the course to prevent collisions. Find the smallest number of highways in a possible course.

Solution. At a first glance, the problem looks like a bog-standard tree DP problem, but the constraints quickly destroy all the normal ideas. A $\Theta(N^2)$ solution for the first subtask ($N \leq 1\,000$) comes immediately: check all the possible courses. For the subtask $K \leq 100$, the right tree DP question is "what's the path with fewest highways of length k that has one endpoint at v and the other in the subtree of v?" These answers can be computed simultaneously for all lengths $k \leq 100$ given all the answers for the child vertices to get a $\Theta(NK)$ solution, clearly not enough for the full problem.

Instead, we must generalize the divide and conquer approach to trees. When counting inversions, the key insight was that an inversion has either both elements in one of the two halves of the array or one element in each, so that it must cross the midpoint of the array. The same applies here. If you pick any vertex v, the race course must either lie strictly inside one of the subtrees, or it must pass through the vertex. The problem can thus be reduced to only finding the best course that goes through v, and then recursively solving the problem for each subtree. This can be done in linear time.

Exercise 12.30. Devise a linear time algorithm to find the best race course that passes through a given vertex v.

Just any choice of v won't work, however. If we're not careful, we can recurse into a subtree a linear number of time, degrading the performance to quadratic. The counterexample from Meeting Point is applicable here again – for the star graph, choosing v as one

[3] If you got lost along the way due to the the probability theory, you might want to spend some time in a discrete mathematics textbook to brush up your knowledge. In particular, the union bound is useful in the proofs of some probabilistic algorithms.

of the leaves leaves us with the same tree minus a single vertex, and by Exercise 12.23, this happens a linear number of times with high probability. We have already learned what the right choice of v is if we want small subtrees: the centroid. It's not too hard to convince yourself that the worst-case is a graph that is split into only two subtrees, with as even size as possible, to get the time complexity recurrence $T(n) = 2T(\frac{n}{2}) + \Theta(n)$, which we know has the solution $T(n) = \Theta(n \log n)$. □

Problem 12.31. *Cities* cities

Additional Exercises

Problem 12.32.

Bell Ringing	bells
Routing	routing
Graškrižja	graskrizja
Slikar	slikar2
Mountain Running	mountainrunning
Unscrambling a Messy Bug	unscrambling
Wi-Fi	wifi
Nice Path	trevligvag
Julmust	julmust
Fence Bowling	fencebowling
Financial Planning	financialplanning

Notes

The Boyer-Moore majority algorithm [8] was later generalized to find those elements with at least frequency $\frac{1}{c}$ by Misra and Grives [38]. While the Boyer-Moore algorithm was devised in 1980, the paper written in the year after was not published until 1991, even though it's referenced (as a future paper) by Mirsa and Grives in 1982 as a paper "to be published". The Boyer-Moore paper, on the other hand, cites the Misra-Grives paper, introducing a circular reference (which would be a problem if you apply the paper reading strategy from *Citations*, pp. 161).

In this chapter, we have ignored plenty of divide and conquer algorithms that area of theoretical interest, but is either too hard to be of practical use in most problem solving, or not relevant for most contests. For example, the *quickselect* algorithm that can find the k'th smallest element in an **unordered** list in $\Theta(n)$ time, while neat, is not sufficiently much better than the various $\Theta(n \log n)$ approaches to warrant taking up space here. If you're interested in how it works, you can read Hoare's original paper [23]. Other topics we skipped were e.g. faster matrix multiplication (such as the *Strassen algorithm* [50]). Again, the algorithms that are simple enough to be of practical use don't improve that much over the $\Theta(n^3)$ naive algorithm. One of the earliest notable divide and conquer algorithms

is *Karatsuba's algoritm* [27], developed by Russian mathematician Anatoly Karatsuba and published in the early 1960's. It was the first example of integer multiplication in sub-quadratic time (in the number of digits). While this is actually of practical use, it's significantly less efficient than *Fast Fourier Transform* based multiplication.

In this chapter we never had to do very hard analysis of time complexities. For more complex divide and conquer recurrences, the *master method* is able to resolve most that arise in practice. If you ever stumble upon one of them, that's the primary tool to use (see e.g. [11] for the details).

Chapter 13

Data Structures

This chapter extends Chapter 6 further by showing some of the more advanced core data structures that make an appearence in algorithmic problem solving. The approach is not to explain the data structures directly. We instead present them in the same was as we solve problems or derive algorithms, by gradually improving an initial naive solution using additional insights. Some techniques shown in that process are useful in their own right, so it might be worthwhile reading the description of a data structure you already know.

Not everything in the chapter is an actual data structure. Techniques such as *square root decomposition* or *merge smaller into larger* are instead general techniques for different data structures. On the other hand, something like *two pointers* is not even that. The common theme is instead the type of problems that we solve, and to some extent what algorithmic problem solving tradition has classified techniques as data structure related.

13.1 Union-Find

The first problem we're studying has had many names througout history: *set union*, *equivalence*, *incremental graph connectivity*, *union find*. It's most frequently called *union find* in the algorithmic problem solving community, so that's the name used in this book. Union-Find is an easy to explain problem with a surprisingly short solution.

Union-Find – unionfind

Given the $N \leq 1\,000\,000$ sets $\{1\}, \{2\}, \ldots, \{N\}$, process $Q \leq 1\,000\,000$ of the following queries:

- = a b – replace the sets containing a and b with their union,
- ? a b – determine whether a and b are members of the same set.

Solution. The problem is often formulated in terms of graphs. Given a graph containing N isolated vertices, you need to support two operations: adding an edge between two vertices a and b, and determining whether two vertices a and b are in the same connected component. That's exactly the same as Union-Find, since adding an edge between two vertices is equivalent to merging their connected components.

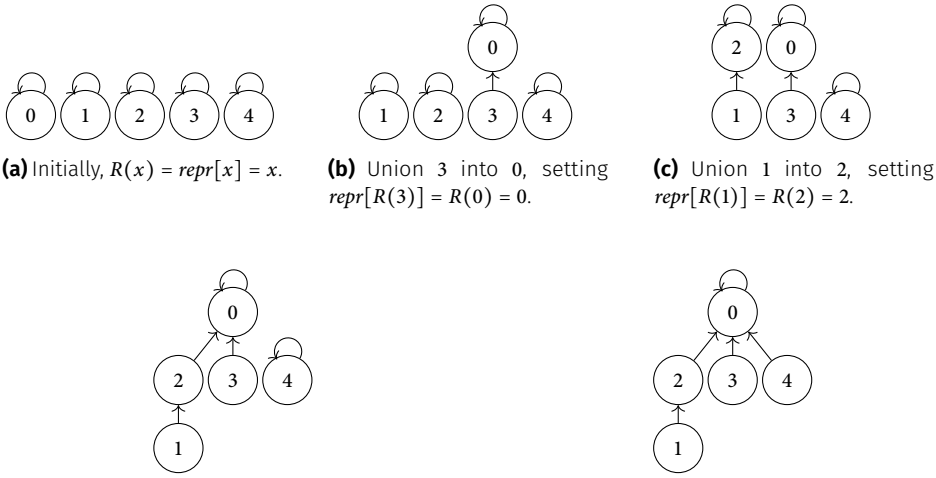
The problem can easily be solved in $O(NQ)$ time. Represent all sets explicitly using vectors. The sets containing a and b can be found in $O(N)$ time by looping through all the sets and can then be merged by moving all elements from one of the sets to the other.

A good way of attacking data structure problems with multiple query types is to first try and make each individual query type faster than a naive solution. If we're lucky, we might come up with a data structure that can then be generalized to all query types. Let's start by looking at the ? type queries. To make them faster than $O(N)$, we need to quickly find the set that a given element is in. The normal way is to designate one of the elements in each merged set as its *representative*. Each element x stores its representative $repr[x]$ in an array, so that a and b are in the same set if and only if $repr[a] = repr[b]$. Initially each element is in its own set, so $repr[x] = x$. When merging two sets, we pick the representative of one set and update the representatives of all elements in the other set to be the first set's representative. Unions take no extra time, but reduces the time for ? queries to $\Theta(1)$.

In the worst case, the merge operations can still take linear time. For each union we must choose which set should have its elements moved into the other set. If we merge the set containing 1 with each other set in order, we'll update all values in the large set each time if we're unlucky. There's a natural fix though. It's clearly better to always insert the elements of the smaller set into the larger rather than the other away, a technique called *merge smaller into larger*. This merging strategy has a worst-case amortized complexity of $\Theta(\log N)$ for a join query, while keeping constant time finds. If an element that is part of a l-size set is merged into another (by assumption larger) set, the union set gets a size $\geq 2l$. The size of the set an element an element is moved into thus at least doubles every time. Since the size can't exceed N, this can happen at most $\log_2 N$ times. This holds for each element, so in total there can be at most $N \log_2 N$ times an element must be merged into another set.

For further improvement, we must think differently. Let $R(x)$ denote the representative of the set containing x. When we merge two sets a and b, let's only update *repr* of the representatives of the set. More specifically, we set $repr[R(a)] = R(b)$. It's no longer possible to determine whether two elements are in the same set by comparing only *repr* now, since this value is not necessarily equal to the representative of the entire set. This new procedure is best explained by Figure 13.1. The main idea is to view each set as a rooted tree, where $repr[x]$ points to the parent of x in the tree. To merge two sets, it's enough to take the root of one tree (i.e. the representative of the set the tree corresponds to) and set its parent to the root of the other tree. To find the representative of the set in which the element x is present, we go upwards in the tree from x repeatedly by setting $x \leftarrow repr[x]$, until we reach the root (an element with $repr[x] = x$. We are again presented with the choice of which tree should point to the other. This time, the find operation takes at most linear time in the height of the tree. Always pointing the smaller tree, measured by the number of elements in it, into the larger guarantees that trees are logarithmic in height. In the context of union-find, this is called *union-by-size*.

Exercise 13.1. Prove that the height of all trees are bounded by $O(\log n)$ if the smaller tree is always pointed to the larger tree.



(a) Initially, $R(x) = repr[x] = x$.

(b) Union 3 into 0, setting $repr[R(3)] = R(0) = 0$.

(c) Union 1 into 2, setting $repr[R(1)] = R(2) = 2$.

(d) Union 1 into 3, setting $repr[R(1)] = R(3) = 0$.

(e) Union 4 into 2, setting $repr[R(4)] = R(2) = 0$

Figure 13.1: Union-Find unions over 5 elements.

The representative $R(a)$ is found by a simple recursive function. Union also has a rather short pseudo code.

```
1: procedure Union(a, b)
2:     a ← Find(a)
3:     b ← Find(b)
4:     if a = b then
5:         return
6:     if size[a] < size[b] then
7:         swap a and b
8:     size[a] ← size[a] + size[b]
9:     repr[b] ← a
```

```
1: procedure Find(a)
2:     if repr[a] = a then
3:         return a
4:     return Find(repr[a])
```

```
1: procedure Size(a)
2:     return size[Find(a)]
```

```
1: procedure SameSet(a, b)
2:     return Find(a) = Find(b)
```

The implementation is short, but still of logarithmic complexity. We'll improve this with a final touch. During the recursion to find the root $R(a)$, we can update all elements on the path have $R(a)$ as their parent with this one-line change to Find to gain an asymptotic speedup:

```
4: return repr[a] = Find(repr[a])
```

Intuitively, this compresses paths in the tree when they're traversed, saving time during future traverals. The complexity is instead amortized $O(\alpha(N))$, where $\alpha(N)$ is the inverse Ackermann function, an extremely slow growing function that we treat as $\Theta(1)$ for all reasonable N. We omit the proof here; see the notes for a reference if you're interested. □

Competitive Tip

To save memory, the *size* and *repr* arrays are often merged into one. Instead of using $repr[a] = a$ to signify that a is a root, we set $repr[a] = -size[a]$ for those a that are roots. Then $repr[a]$ is negative if and only if a is a root. The necessary changes are to use $-repr[a]$ instead of $size[a]$, and updating Find to check whether $repr[a] < 0$ instead of $= a$.

Interestingly, path compression is also enough to guarantee amortized logarithmic time. It's harder to prove that than union-by-size being amortized logarithmic though.

Problem 13.2.

Where's my Internet	wheresmyinternet
Tildes	tildes

The way union-find is used in problems vary quite a lot. In our first example, the data structure is only used to keep track of the sizes of connected components as edges are added.

Bee Hives – bikupor

By Nils Gustafsson. Swedish Olympiad in Informatics 2021, Online Qualifiers.

Bie the Beekeeper is moving her bees out into old man Ljungström's forest. The forest is an undirected graph with $N \leq 200\,000$ vertices and $M \leq 400\,000$ edges. Bie may choose between 1 and $N - K$ vertices to place her bees on. Ljungström is also a beekeper, and will afterwards place his bees on the K highest indexed vertices among those that Bie did not choose. His bees are unusually agressive, so it's important that none of his vertices are adjacent to any vertex Bie picked.

Given the layout of the graph and $1 \leq K < N$, find a valid set of vertices that Bie should choose.

Solution. If there's a vertex not adjacent to any of the K highest indexed vertices, Bie can pick that vertex as her solution. Otherwise, let's study the structure of a valid vertex set. We borrow a page from the playbook for greedy constructions and look for constraints that we can prove applies to at least one solution. The goal is to prove that there is ia valid set with a simple structure that makes it easy to find.

Assume that the vertex with lowest index among those Ljungström picked is x. A simple constraint is that we never need to pick a vertex with a lower index than x. In a valid solution those vertices can be discarded since they don't affect Ljungström's choice.

A consequence is that if we throw away all vertices with an index below x, all of of the $N - x$ remaining vertices are chosen either by Bie or Ljungström. Call this new graph G_x. Both Bie's and Ljungström's vertices must be a set of connected components, or Bie necessarily have a vertex adjacent to one of Ljungström's. This gives us a cubic solution: for each possible x, construct G_x, find its connected components with a DFS, and choose Ljungström's subset of in total K vertices using a quadratic time knapsack DP. For a quadratic solution we need another constraint that removes the need for the knapsack.

Exercise 13.3. Prove that for some value of x, Bie can choose a single connected component in the graph.

Now only some implementation work remains to reduce the solution to linear. Thanks to Exercise 13.3 the only thing we need to know for each x is if there's a component of size $(N - x) - K$ in G_x. That's where union-find enters. The only difference between G_{x+1} and G_x is that the latter also contains the vertex x and it's adjacent edges, so the sequence of graphs $G_{N-1}, G_{N-2}, \ldots, G_0$ can be constructed by only edge additions. A union-find structure keeps track of all component sizes in the graph when new edges are added, so we just need to keep track of them in a set for easy lookup.

```
1:  uf ← new union-find structure
2:  sizes ← new set
3:  for i = N − 1 to 0 do
4:      sizes.add(1)                    ▷ Vertex i is an isolated component before its edges are added
5:      for each neighbor j of i do
6:          if i < j and not uf.sameSet(i, j) then      ▷ Only add edges to previously added vertices
7:              sizes.remove(uf.size(i))
8:              sizes.remove(uf.size(j))
9:              uf.union(i, j)
10:             sizes.add(uf.size(i))
```

□

Exercise 13.4. Prove that for some value of x, Bie can choose the *smallest* connected component in the graph.

Problem 13.5.

Killing Chaos	killingchaos
Park	park

Sometimes keeping track of the components is a more essential aspect of the problem. We may also need to augment the union-find structure with more data. This is the case in the following problem.

Subway Planning – subwayplanning

By Simon Lindholm. Nordic Olympiad in Informatics 2017.

The Stockholm subway system consists of N stations with $N - 1$ pairs of stations directly connected by tracks, such that it's possible to travel between any pair of stations. To minimize average commute time, the city council has decided to change the subway. The new plan also consists of a set of $N - 1$ tracks between the N stations, possibly with some tracks in common with the old plan. Each weekend you may remove the tracks between one pair of stations and build tracks between another pair of stations. It must be possible to travel between all pairs of stations after each weekend.

Compute the smallest number of weekends you need to implement the new subway plan, and determine what track should be closed and opened during each weekend.

Solution. The subway plans represent two trees. The task is to transform one into the other by repeatedly replacing an edge with another while ensuring connectivity after each replacement. If we were to make an optimistic guess about the number of times an edge must be moved, we'd probably guess at the only obvious lower bound: the number of tracks in the new plan that are not present in the first plan, which by symmetry equals the number of tracks only present in the old plan. This bound can be attained if and only if we can remove an edge from the old plan and add an edge in the new plan every weekend. It's always possible to do so, and it's easy to determine how in linear time for each weekend, resulting in a simple quadratic algorithm.

Exercise 13.6. Prove that for any edge e present only in the new plan, it's possible to remove an edge from the old plan and add e during the same night.

For a faster algorithm, we must restrict which edge we move each night. Assume that there's a leaf in the old plan where its single adjacent edge is not present in the new plan. This edge can always be chosen to be replaced by an edge from the new plan. Since the edge was adjacent to a leaf, the new edge must also be adjacent to the leaf but have a different second endpoint to keep the leaf connected. This other endpoint must be one of the vertices to which the leaf is connected in the new plan, of which at least one must exist if the new plan is a tree. If the edges of the tree are stored in adjacency lists, finding this replacement edge is a as simple as picking any of its neighbours in $\Theta(1)$.

The tricky part is to quickly find a leaf. An immediate obstacle is that a leaf of this specific type might not exist, because all edges adjacent to a leaf in the old plan might be present in the new plan too.

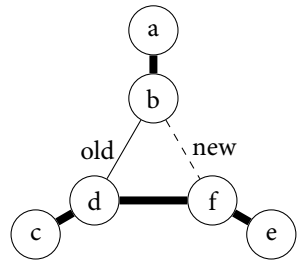


Figure 13.2: An example where the two plans differ with exactly one edge. Edges common to the two plans are drawn in bold. The edges adjacent to the three leaves in the old plan are all present in the new plan as well.

To fix this we must use another graph theoretical tool – that of ***edge contractions***. We already know that if an edge is present in both the old and the new tree, it will never be replaced at any point. These edges are thus unhelpful in solving the problem. Given such an edge $\{u, v\}$, we can *contract it*, meaning that the edge is removed and the vertices u

and v are replaced with a new vertex uv. An edge with adjacent to either u or v is instead assigned the vertex uv as endpoint.

Exercise 13.7. Prove that if an edge is contracted in a tree, the resulting graph is still a tree.

In the graph where all the common edges are contracted, the same problem doesn't arise. It's not immediately obvious why we can work on the contracted trees rather than the original ones. The insight is that each set of vertices that makes up one of the contracted vertices forms connected components after each night, since they contain only edges that are present in the new plan. Thus, as long as each of these grouped vertices are connected, the entire tree is connected too.

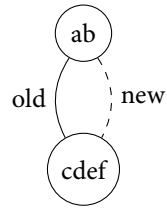


Figure 13.3: The same graph with all common edges contracted.

As the contracted graph in Figure 13.3 shows, a consequence is that there might be both an old and a new edge connecting the same pair of contracted vertices. This is not a problem, but it highlights that during the contraction process we must also keep track of the *original* endpoints of each edge, so that we can output the exact edges moved each night correctly.

Maintaining this contracted graph is where union-find enters. Contracting edges is almost exactly what the union operation does. The main difference is that we must still keep track of the edges that are adjacent to each contracted vertex (i.e. union-find component). Augmenting the union-find structure with lists of what edges are adjacent to each component – in both the old and the new trees – and merging them when two components are joined due to an edge contraction is enough. □

Edge contraction can be used for general graphs as well, but then you have to be careful about merging the component *with fewer adjacent edges* into the one with more adjacent edges, or you would get a quadratic-time worst case. For trees this is not necessary.

Exercise 13.8. Prove that contracting edges with union-find in a tree is $\Theta(N \log N)$ with the standard approach.

The merge smaller into larger trick is generally applicable when you're repeatedly merging data structures for which insertions are fast, since the total number of insertions is bounded by $N \log N$ no matter the data structure.

Prominence – primarfaktor

By Nils Gustafsson. Swedish IOI Selection 2017.

> What's the world's highest moutain? The peak of Mount Everest. OK, but what's the second highest mountain? The second highest peak of Mount Everest, of course.

This exchange highlights a problem with ordering the world's mountain peaks solely by their altitude. A possible solution is to instead order the peaks by their *prominence*. The prominence of a peak is the minimum altitude you need to descend to walk to to an even higher peak. By ignoring all peaks with a too low prominence you avoid all the silly small peaks that are actually just a part of a higher mountain.

Given a graph with $N \leq 100\,000$ vertices and $M \leq 400\,000$ edges, where each vertex has an altitude, compute the prominence of each vertex, i.e. the minimum altitude you must descend on a path to a vertex with a higher altitude. If it's impossible to reach a higher altitude, the prominence is equal to the altitude of the vertex.

Solution. Let's start with a slow solution, where we try to find the prominence for only a single vertex v. Since we're trying to find the minimum altitude at which something – walking to an even higher peak – is possible, a binary search over the answer seems like a decent attempt[1]. An answer h is possible if there's a path from v to a higher peak when restricted to passing only vertex at an altitude at most h less than the altitude of v. That's a plain connectivity problem, which we can solve with a DFS for a $\Theta(M \log N)$ solution.

A reduction to connectivity is a strong for a union-find solution. The standard transformation to solve the problem for all vertices is replace the binary search with something akin to a linear search instead. Start with the empty graph, and add vertices back in order of descending altitude. When a vertex is added, all of its edges adjacent to a vertex at a higher altitude should also be added. The first time that a vertex has a path to a vertex at a higher altitude, the prominence of that vertex is given by the altitude of the vertex that was just added. This change gives us a $\Theta(NM)$ algorithm to find the prominence for all vertices: after each vertex addition, partition the graph into its components and see which vertices for the first time got another in the same component at a higher altitude.

Partitioning a graph into its components while adding edges to it quickly is exactly what we have union-find for. Augment the union-find structure with a list of vertices in each component that hasn't yet found a path to a higher altitude vertex. Consider what happens when an edge is added between two components, where the maximum altitude vertex in them are h_1 and h_2 respectively. If $h_1 > h_2$, all vertices in the second component now gets a path to a higher altitude vertex, namely the one in the first component at altitude h_1. All vertices in the first component could already reach it, so none of the vertices in that component finds such a path. Thus, after the merge we get an answer for all remaining vertices in the second component, but still need to find the answer for all

[1] Unless you have cheated and learned some of the graph algorithms from Chapter 14 on your own, which shows more standard approaches like a modified Dijkstra's.

remaining vertices in the first.

One small detail remains. By symmetry, we can handle the case where $h_1 < h_2$ as well, but what if $h_1 = h_2$? Then we can't find a path to a higher altitude vertex for *any* of the vertices in the two components. Instead, we must merge the two lists of vertices we are yet to find the answer for. Since these lists never exceed the total number of vertices in the respective components, this merge is amortized $O(N \log N)$ assuming we do union-by-size. □

Exercise 13.9. *Bandwidth 2* bandwidth2

13.2 Range Queries

We now turn our attention to queries on sequences, probably the richest topic in data structure when it comes to algorithmic problem solving. They typically consist of solving some given problem on an array of e.g. integers, but we have to solve the problems not only for one array, but rather a large number of sub-arrays of a single, larger array. In computer science, this is typically called a ***range query***. Algorithmic problem solving often mixes in the term "interval", "segment" and "sub-array" when talking about the same thing: a contigious subsequence of values in an array. This chapter freely mixes the terms depending on what the predominant terminology is.

While it's usually easy to solve the problem for a single interval in linear time, the large number of queries often forces us to solve the problem in e.g. logarithmic time. The common theme is precomputation – trading some extra memory by storing some relevant values ahead of time to answer queries quickly.

Prefix Precomputation

The simplest of the range queries is to compute sums of different intervals.

Interval Sum

Given a sequence of integers $a_0, a_1, \ldots, a_{N-1}$, you will be given Q queries of the form $[L, R)$. For each query, compute $S(L, R) = a_L + a_{L+1} + \cdots + a_{R-1}$.

Solution. The solution requires a transformation we have used before in the book, for example in the section on digit DP. Instead of computing the sum of the interval $[L, R)$, we can just as well compute the sum of the interval $[0, R)$ and remove the sum of the interval $[0, L)$, i.e: $S(L, R) = S(0, R) - S(0, L)$. This reduces the problem to only computing the sums $S(0, i)$. Of course we don't need to do this in the naive quadratic manner by computing each sum one at a time. Since $S(0, i + 1) = S(0, i) + a_i$, the sums can all be computed in amortized constant time one at a time. □

This precomputation is called *prefix sums*. Problems rarely ask you to only compute sums of individual intervals. Instead, it's often a small tool in a more complex algorithm.

This same technique works for any associative[2], invertible operation, not only addition. For example, Note that even operations we might normally think of as invertible aren't invertible for all elements, such as multiplication; the inverse operation is divisible, but division by zero is undefined.

Exercise 13.10. Adapt prefix precomputation to work with multiplication, even where some elements might be 0.

Another small caveat is that some care needs to be taken in case the operation is not commutative[3].

Exercise 13.11. Let $\star$ be a non-commutative operation. Show how to compute intervals $a_i \star a_{i+1} \star \cdots \star a_j$ using prefix precomputation.

Exercise 13.12. Show how to extend prefix sums to the two-dimensional case, i.e. given a matrix of integers $a_{i,j}$ where $0 \le i < N$ and $0 \le j < M$, compute sums of the form $\sum_{i=b}^{t} \sum_{j=l}^{r} a_{i,j}$ quickly.

A common variation of prefix sums is to count the number of times each prefix appears. The following problem is a normal application of the principle.

Counting Subsequences – subseqhard

By Michal Forišek. Internet Problem Solving Contest 2006. CC BY-SA 3.0. Shortened

You are given a sequence S of integers we saw somewhere in the nature. Compute the number of *contiguous* subsequences of S that sum to 47.

Solution. Applying the prefix way of looking at contiguous subsequences gives us the solution immediately. Assume that $a_i + a_{i+1} + \cdots + a_j = 47$. Letting $S(i) = a_0 + \cdots + a_i$, this is equivalent to $S(j) - S(i-1) = 47$, or $S(i-1) = S(j) - 47$. For each value of j, we thus want to count the number of earlier prefixes with a given value. Keeping a running count of all prefix values in e.g. a hash map as we sweep through the sequence to compute prexies is enough for a linear-time solution. □

Problem 13.13.
Divisible Subsequences divisible

Sparse Tables

The case where a function does not have an inverse is a bit more difficult.

Interval Minimum

[2] An operation $\star$ is associative if $a \star (b \star c) = (a \star b) \star c$.

[3] An operation $\star$ is commutative if $a \star b = b \star a$ for all a and b.

Given a sequence of integers $a_0, a_1, \ldots, a_{N-1}$, you will be given Q queries of the form $[L, R)$. For each query, compute the value

$$M(L, R) = min(a_L, a_{L+1}, \ldots, a_{R-1})$$

Generally, you cannot compute the minimum of an interval based only on a constant number of prefix minimums of a sequence. We need to modify our approach. If we consider the naive approach, where we simply answer the queries by computing it explicitly, by looping over all the $R - L$ numbers in the interval, this is $\Theta(len)$. A simple idea will improve the time used to answer queries by a factor 2 compared to this. If we precompute the minimum of every *pair* of adjacent elements, we cut down the number of elements we need to check in half. We can take it one step further, by using this information to precompute the minimum of all subarrays of four elements, by taking the minimum of two pairs. By repeating this procedure for very power of two, we will end up with a table $m[l][i]$ containing the minimum of the interval $[l, l + 2^i)$, computable in $\Theta(N \log N)$.

Sparse Table

```
vector<vi> ST(const vi& A) {
  vector<vi> ST(__builtin_popcount(sz(A)), vi(sz(A)));
  ST[0] = A;
  rep(len,1,ST.size()) {
    rep(i,0,n - (1 << len) + 1) {
      ST[len][i] = max(ST[len - 1][i], ST[len - 1][i + 1 << (len - 1)]);
    }
  }
  return ST;
}
```

■

Given this, we can compute the minimum of an entire interval in logarithmic time. Consider the binary expansion of the length $len = 2^{k_1} + 2^{k_2} + \cdots + 2^{k_l}$. This consists of at most $log_2 len$ terms. However, this means that the intervals

$$[L, L + 2^{k_1})$$

$$[L + 2^{k_1}, L + 2^{k_1} + 2^{k_2})$$

$$\ldots$$

$$[L + 2^{k_1} + \cdots + 2^{k_{l-1}}, L + len)$$

together cover $[L, L + len)$. Thus we can compute the minimum of $[L, L + len)$ as the minimum of $\log_2 len$ intervals.

Sparse Table Querying

```
int rangeMinimum(const vector<vi>& table, int L, int R) {
  int len = R - L;
  int ans = std::numeric_limits<int>::max();
  for (int i = sz(table) - 1; i >= 0; --i) {
    if (len & (1 << i)) {
      ans = min(ans, table[i][L]);
      L += 1 << i;
    }
  }
  return ans;
}
```

■

This is $\Theta((N+Q)\log N)$ time, since the preprocessing uses $\Theta(N \log N)$ time and each query requires $\Theta(\log Q)$ time. This structure is called a *Sparse Table*, or sometimes just the *Range Minimum Query* data structure.

We can improve the query time to $\Theta(1)$ by using that the min operation is idempotent, meaning that $\min(a, a) = a$. Whenever this is the case (and the operation at hand is commutative), we can use just two intervals to cover the entire interval. If 2^k is the largest power of two that is at most $R - L$, then

$$[L, L + 2^k)$$

$$[R - 2^k, R)$$

covers the entire interval.

```
int rangeMinimum(const vector<vi>& table, int L, int R) {
  int maxLen = 31 - __builtin_clz(R - L);
  return min(table[maxLen][L], table[maxLen][R - (1 << maxLen)]);
}
```

While most functions either have inverses (so that we can use the prefix precomputation) or has idempotent (so that we can use the $\Theta(1)$ sparse table), some functions do not. In such cases (for example matrix multiplication), we must use the logarithmic querying of the sparse table.

Segment Trees

The most interesting range queries occur on *dynamic* sequences, where values can change.

Dynamic Interval Sum

Given a sequence of integers $a_0, a_1, \ldots, a_{N-1}$, you will be given Q queries. The queries are of two types:

1. Given an interval $[L, R)$, compute $S(L, R) = a_L + a_{L+1} + \cdots + a_{R-1}$.

2. Given an index i and an integer v, set $a_i := v$.

To solve the dynamic interval problem, we will use a similar approach as the general sparse table. Using a sparse table as-is for the dynamic version, we would need to update $\Theta(N)$ intervals, meaning the complexity would be $\Theta(\log N)$ for interval queries and $\Theta(N)$ for updates. It turns out the sparse table as we formulated it contains an unnecessary redundancy.

If we accept using $2 \log N$ intervals to cover each query instead of $\log N$, we can reduce memory usage (and precomputation time!) to $\Theta(N)$ instead of $\Theta(\log N)$. We will use the same decomposition as in merge sort (Section **??**). In Figure 13.4, you can see this decomposition, with an example of how a certain interval can be covered. In this context, the decomposition is called a *segment tree*.

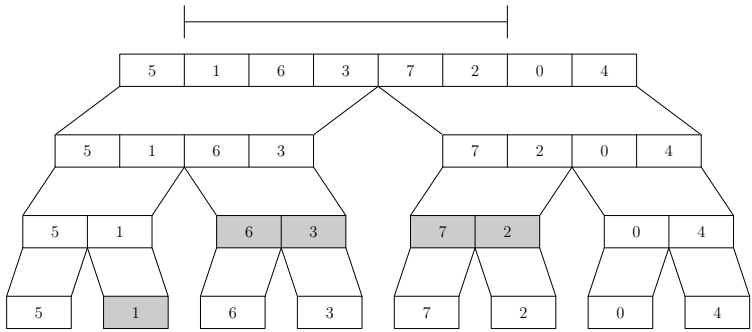


Figure 13.4: The $2N - 1$ intervals to precompute.

Usually, this construction is represented as a flat, 1-indexed array of length $2^{\lceil \log_2 N \rceil}$. The extraneous are set to some sentinel value that does not affect queries (i.e. 0 in the case of sum queries). From this point, we assume N to be a power of two, with the array padded by these sentinel values.

```
1: procedure MakeTree(sequence A)
2:     tree ← new int[2|N|]
3:     for i = |N| to 2|N| − 1 do
4:         tree[i] ← A[i − |N|]
5:     for i = |N| − 1 to 1 do
6:         tree[i] ← tree[2 · i] + tree[2 · i + 1]
7:     return P
```

In the construction, we label each interval $1, 2, 3, \ldots$ in order, meaning the entire interval will have index 1, the two halves indices 2, 3 and so on. This means that the two halves of the interval numbered i will have indices $2i$ and $2i + 1$, which explains the precomputation loop.

We can compute the sum of each of these intervals in $\Theta(1)$, assuming the sum of all the smaller intervals have already been computed, since each interval is composed by exactly two smaller intervals (except the length 1 leaves). The height of this tree is logarithmic in N.

Note that any particular element of the array is included in $\log N$ intervals – one for each size. This means that updating an element requires only $\log N$ intervals to be updated, which means the update time is $\Theta(\log N)$ instead of $\Theta(N)$ which was the case for sparse tables.

```
1: procedure UpdateTree(tree T, index i, value v)
2:     index ← i + N
3:     tree[index] ← v
4:     while index ≠ 0 do
5:         index ← index/2
6:         tree[index] ← tree[2 · index] + tree[2 · index + 1]
```

It is more difficult to construct an appropriate cover if the interval we are to compute the sum of. A recursive rule can be used. We start at the interval $[0, N)$. One of three cases must now apply:

- We are querying the entire interval $[0, N)$
- We are querying an interval that lies in either $[0, \frac{N}{2})$ or $[\frac{N}{2}, N)$
- We are querying an interval that lies in both $[0, \frac{N}{2})$ or $[\frac{N}{2}, N)$

In the first case, we are done (and respond with the sum of the current interval). In the second case, we perform a recursive call on the half of the interval that the query lies in. In the third case, we make the same recursive construction for both the left and the right interval.

Since there is a possibility we perform two recursive calls, we might think that the worst-case complexity of this query would be $\Theta(N)$ time. However, the calls that the third call results in will have a very specific form – they will always have one endpoint in common with the interval in the tree. In this case, the only time the recursion will branch is to one interval that is entirely contained in the query, and one that is not. The first call will not make any further calls. All in all, this means that there will me at most two branches of logarithmic height, so that queries are $O(\log N)$.

```
1: procedure QueryTree(tree T, index i, query [L, R), tree interval [L′, R′))
2:     if R ≤ L′ or L > R′ then
3:         return 0
4:     if L = L′ and R = R′ then
5:         return T[i]
6:     M = (L′ + R′)/2
```

```
7:  lsum = QueryTree(T, 2i, [L, min(R, M)), [L', M))
8:  rsum = QueryTree(T, 2i + 1, [max(L, M), R), [M, R))
9:  return lsum + rsum
```

13.3 Sliding Windows

Two Pointers

Monotone Queues

Additional Exercises

Exercise 13.14. Consider the union-find with neither path compression nor union-by-size. Instead, assign each element x a randomly chosen value $f(x)$, and merge the set with the higher value $f(repr[x])$ into the one with the lower value. Prove that this is expected $O(\log N)$ time.

Problem 13.15.

Loza	loza
Rings	rings
Almost Union-Find	almostunionfind
Peaks	peaks

Notes

Part III

Other Topics

Chapter 14

Graph Algorithms

It is time to extend our knowledge of graph algorithms further. That graph theory makes a return appearance is no coincidence: it's one of the richest topics in algorithmic problem solving and the one that takes the most space in the book. We already dealt with some of the basics back in Chapter 8, and the next chapter on *Flows and Matchings* is dedicated to just a specific sub-topic of graph algorithms. That there's three whole chapters – not to mention the many guest appearances in other chapters – on just graphs hopefully reinforces its importance.

14.1 Weighted Shortest Path

The theory of computing shortest paths in the case of weighted graphs is substantially richer than for the unweighted case. There are chiefly three algorithms that are used, depending on whether all edge weights are non-negative or not and if we seek shortest paths only from a single vertex or between all pairs of vertices. Non-negative weights is by far the most common case, so that's where we start. The algorithm typically used is called ***Dijkstra's algorithm***.

Single-Source Shortest Path, non-negative weights – shortestpath1

Given a graph where all edges have a non-negative weight, compute the shortest distances from a given source vertex s to all other vertices.

Solution. The approach is similar to the inductive BFS algorithm, where we iteratively compute the shortest distances to all other vertices ordered by distance. In the BFS this was simple: all neighbors of s has distance 1, their neighbors distance 2, and so on.

The same approach doesn't work as-is for the non-weighted case. The crucial difference is that even the shortest path to a vertex adjacent to s may first pass through another vertex (Figure 14.1). To get around this we need a key insight: there must be at least *some* neighbor of s where the shortest path is the direct edge from s. Specifically, the edge $\{s, v\}$ with the smallest edge weight (in Figure 14.1, that is the edge $\{s, d\}$) is always a shortest path to v. Any path to v must pass through one of the edges adjacent to s, and thus would be at least as long as the direct edge between s and v.

A generalization of this argument gives us Dijkstra's algorithm. Assume that you have found the distances of the n vertices closest to s, and call the set of these vertices C. We

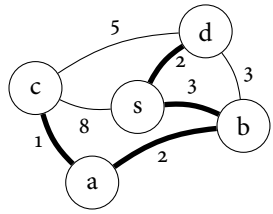
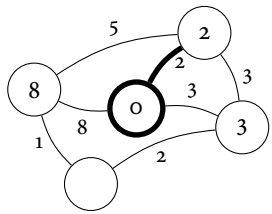


Figure 14.1: An example where the shortest path from s to the adjacent vertex c passes through several other vertices. Vertex d is the one closest to s.

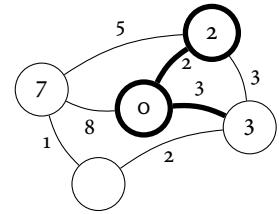
claim that the $(n+1)$'th closest vertex v has a neighbor u in C such that a shortest path to v is $s \to \cdots \to u \to v$.

For a proof, consider a shortest path to v. Let $\{u, v'\}$ be the *first* edge on this path where u is in C and v' is not, so that v' is a neighbor of C. Since v' is on the shortest path to v it holds that $d(s, v') \leq d(s, v)$ (Exercise 14.1). But v is the next closest vertex to s of those not in C, so we also have $d(s, v') \geq d(s, v)$ meaning that v and v' must have the same distance. Thus v' is also a $(n+1)$'th closest vertex, and it does indeed have a neighbor u in C such that its shortest path is $s \to \cdots \to u \to v$.

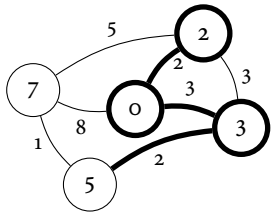
Exercise 14.1. Let $s \to v_1 \to \cdots \to v_n$ be a shortest path from s to v_n. Prove that $s \to v_1 \to \cdots \to v_i$ is a shortest path to v_i for all i.



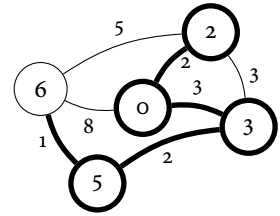
(a) The base case: the closest vertex to s is the neighbor with the shortest edge.



(b) The second closest vertex to now has distance 3, also a neighbor.



(c) Among the two remaining vertices, the one at distance 5 is closest to s.



(d) Finally, we find the distance of last vertex, which is 6.

Figure 14.2: An example of Dijkstra's algorithm finding the distances in a graph.

This insight results in a similar iterative algorithm, exemplified in Figure 14.2. Initially the set of vertices C of the i vertices closest to s consists only of s. We then repeatedly extend C with one additional vertex at a time, by looping through the neighbors of C and picking the one of these that are closest to s.

Formalizing the algorithm leads us to a $O(V^3)$ complexity and the following pseudo code:

```
1:  procedure DijkstraSlow(source vertex s, vertex set V)
2:      dists ← new int[|V|] filled with ∞
3:      dists[s] ← 0                          ▷ Base case: s is at distance 0 to itself
4:      C ← {s}
5:      while C ≠ V do                        ▷ V is as usual the set of vertices in the graph
6:          next ← −1
7:          dist ← ∞
8:          for each u ∈ C do
9:              for each edge (v, weight) of u where v ∉ C do
10:                 toDist ← dists[u] + weight
11:                 if toDist < dist then
12:                     next ← v
13:                     dist ← toDist
14:         dists[next] ← toDist
15:         add next to C
16:     return dists
```

What remains is an exercise in optimization. The first one brings us down to quadratic time. Instead of running the loop on lines 9-13 for every $c \in C$ each time, we can keep track of what the best distance is so far *for each vertex to* at all times, and only run the loop once a new vertex is added to C. In fact, we can reuse the *dists* array for this in the following way:

```
1:  procedure DijkstraQuadratic(source vertex s)
2:      dists ← new int[|V|] filled with ∞
3:      dists[s] ← 0
4:      while C ≠ V do
5:          v ← the vertex not in C with the lowest value
    of dists[v]
6:          Update(v)
7:          add v to C
8:      return dists
```

```
1:  procedure Update(vertex u)
2:      for each edge (v, weight) of u do
3:          toDist ← dists[u] + weight
4:          if toDist < dists[v] then
5:              dists[v] ← toDist
```

The way this is written is perhaps closer to how you would perform the algorithm by hand, scribbling the currently best distance next to each vertex, updating them as you add new vertices to C. Update is called exactly once for each vertex, so takes amortized time $\Theta(E)$. The main loop on lines 4-6 in DijkstraQuadratic takes $\Theta(V^2)$ time instead –

it makes V iterations, with a linear-time find operation on line 5. In total, this takes time $\Theta(V^2)$.

The final optimization is to speed up the linear step on line 5. Instead of finding the minimum distance among the remaining vertices by scanning through the array, you can store all of them in a priority queues instead, reducing the worst-case time complexity to $\Theta(E \log V)$ instead. This is typically coded using a `set<pair<int, int>>` in C++ rather than an actual priority queue. We show C++ code to demonstrate short way of implementing it:

```
1  const int INF = numeric_limits<int>::max();
2
3  // G is an adjacency list, storing edges as (destination, weight) pairs
4  vector<int> dijkstra(vector<vector<pair<int, int>>>& G, int s) {
5    vector<int> D(G.size(), INF);
6    D[s] = 0;
7    set<pair<int, int>> Q;
8    Q.emplace(0, s);
9    while (!Q.empty()) {
10     int next = Q.begin()->second;
11     Q.erase(Q.begin());
12     for (auto ed : G[next]) {
13       int v = ed.first;
14       int dist = D[next] + ed.second;
15       if (dist < D[v]) {
16         if (D[v] != INF) {
17           Q.erase(make_pair(D[v], v));
18         }
19         D[v] = dist;
20         Q.emplace(D[v], v);
21       }
22     }
23   }
24   return D;
25 }
```

In the C++ code we no longer represent the set C explicitly at all. Instead, Q contains the remaining vertices with a non-∞ distance and loops until there are none left. The other major change is some book-keeping to make sure that Q is actually kept up to date when the known best distances change. □

Exercise 14.2. Throughout the solution we have never mentioned anything about whether edges are directed or undirected. Does the solution assume that edges are undirected anywhere?

Some people prefer to implement Dijkstra's algorithm using real priority queues instead of sorted sets in C++. The downside to this is that it's not possible to update values in a C++ `priority_queue`, so you have to insert a new value if you find a shorter distance. The time complexity is the same, but the memory complexity is $O(E)$ rather than $O(V)$. Our experience is that this is sometimes worse by a non-trivial constant factor, and almost never better.

Problem 14.3.

Cross Country	crosscountry
Human Cannonball	humancannonball

Our first example deals with how a shortest path can be reconstructed using the distances computed.

Shopping Malls

By Jon Ander Gómez. Southwestern Europe Regional Contest 2013. CC BY. Shortened.

A shopping mall has $N \leq 200$ different sections, and $M \leq 1000$ different connections of four types: walking paths, stairs, lifts and escalators. We want to create a smartphone application to help visitors calculate the shortest path between some pairs of locations in the mall. Given a set of $Q \leq 1000$ pairs of points F, T, compute a path from F to T that requires the least walking.

The sections are located on different floors, each of which can be represented as a 2D plane. The distance between two adjacent floors is 5 meters. Each of the M connections connect two points A and B and can be used in either directions. When using a walking path or stairs, the distance a customer must walk equals the Euclidean distance between the points. A lift on the other hand only requires 1 meter of walking: 0.5 m when entering and exiting. Finally, an escalator only requires walking 1 meter when moving from A to B, but you can also use it to walk in the opposite direction of escalation. Then it instead requires walking a total of 3 times the Euclidean distance between B and A.

Solution. The sections and corrections together describe a weighted graph. Some care must be taken when adding the edges due to the complicated rules on how much walking is actually involved for each connection. After that it's a standard application of shortest path, where one invocation of Dijkstra's algorithm for each query results in the acceptable worst-case complexity $\Theta(QM \log N)$ (or even better $\Theta(NM \log N)$ by running Dijkstra once for each starting point).

The problem is not satisfied with only knowing the shortest distance for each query though, but wants us to reconstruct a shortest path. This is done almost in the exact same way as in the breadth-first search case. For each vertex v, we need to know what the previous vertex on its shortest path is, and then backtrack along these starting at the goal. To find the previous vertex, we must each time we find a new best distance to v also store what vertex was used to find this distance. While this might be updated several times, you know that it's the last time you find a better distance to v that determines the previous vertex on its shortest path. □

Problem 14.4.

Passing Secrets	passingsecrets
Block Crusher	blockcrusher
Robot Turtles	robotturtles

Just as in the undirected case, the graph you are supposed to find a shortest path in might not be given explicitly. Finding the reduction to a graph problem by figuring out what a vertex is supposed to represent and what edges is sometimes the goal of a problem.

Problem 14.5.
Full Tank fulltank

The set of edges used for path reconstruction forms a directed spanning tree called a ***shortest-path tree***. If the graph is undirected, we direct the edges away from s along the shortest paths. When a graph contains multiple shortest paths from a source there is a useful generalization of this tree, formed by all edges that lie on *any* shortest from s to another vertex in the graph. If there are no zero-length cycles (zero-length undirected edges in the undirected case), this subgraph forms a directed acyclic graph called the ***shortest-path DAG***.

Exercise 14.6. Let G be the graph containing only (directed) edges that lie on some shortest path from a vertex s to another vertex in the graph. Prove that any path from s in G is a shortest path.

Intercept – intercept

By Marc Vinyals. KTH Challenge 2014. CC BY-SA

The Stockholm subway network consists of $N \leq 100\,000$ stations. There are $M \leq 100\,000$ one-way lines between some pairs of stations, each of which take different times $0 < w_i \leq 10^9$ to travel. Fatima commutes from KTH to home by subway every day, between the stations closest to KTH and her home. Today Robert decided to surprise Fatima by baking cookies and bringing them to an intermediate station. Fatima does not always take the same route home, but she always optimizes her travel by taking the shortest route. List all possible stations Robert can go to in order to surely intercept Fatima.

Solution. Call the stations closest to KTH and her home s and t, respectively. No subway line takes 0 time to travel, so the graph containing all the shortest paths from s forms a DAG. In this problem we're not interested in all edges on a shortest path, only those on a shortest path to t. They also form a DAG since they're a subset of the shortest path DAG. To compute the shortest-$(s - t)$-path DAG we use the following fact.

Exercise 14.7. Let $d_s(x)$ be the distance from s to x, and let $d_t(x)$ be the distance from x from t. Prove that an edge $u \to v$ with weight w lies on a shortest path from s to t if and only if $d_s(t) = d_s(u) + w + d_t(v)$.

By this equality, it's enough to compute d_s and d_t to find the DAG. Running Dijkstra's algorithm from s gives us d_s, and once more from t but with all directed edges reversed gives d_t. The problem is now equivalent to finding those vertices that any $(s \to t)$-path must pass, a surprisingly difficult problem. Later on we learn how to these vertices with a single DFS, but instead we show another useful technique.

With dynamic programming it's easy to compute the number of paths p from s to t in the graph. Furthermore, we can compute the number of paths $p_s(v)$ from s to any vertex v, and the number of paths $p_t(v)$ from v to t. The number of $(s-t)$-paths passing v is $p_s(v) \cdot p_t(v)$, which is equal to p if and only if all paths pass through v. Unfortunately p can be exponential in N and easily overflows 64-bit integers. To get around this, compute p, p_s and p_t modulo a random large prime M (on the order of 2^{30} to avoid overflow issues). The probability that the equality holds modulo a large random prime even if the two sides aren't equal is very small (something like 1% over 100 test cases). □

Problem 14.8.

A Walk Through The Forest	walkforest
Customs Control	customscontrols

Sometimes Dijkstra's algorithm is the right idea, but modeling the problem in a small enough graph may require some ingenuity.

Trams – bybana

By David Wärn and Fredrik Ekholm. Swedish IOI Selection 2021.

The public transport of Stackville consists of a tram system with $N \leq 10^5$ stations and $M \leq N$ lines. A line is an ordered sequence of stations. A trip consists of boarding the tram at any one of those stations and exiting it at any other station. Together, the M lines make at most $3 \cdot 10^5$ stops.

To avoid that people take very short trips instead of walking, the municipal government has introduced a pricing system where the price of a trip is proportional to its *waste*. The waste of a trip is defined as the number of stations on the line that the trip does not pass. For example, if a line consists of the stations $(3, 7, 5, 2, 1, 9)$, a trip between stations 1 and 5 has a waste of three, since the stations 3, 7 and 9 were not passed.

Compute the minimum possible sum of the wastes when traveling from station 1 to N through a series of trips.

Solution. The obvious solution is to create a graph with an edge for each possible tram trip and run Dijkstra's algorithm on it. Unfortunately there can be on the order of 10^{10} such edges, far too many. There's a beautiful solution that's almost as short as the naive solution.

Exercise 14.9. Prove that there exists an optimal series of trips where each trip is to or from either the first or the last station on the tram line that is taken.

The authors didn't find this insight when first solving the problem. Our solution is a good example of how to model similar processes with graphs, so we show it as well.

Let a line consist of the stations $a_1, a_2, \ldots, a_m$, and consider a given trip from a_i to a_j where $a_i < a_j$ (the case $a_j < a_i$ is handled similarly). The waste then equals the number of stations to the left of i which is $i-1$, plus the number to the right of j which is $m-j$. These numbers are independent of each other, so we can model the cost of the trip as paying two different fees: $i-1$ when boarding the tram at station i, and $m-j$ when exiting it at

station j. Since these costs are associated with a linear number of origins and destinations rather than a quadratic number of possible trips, we hope that fewer edges are needed.

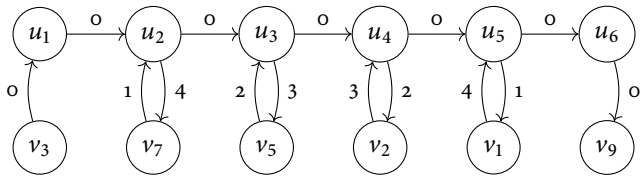


Figure 14.3: An example encoding the line with stations $(3, 7, 5, 2, 1, 9)$

To encode all this in a graph, first introduce a vertex v_i representing standing at station i. For each line l, add vertices $u_{l,s}$ representing traveling on the l'th tram from first to last station, currently being at its s'th stop. Three kinds of edges then needs to be added for each $u_{l,s}$: one edge with the cost to board the line, one edge with the cost to leave the line, and then one edge with the cost to travel on the line to the next station (see Figure 14.3). □

Problem 14.10.
Moving Walkway rullbandet

Finally, we end our treatment with a small modification of Dijkstra's algorithm. This shows you that the algorithm is not only capable of computing shortest paths in a graph, but is rather a more general framework for certain kinds of inductive computations. The difference is often a change in how "distance" is defined.

Crocodile

By Mihai Pătrașcu. International Olympiad in Informatics 2011.

Archaeologist Benjamas is trying to escape the mysterious Crocodile's Underground City. The city has $N \leq 100\,000$ chambers and $M \leq 1\,000\,000$ bidirectional corridors, each connecting a pair of chambers, the i'th of which takes t_i time to run through. Some of the chambers are exits that allow her to escape.

Benjamas starts in chamber 0 and wants to reach an exit chamber as quickly as possible. The Crocodile gatekeeper wants to prevent Benjamas from escaping by controlling secret doors that can block a single corridor at a time. More specifically, when Benjamas enters a chamber, the Crocodile can choose at most one of the corridors adjacent to the chamber and block it. Benjamas can then leave the chamber through any of the adjacent corridors except the blocked one. When Benjamas enters the next chamber, the Crocodile may again choose the block one of the outgoing corridors (possibly the corridor that Benjamas just followed), whereupon the previously blocked corridor is opened.

Help Benjamas construct an exit plan that minimizes the maximum time the Crocodile can force Benjamas to remain in the city. An exit plan consists of an instruction for each chamber that's

not an exit chamber, that tells Benjamas through which corridor she should exit, or if that corridor is blocked, which through corridor she should exit instead. It is guaranteed that such an exit plan exists.

Solution. Let's first try to characterize the best possible escape time from a given vertex. If you could, you would always try to go through the corridor that gives the quickest escape. Symmetrically, that's the corridor that the Crocodile would block. More formally, if a chamber have adjacent corridors that take $t_1, t_2, \ldots$ seconds to run through, and they lead to corridors where you can escape in at most $p_1, p_2, \ldots$ seconds, the Crocodile will block the corridor that minimizes $t_i + p_i$, so you will pick the second best option. This leads to a new definition of "distance" from a chamber v to an exit, that we must compute.

To see if this can be attacked with a variation of Dijkstra's algorithm, we should try to formulate a similar inductive way of repeatedly determining the escape time for vertex at a time. Initially, we know the escape time only for the chambers that are exits: it is 0 seconds. Assume that the escape times of the K chambers with shortest escape times are known. What must hold for the chamber v with the $(K+1)$'st shortest escape time? Can it be the case that the two corridors from it resulting in the best escape times lead to other chambers than one of those K? By the definition of the best escape time for a chamber, the escape time of v is always greater than that of its two best adjacent chambers, and since v has the $(K+1)$'st shortest escape time, those two chambers must be among those with the k shortest times. This gives us a way to find the chamber v. Among all chambers adjacent to at least two of the K chambers with the shortest escape times, take the one that minimizes the second-best escape time to one of those chambers.

As with the plain Dijkstra's algorithm, this immediately translates into a cubic algorithm. To get the worst-case time down to $\Theta(M \log N)$ you then apply the same optimizations. Note that it's the second-best escape time that the chambers in the queue should be sorted by, so you need to always keep track of the two corridors with the best and the second-best escape times for each chamber. □

Among all shortest path problems, those where you have to modify Dijkstra's algorithm are among the most common and varied ones, so we give unusually many exercises on the theme.

Problem 14.11.

Hopscotch 50	hopscotch50
Single source shortest path, time table	shortestpath2
Millionare Madness	millionairemadness
Emptying the Baltic	emptyingbaltic
Tide	tide

Negative weights

When edges can have negative weights, the idea behind Dijkstra's algorithm no longer works. It is very much possible that a negative weight edge somewhere in the graph could result in a shorter path back to a vertex that Dijkstra's algorithm has already determined the distance to. An alternative inductive idea still works, but results in a slower $\Theta(VE)$ algorithm.

Single-Source Shortest Path, negative weights – shortestpath3

Given a graph where all edges have a (possibly negative) weight, compute the shortest distances from a given source vertex s to all other vertices.

Solution. We need to draw some help from one of our old tools, namely dynamic programming. As is often the case in DP, the right idea is to compute something stronger than what the problems asks for. For shortest paths, the right problem to solve is instead: what's the shortest distance to a given vertex v that uses *at most* k edges? Since a shortest path includes each vertex in the graph at most once, choosing $k = |V| - 1$ is equivalent to what we're actually looking for. One might come to think about this stronger version by looking at the BFS, which in a very similar spirit iteratively finds all vertices at most k edges away from the source.

The DP then recurses on the number of edges used. To get to a vertex v using at most k edges, you must first get to one a vertex with an edge pointing at v using at most $k-1$ edges, and then traverse that final edge. Letting $D(k, v)$ denote the distance to v using at most k edges, the right recursion is

$$D(k,v) = \min \begin{cases} 0 & \text{if } v = s \\ D(k-1,v) & \text{if } k > 0 \\ \min_{e=(u,v)\in E} D(k-1,u) + w(e) & \text{if } k > 0 \end{cases}$$

The bottom-up implementation of this is wonderfully short. By evaluating the recursion for all v at the same time for a given k, it's enough to iterate through the entire edge list to find the right values.

```
1: procedure BellmanFord(vertices V, edges E, vertex s)
2:     D ← new int[|V|][|V|] filled with ∞
3:     D[0][s] ← 0                                  ▷ The base case of the recursion
4:     for k = 1 to |V| − 1 do
5:         D[k] = D[k − 1]                          ▷ The second case of the recursion
6:         for e = (u, v) ∈ E do
7:             D[k][v] = min(D[k][v], D[k − 1][u] + W(e))
8:     return D[|V| − 1]
```

These two nested loops together take $\Theta(VE)$ time.

In practice, Bellman-Ford is implemented in a shorter way. Instead of storing all the values of $D(k, v)$, we keep only a single value $D(v)$ that is always a lower bound of $D(k, v)$ and maintain that it is so after each iteration of the loop on line 4. As a bonus we get away with $\Theta(V)$ memory instead of $\Theta(V^2)$, a win for sparse graphs.

```
1: procedure BellmanFord(vertices V, edges E, vertex s)
2:     D ← new int[|V|] filled with ∞
3:     D[s] ← 0
4:     for k = 1 to |V| − 1 do
5:         for e = (u, v) ∈ E do
6:             D[v] = min(D[v], D[u] + W(e))
7:     return D
```

We mentioned that Bellman-Ford doesn't work if there are negative length cycles. In concrete terms, this means that the distances doesn't converge, since you can attain arbitrarily short distances with arbitrarily short walks. This takes $O(E)$ time by the following exercise:

Exercise 14.12. Prove that a vertex has an arbitrarily short distance if and only if it's reachable from a vertex v where $D(|V| - 1, v) > D(|V|, v)$.

The above holds true even for the simplified version where $D(v)$ is only a lower bound on $D(k, v)$ after each loop iteration. □

The most interesting part about Bellman-Ford is working with negative length cycles, so that's the topic of our example problem.

Wormholes – wormholes2

By Jaap Eldering, Gerben Stavenga, and Jan Kuipers. NWERC 2009. CC BY-SA. Shortened

A friend of yours has built a spaceship recently and wants to explore space. It takes t seconds to travel between two points with Euclidean distance t. wormholes2 During his first voyages, he discovered that the universe contains $N \leq 50$ wormholes created by aliens that transport the ship through time and space. Each wormhole connects two distinct points in space and has a given time shift $-10^6 \leq d_i \leq 10^6$ when traveling through it. They are also created by the aliens at different times $-10^6 \leq t_i \leq 10^6$ at which point it becomes available for travel. Given the current position of your ship at time 0 and a given position that you and your friend wants to explore, compute the earliest possible arrival time.

Solution. On the surface a plain shortest path problem where wormholes are edges between points in space, and the time shift of an edge is its weight. A small modification needs to be made to whatever shortest path algorithm used to add some extra weight when trying to use a wormhole before it is created.

Unfortunately the negative time shifts discard Dijkstra as an option, and Bellman-Ford fails due to negative length cycles, or at least it would in the general case. In the given

task each edge has an earliest time at which it can be traversed, so one can never arrive at a point at an arbitrarily early time – Bellman-Ford would eventually converge to the right answer. This can take a long time however, since the shortest walk can contain on the order of $N \cdot 10^6$ edges! While running Bellman-Ford a bunch of extra times would be correct, it's simply too slow. To fix it, we need some clever optimization.

Once you are able to traverse a wormhole exactly at the time of its creation, that wormhole can never again be used to arrive even earlier to its destination. Consequently, if there is a negative length cycle in the graph (by necessity including a wormhole), we could travel around that cycle repeatedly until we reach a wormhole before it's created, and then remove it. Since there are at most N wormholes, we would only need to identify a negative length cycle and simulate traveling along it at most N times before eliminating all negative length cycles.

Simulating traveling around a negative length cycle until it breaks is a fun implementation task, but identifying one is slightly harder – until now we've only described how to find the set of vertices reachable from a negative length cycle. The trick is to reuse the path reconstruction mechanism from Dijkstra's algorithm, i.e. for each vertex keeping track of the edge last used to improve the distance to it. Normally this produces a shortest-path tree, but if a negative length cycle exists it will appear in this set of edges instead after running Bellman-Ford plus the extra iteration needed to ensure negative cycles are detected.

Exercise 14.13. Prove that:

1. a cycle in the path reconstruction graph is a negative length cycle in the original graph.
2. if a graph contains a negative length cycle, the path reconstruction graph must have a cycle.

Finding a cycle in a graph can be done in $O(E)$ time with a single DFS, so the total complexity is dominated by the N Bellman-Ford iterations taking $O(V^2E) = O(V^4)$ time. □

Problem 14.14.

Dangerous Skiing	dangerousskiing
XYZZY	xyzzy

All-Pairs Shortest Paths

For a dense graph, the $\Theta(V^2)$ version of Dijkstra beats the otherwise faster $\Theta(E \log V)$ version. Consequently it never takes more than $\Theta(V^3)$ time to find the distance between *every* pair of vertices, and for sparse graphs $\Theta(VE \log V)$ is considerably faster. When edges can have negative weights Dijkstra no longer works, but a remarkable algorithm lets us keep the $\Theta(V^3)$ complexity.

All-Pairs Shortest Paths – allpairspath

Given a graph where all edges have a (possibly negative) weight, compute the shortest distance between each pair of vertices.

Solution. As usual, assume that all vertices are numbered $0, 1, \ldots, |V| - 1$. Say that you want to compute the shortest path from v to u, and that this path happens to be $v \to a_1 \to \cdots \to a_k \to u$. Take a_i to be the highest-numbered vertex on the path. The shortest path can now be split up into two smaller shortest paths: one from $v \to a_i$ and one from $a_i \to u$. In both of these smaller paths, any intermediate vertices have lower numbers than a_i. Thus it would be enough to first compute all paths where the intermediate vertices are less than a_i. The same thing applies for any such path $s \to b_1 \to \cdots \to b_k \to t$ too, except this time the highest b_j is smaller than the highest a_i. This breaking down can't continue forever though. At some point the smaller paths will be single edges $x \to y$, which serves as the base cases.

If we let $D(i, j, k)$ be the distance from i to j where all intermediate vertices at between 0 and k, this results in the recursion

$$D(i,j,k) = \min \begin{cases} w & \text{if } i \to j \text{ is an edge of weight w} \\ D(i,j,k-1) & \\ D(i,k,k-1) + D(k,j,k-1) & \end{cases}.$$

In the second case the vertex k doesn't lie on the path. In the third it does, whereupon we recursively find the two smaller paths as described.

The algorithm is never written in a top-down manner. Bottom-up, it's very compact:

```
1: procedure Floyd-Warshall
2:     D ← new int[|V|][|V|][|V|] filled with ∞
3:     D[0] ← the weighted adjacency matrix of the graph        ▷ This handles the base cases
4:     for k = 0 to |V| − 1 do
5:         for i = 0 to |V| − 1 do
6:             for j = 0 to |V| − 1 do
7:                 D[k][i][j] = min(D[k − 1][i][j], D[k − 1][i][k] + D[k − 1][k][j])
8:     return D[|V| − 1]
```

The same shortening trick that we used for Bellman-Ford works here too, i.e. that it's enough to store a single distance matrix $D(i, j)$ and make sure that $D(i, j) \leq D(i, j, k)$ as we loop over k. The shortened version is what's known as the ***Floyd-Warshall*** algorithm and takes $\Theta(V^3)$ time.

```
1: procedure Floyd-Warshall
2:     D ← the weighted adjacency matrix of the graph
3:     for k = 0 to |V| − 1 do
4:         for i = 0 to |V| − 1 do
```

```
5:          for j = 0 to |V| − 1 do
6:              D[i][j] = min(D[i][j], D[i][k] + D[k][j])
7:      return D
```

Finding pairs of vertices at an arbitrarily short distance is much easier than for Floyd-Warshall.

Exercise 14.15. Prove that

1. if v lies on a negative length cycle, then $D(v,v) < 0$.
2. if $D(v,v) < 0$, v lies on a negative closed walk.

The DFS step needed by Bellman-Ford is unnecessary since checking if v can reach u is equivalent to $D(v,u) \neq \infty$. □

Exercise 14.16. Adapt Floyd-Warshall to allow for reconstructing paths.

Transportation Planning – transportationplanning

By Greg Hamerly. Baylor Competitive Learning course. CC BY-SA

As a city planner, you have spent a lot of time building a set of two-way roads so that people can get from anywhere in the city to anywhere else just by driving on the roads. Now that that's done, you are concerned with how much time commuters spend on the road each day, and how you can reduce this. You have modeled the city as a set of $N \leq 100$ intersections connected by $M \leq \frac{N(N-1)}{2}$ roads. You have come up with a measure of the total commute time: it is the sum of the shortest driving time between all pairs of intersections in the city using the available roads. You have gotten the city to agree to build one more road between two existing intersections, and you want to add a single road that will reduce this measure of commute time the most. Which road should you add?

Solution. Thanks to Floyd-Warshall we can in $\Theta(N^3)$ time find the original commuter times between all pairs of intersections. If the distance between two intersections u to v decreases after an edge $\{a,b\}$ is added, it must be because that edge is used on an even shorter path. There are only two possible distances including this edge: $d(u,a) + w(\{a,b\}) + d(b,v)$ or $d(u,b) + w(\{a,b\}) + d(a,v)$. Thus it's easy to find the improvement for any pair u, v after adding a single edge in $\Theta(1)$, giving a total time complexity of $\Theta(N^4)$. □

The problem raises a general question: how do you maintain the Floyd-Warshall distance matrix when additional edges are added? In general, when an extra edge $\{a,b\}$ is added, any new shortest paths in the graph has either a or b is an intermediate vertex, except possibly for the path between a and b itself. Handling this is the whole point of the Floyd-Warshall DP, so it's enough to run two more iterations where $k = a, b$ respectively.

Problem 14.17.

Kastenlauf	kastenlauf
Transportation Planning	transportationplanning
Secret Chamber at Mount Rushmore	secretchamber

In most problems that requires solving all-pairs shortest path the $\Theta(V^3)$ complexity of Floyd-Warshall is fine. Only when the graph has no negative weight edges, is sparse, and has more than a few hundred vertices is Dijkstra's algorithm the right way.

14.2 Eulerian Walks

In Chapter 8, you had the chance to solve some exercises on the topic of *Eulerian Walks*. At that point we only asked you to verify whether a walk was Eulerian or not. It's now time to learn how to construct them.

Eulerian Walk – eulerianpath

Given a directed graph, find a ***walk*** – a path on which vertices don't have to be distinct – starting and ending at arbitrary vertices (possibly the same) that uses each edge in the graph exactly once.

Solution. There are two necessary conditions for the existence of an Eulerian walk. First of all, the graph must be weakly connected, meaning that it would be connected if all edges were undirected. Secondly, since each time the walk enters a given vertex it must also exit that same vertex, the in-degree of a vertex must equal its out-degree. At least, this must be true except if the walk starts and ends in different vertices. Then the first vertex must have an out-degree 1 more than its in-degree and the last an out-degree 1 less. As luck would have it, these simple criteria are also sufficient.

We focus on the case where the walk starts and ends at the same vertex, the other case is similar. The basis of the solution is to pick a vertex and arbitrarily walk along a previously unused edge for as long as possible.

Exercise 14.18. An arbitrary walk starting at a vertex v eventually arrives at a vertex without edges to follow. Prove that this is v.

Let this path be $p_1 \to p_2 \to \dots p_n \to p_1$. If we were lucky, this path might have included each edge in the graph. In the other case, the following must hold.

Exercise 14.19. Prove that if there are any unused edges left, at least one of them is outgoing from one of the vertices p_i.

Let p_i be one of these vertices with an outgoing unused edges. If you from p_i perform the exact same kind of walk, you will again get some walk $p_i \to q_1 \to \dots \to q_m \to p_i$. This extra walk can then be spliced into the original one, so that it goes

$$\dots \to p_{i-1} \to p_i \to q_1 \to \dots \to q_m \to p_i \to p_{i+1} \to \dots$$

These two steps can be repeated until there are no edges left, at which we have produced an Eulerian walk.

Time complexity is then a matter of implementation. Naively this is quadratic in the number of edges, but performing these walks in a smart manner similar to a DFS lets us find the walk in linear time.

```
1: procedure EulerWalk(vertex v)
2:     for each unused edge v → u do
3:         mark v → u as used
4:         EulerWalk(u)
5:     prepend v to the walk
```

Effectively, this recursion first finds the tour p and then backtracks up to the *last* p_i that still has an unused edge. There, it repeats the procedure by splicing in the new tour q recursively, before then backtracking further back in the path p.

Exercise 14.20. Prove that for the case where one vertex s has an extra out-degree and another vertex t has an extra in-degree, the algorithm works as long as the walk starts from s.

Exercise 14.21. Prove that the same algorithm works in the undirected case, but that the degree condition is that all vertices must have even degrees, except that exactly two vertices are allowed to have an odd degree.

A small detail makes the implementation a bit complex. While iterating through all neighbors of v, it's possible that the walk will again visit v and try to iterate through all neighbors. To avoid conflicts between multiple calls with the same v, use adjacency lists and pop an edge at the time from the back of the list:

```
while (!edges[v].empty()) {
        int u = edges[v].back();
        edges[v].pop_back();
        EulerWalk(u);
    }
```

The recursive procedure is sometimes popularly written in an iterative manner using a stack, but our experience is that at least in C++ this makes little difference. □

Problem 14.22. *Acrobat* acrobat

The algorithm we described is called ***Hierholzer's algorithm***. The other well-known algorithm is called ***Fleury's algorithm***. Its original description leads to an inefficient algorithm, but the following modified version runs in linear time.

Exercise 14.23. Let G be a weakly connected directed graph where each vertex has equal in-degree and out-degree. Pick a vertex v and find a directed spanning tree where all edges point towards v. Perform a walk in the graph, never using an edge twice, and using an

edge in the spanning tree only if it is the last out-edge from a vertex. Prove that this walk is Eulerian.

Many problems use the general insights that Hierholzer's algorithm is based on, rather than explicitly looking for Eulerian walks.

Fair Secret Santa

A competitive programming club with $N \leq 100\,000$ members is hosting a secret Santa gift giving event. Among the members $M \leq 300\,000$ pairs have been selected, each in which one of the members should to buy a gift for the other. For the sake of fairness, it has been decided that everyone should give and receive the same number of gifts. This is not possible if a person is in an odd number of pairs. They may then receive either one more or one less gift than the give. Decide for each pair of members which of them should give a gift to the other.

Solution. The problem can be formulated as asking for the edges in the graph to be directed, where a directed edge $a \to b$ represents a giving a gift to b. Each vertex should then have an equal in-degree and out-degree (for an even degree) or they must differ by at most one (for an odd degree).

Odd degree vertices can be iteratively eliminated from the problem two at a time. Pick any odd-degree vertex and perform a walk using each edge at most once. This walk is guaranteed to end up at another edge by the same arguments as for Euler walks. By directing the edges on that walk in the direction of traversal, an intermediate person on the path will give and receive one additional gift, while the endpoints either gives or receives one extra gift. After removing this path the first and last vertex have an even number of edges too. Once all vertices have an even degree it's easy to find a direction of all edges such that the in-degree and out-degree are equal: find an Euler cycle for each connected component. □

For extra credit, note that we didn't need to find Euler cycles specifically, a decomposition into multiple cyclic walks is fine too. In fact a graph containing an Eulerian cycle can always be partitioned into *simple* cycles as well as required, with a similar algorithm.

Problem 14.24.

Railroad	railroad2
Catenyms	catenyms

14.3 The Depth-First Search

We first studied the depth-first search in Chapter 8 together with the breadth-first search. So far it's been used mainly for two purposes: as a shorter version of the BFS when we're traversing a graph without regard to order, and as a way of traversing a rooted tree in a way such that all descendants of a vertex are processed before the vertex itself. In this section we show more powerful applications, solving a few standard problems centered around

various notions of connectivity. These applications are slightly different depending on whether the graph is directed or not, so we start with the simpler, undirected case. They are based on the fact that the DFS generates a special type of spanning tree in graphs.

Normal Spanning Tree – normalspanningtree

In a graph G, let T be a spanning tree that has been rooted in some vertex. If, for each edge $\{u, v\}$ in G, either u is an ancestor of v or v is an ancestor of u in the rooted tree T, we call T a ***normal spanning tree***.

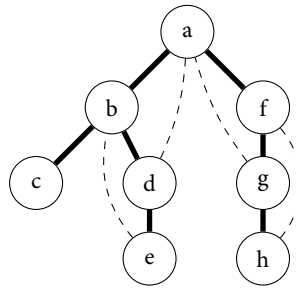


Figure 14.4: In the above graph, a normal spanning tree is marked by the bold edges.

Given a connected graph on $N \leq 100\,000$ vertices and $M \leq 300\,000$ edges, find a normal spanning tree.

Solution. As is often the right approach for tree problem, we look for some kind of recursive construction. Studying the normal spanning tree in Figure 14.4, we might notice that, ignoring the non-tree edges going up to a, the subtrees rooted in b and f are themselves normal. In fact, any subtree is normal (except that there might be edges going to an ancestor of the subtree's root).

Exercise 14.25. Prove that any subtree of a normal tree is also normal.

More interestingly, this is also a sufficient condition: if the subtrees of the root in the tree are normal, the whole tree is so too. Each edge either lies strictly within a normal subtree and thus by definition goes between a vertex and its ancestor, or it goes up to the root which is always an ancestor to the other end of the edge.

We can now give a constructive proof by induction that any graph contains a normal spanning tree rooted in **any** vertex. Consider an n-vertex graph, and assume that any graph up to size $n-1$ vertices have a normal spanning tree. Pick an arbitrary vertex v to be the root of the spanning tree, and consider the connected components that would form if v was removed from the graph. For each component, find a normal spanning tree in it with a root that was a neighbor of v in the original graph. By connecting the root of each of these smaller trees with v, we get a normal spanning tree of the entire graph.

As described this method runs in quadratic time. Splitting the graph into components takes linear time, and we might have to do this a linear number of times (for example if the graph is a single long path).

```
1: procedure MakeTree(vertices V, vertex root)
2:     for each component C of V ∖ {root} do
3:         find a neighbor u of root in C
4:         add (root, u) to the tree
5:         MakeTree(C, u)
```

To speed this up, keep track of the partitioning into components implicitly instead. When picking the first component C and root u, clearly any neighbor of *root* can be chosen. After constructing the tree rooted at u, all vertices in the component to which u belongs have been added to the tree, so to find a vertex in a different component any neighbor of *root* that hasn't get been added to the tree can be chosen, and so on, leading to the following simplification:

```
1: procedure MakeTree(vertex root)
2:     for each neighbor u of root do
3:         if u has not been added to the tree then
4:             add (root, u) to the tree
5:             MakeTree(u)
```

You should recognize this pseudo code – it's exactly a depth-first search. □

This property of the DFS is usually explained the other way around, i.e. by showing that the DFS tree is normal. We prove this too, to gain further understanding of the DFS tree. Assume that $\{u, v\}$ is an edge in the graph, and that without loss of generality u is visited before v during a DFS. Since v is a neighbor of u, we know that v must be visited before we have finished processing u. When that happens, there is some path $u \to x_1 \to \cdots \to x_k \to v$ the DFS took in order to get to v (it might be that $k = 0$ and the path is $u \to v$), which is part of the DFS tree. That means the DFS tree contains a directed path from u to v, so that u is indeed an ancestor of v. Finally, not only are DFS trees normal, but any normal spanning tree can be found by a DFS that visits vertices in some specific order:

Exercise 14.26. Prove that every normal spanning tree is also a DFS tree.

Problems that specifically require a DFS, i.e. are not just based around finding a spanning tree in general, in one way or another use that the tree is normal.

Joint Excavation – jointexcavation

By Timon Knigge. NWERC 2020. CC-BY SA.

A mole family recently decided to dig a new tunnel network. The layout consists of $N \leq 2 \cdot 10^5$ chambers and $M \leq 2 \cdot 10^5$ bidirectional tunnels connecting them, forming a connected graph. Mother mole wants to use the opportunity to teach her two mole kids how to dig a tunnel network.

As an initial demonstration, mother mole is going to start by digging out a few of the chambers and tunnels, in the form of a non-self-intersecting path in the planned tunnel network. She will

then divide the remaining chambers between the two kids, making sure that each kid has to dig out the same number of chambers.

Since the kids do not have much experience with digging tunnel networks, mother mole realizes one issue with her plan: if there is a tunnel between a pair of chambers that are assigned to different kids, there is a risk of an accident if the kids happen to excavate the tunnel from their respective chambers at the same time.

Help mother mole decide which path to use for her initial demonstration, and how to divide the remaining chambers evenly, so that no tunnel connects a pair of chambers assigned to different mole kids. The initial path must consist of at least one chamber and must not visit a chamber more than once.

Solution. The hardest part of the problem might be to even consider that a DFS is the right solution. Now that we have learned about the connection between the DFS and normal trees, there is a small hint: we are seeking a partition where no edges go between the two parts. That is similar to the key property of a normal spanning tree, that edges don't go between the subtrees of the root.

We formulate a solution in terms of the quadratic-time method used to construct a normal spanning tree. Pick an arbitrary vertex where mother mole starts her path and partition the remaining graph into its connected components. Order these components arbitrarily and give as large of a prefix of components as possible without exceeding $\frac{N-1}{2}$ chambers to the first kid, and a corresponding suffix to the second kid. Either all chambers were evenly split, or there is one component remaining. We let the mother's path continue into that component, and do the same thing: splitting the remaining vertices up into components and giving as large of a prefix and suffix to the kids as possible, without any one of them exceeding $\frac{N-2}{2}$ assigned chambers, including those given in the previous step.

The process continues precisely as long as there is at least one unassigned chamber left, but must be finite. Assume that it stops after the mother has dug a path through K chambers. By construction, no kid ever has more than $\frac{N-K}{2}$ assigned chambers – if at any point one kid was assigned exactly $\frac{N-K}{2}$ chambers, the other kid could be given all the remaining chambers for an even split. Since exactly $N - K$ chambers aren't on the mother's path, if all of them are assigned each kid must have gotten exactly $\frac{N-K}{2}$.

There is also a more direct solution, is almost the analogue of moving to the linear-time construction of a normal spanning tree. During a DFS, each vertex can be classified into as having been visited and backtracked from, having been visited but not yet back-tracked from, and not yet having been visited. In a DFS, there is never an edge between a vertex that has been visited and backtracked from, and a vertex not yet visited – otherwise we wouldn't have backtracked from that vertex yet. Initially, there are 0 vertices of the first kind and N vertices of the third kind, while in the end, there are N vertices of the first kind and 0 of the second. Each step during the DFS either increases the number of backtracked from vertices by 1, or decreases the number of unvisited vertices by 1. As such, these two numbers must at some point be equal, providing us with a valid partition. □

Problem 14.27.

Biconnected Components

The DFS tree is very useful in a large number of graph connectivity problems thanks to the fact that different subtrees are only connected by their common ancestors.

> **Definition 14.1 — Biconnectivity**
> Consider a connected, undirected graph. If removing a vertex v (and its adjacent edges) results in a disconnected graph, the vertex is a ***cutvertex***. If the graph lacks cutvertices, it is ***biconnected***. The maximal vertex subsets that are biconnected are the graph's ***biconnected components***. If removing an edge $\{u, v\}$ disconnects the graph, the edge is a ***bridge***.

Most textbooks define a biconnected graph slightly differently, in that they require the graph to have at least 3 vertices. We deviate in this chapter from the normal definition since it lets us avoid treating bridges as exceptions in a few theorems on biconnectivity.

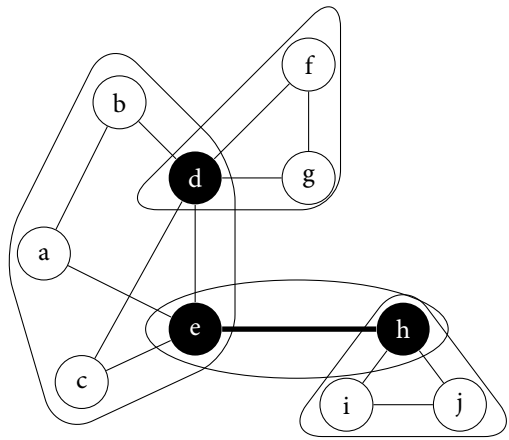


Figure 14.5: A graph with 4 biconnected components (encircled), 3 cutvertices (black vertices), and one bridge (bold edge).

As Figure 14.5 shows, a vertex can be part of multiple biconnected components. This occasionally makes it a bit messy to look at biconnected components as sets of vertices. The following fact enables us to view the components as sets of edges instead.

Exercise 14.28. Prove that two biconnected components have at most one vertex in common.

Consequently, no two biconnected components can both contain the ends of an edge $\{u, v\}$. However, there is always *some* biconnected component that contains both u and v. Since $\{u, v\}$ itself a biconnected subset it is also part of some maximal biconnected subset. Combining these two facts, we see that the biconnected components are a partition of the edge set.

Biconnected components, and the DFS in general, are in a sense about describing interconnectivity between the cycles of a graph. The most important fact in understanding them is the following.

Exercise 14.29. Prove that all edges in a cycle graph belong to the same biconnected component.

As you might have spotted from Figure 14.5, bridges and cutvertices can be described in terms of the biconnected components of a graph.

Exercise 14.30. Prove that

1. an edge is a bridge if and only if it is the only edge in a biconnected component.
2. a vertex is a cutvertex if and only if it's shared by two biconnected components.

Biconnected Components – biconnectedcomponents

Decompose an undirected, connected graph with $V \leq 100\,000$ and $E \leq 300\,000$ edges into its biconnected components.

Solution. Finding biconnected components is surprisingly straightforward. They fall out as part of a single DFS with some extra bookkeeping.

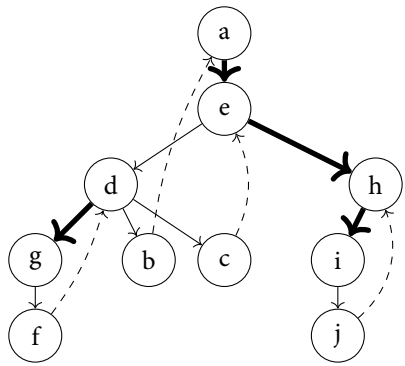


Figure 14.6

In Figure 14.6 one possible DFS tree of the graph in Figure 14.5 is shown, having started the DFS in a. Four edges of the DFS tree are marked in bold, defined as those edges $u \to v$ for which there is no back edge from v's subtree that goes further up in the tree than u.

We now claim two things: each such edge $u \to v$ is in a different biconnected component than any of the other edges adjacent to u, and every other tree edge lies in the same biconnected component that the immediately preceding tree edge lies in. In other words, the biconnected components are found by starting at bold edge and traversing the tree edges downwards, stopping at any other of the bold edges. The biconnected components in our example are thus $\{f, g, d\}$, $\{b, c, d, e, a\}$, $\{j, i, h\}$ and $\{h, e\}$, matching what we knew beforehand.

For the first claim, let $u \to v$ be such an edge and assume that one of u's neighbors w is in the same biconnected component as u and v. If we remove u, v and w are still connected, so there is some path starting in v that ends at w. Since the DFS tree is a normal tree, the only edges that aren't between two vertices within v's subtree are back edges that point to one of v's ancestors. As w is not in v's subtree, the path to w contains at least one such edge. We've removed u though, so that back edge must also be a to an ancestor of u, but by how we chose $u \to v$ no such backedge exists, contradicting that w is part of the biconnected component.

To prove the second part we use Exercise 14.29. Consider two edges that immediately follow in the tree, $p \to u$ and $u \to v$ such that there *is* a back edge from v's subtree to an ancestor of v, we can construct a cycle by going $p \to u \to v$, then continuing down to the back edge, moving along it to v's ancestor, and finally going down in the tree to p. Since both edges were on a cycle, they belong to the same biconnected component.

To find the bold edges in linear time, keep track of the current depth in the tree and compute how far up in the tree each subtree has a back edge during a DFS. To avoid an extra DFS to also find the components, you typically push all edges (including back edges) you see onto a stack, and once you backtrack from a bold edge, pop everything up to that edge into a new biconnected component.

```
1:  D ← new int[]
2:  procedure BiconnectedDFS(v, d)
3:      D[v] ← d
4:      mind_v ← d
5:      for each non-parent neighbor u of v do
6:          if u is unvisited or D[u] < d then
7:              push {v, u} to the stack
8:          if u is unvisited then
9:              mind_u ← BiconnectedDFS(u, d + 1)
10:             if mind_u ≥ d then
11:                 pop all edges up to and including {v, u} as a new component
12:             mind_v ← min(mind_v, mind_u)
13:         mind_v ← min(mind_v, D[u])
        return mind_v
```

It's easy to miss the second condition on line 6, which guarantees that the back edges are

only added in the correct direction (from the descendant up to the ancestor). □

Problem 14.31.

Cave Exploration	caveexploration
The Elk	elk

Most problems on biconnectivity are straightforward after identifying the components, cutvertices and bridges. There are problems that require at least some further understanding of the concept however, not seldom being based around the connectivity properties of the DFS tree.

Directing Streets – directingstreets

In a city, there are $N \leq 100,000$ road junctions, where $M \leq 300\,000$ pairs of junctions are connected by a road. You want to make all roads in the city one-way, so that travel is only allowed in one direction on the road[a]. However, it's important that you can still travel between any pair of road junctions in the city. Determine whether it's possible to direct the roads in this manner, and if so, what direction each road should have.

[a] A directed graph created by making all edges in an undirected graph directed is called an ***orientation*** of the graph.

Solution. As is the case with many DFS problems, you have to first figure out that the DFS tree is the key to solving the problem. We get that hint in this problem by finding a simple criterion for when it's impossible to direct the roads, namely the graph can't contain a bridge. Remember from the last problem that when $u \to v$ *isn't* a bridge, we could construct a path from v back to u by going from v to a back edge further down in the tree that connected upwards to an ancestor of u. If you take a look again at the DFS tree in Figure 14.6, you'll note that this path only traverse edges in the direction that the edges point, i.e. tree edges away from the DFS root and back edges towards it.

Since the DFS tree spans the whole graph, you can always start at the root of the tree and move along the directed tree edges to any vertex in the graph. Using the trick above, you can also go from any vertex in the tree back to the root. Together, this lets you move between any pair of vertices by first going back to the root and then down in the tree to your target. □

Problem 14.32.

One-Way Streets	onewaystreets
Disconnectable Doodads	disconnectabledoodads

Strongly Connected Components

So far, we've only used the DFS on undirected graphs. For directed graphs, it can be used to find a similar kind of decomposition into components.

Definition 14.2 — Strong Connectivity
A directed graph is ***strongly connected*** if there is a directed path from every vertex to every other vertex. The maximal vertex subsets that are strongly connected are the graph's ***strongly connected components*** (abbreviated *SCCs*).

A strongly connected graph is exactly the type that we created by orienting the edges of an undirected graph in *Directing Streets*. There are some differences and similarities between strongly connected components and biconnected components.

Exercise 14.33. Prove that

1. all vertices on a cycle belong to the same strongly connected component.
2. the strongly connected components form a partition of all the vertices.

Exercise 14.34. Consider the undirected graph taken by replacing all directed edges of a strongly connected graph with a single, undirected edge with the same endpoints as the directed graph. Show that the graph may contain cutvertices, but not bridges.

Exercise 14.35. Prove that a graph and its ***transpose graph*** – the same graph where each edge is reversed – have the same strongly conencted components.

Strongly connected components are almost always used only for a single purpose: collapsing a graph into a DAG that, in a sense, preserves the pairwise connectivity in the graph.

Definition 14.3 — Condensation
Given a directed graph G, it's *condensation* C is a directed graph with G's strongly connected components as vertices, and directed edges from one SCC to another if there is a directed edge in G from a vertex in the first SCC to the other.

The condensation of a graph must be acyclic. If a cycle passed through different SCCs they would all be a single SCC by Exercise 14.33.

Strongly Connected Components – scc

Given a directed graph with $V \leq 100\,000$ vertices and $E \leq 300\,000$ edges, find its strongly connected components.

Solution. Finding SCCs is only slightly more complicated than the DFS for biconnected component, but we'll try to deduce what extra bookkeeping is required from first principles. Conceptually, we base the algorithm around the condensation graph. Assume that we start a DFS within some arbitrary SCC of the graph. If we could determine when the DFS backtracks along one of the edges of the condensation graph, i.e. from one SCC back to another, we could find the SCCs using the biconnected component method. During the

DFS we put a newly visited vertex a stack, and whenever we backtrack along one of those edges, we create a new component with all vertices visited after first traversing that edge.

To see why this is correct, we need the condensation graph. The first time the DFS backtracks along one of the edges in the condensation graph, it will be after visiting an SCC without any outgoing edges. At that point, the DFS have visited all vertices in the SCC, pushing all those vertices to the stack since all vertices in the SCC are reachable from every other vertex. Thus when backtracking along the edge, a new component will be created containing all those vertices. After this, the state of the DFS is as if those vertices never existed, so we can repeat the argument once more for the next time we backtrack, and so on.

To determine whether an edge, we do the same thing as for the biconnected case, keeping track of the minimum depth we can reach from backedges in a subtree, with the caveat that we must ignore edges to vertices that we've already placed in a component. Finally, to simplify the implementation, we perform this check at the end of the DFS function rather than when looping over neighbors. This lets us pretend that we backtrack from the first vertex visited as well so that the first SCC visited is correctly detected.

```
1:  D ← new int[]
2:  procedure SccDFS(v, d)
3:      push v to the stack
4:      D[v] ← d
5:      mind ← d
6:      for each edge v → u do
7:          if u hasn't been placed in a component yet then
8:              if u is unvisited then
9:                  mind ← min(mind, SCC(u, d + 1))
10:             mind ← min(mind, D[u])
11:     if mind = d then
12:         pop all vertices up to and including v from the stack as a new SCC
        return mind
13: procedure SCC(vertices V)
14:     for each edge vertex v in V do
15:         if v hasn't been placed in a component yet then
16:             SccDFS(v, 0)
```

□

Problem 14.36.

Dominos dominos

While it's easy enough to generate the full condensation graph after finding the SCCs, there will be plenty of problems where you won't need to. In many cases you might even get away with just some local properties of the SCCs in the graph, for example counting the in- and out-degrees of each SCC.

Semi-Strong Connectivity – semistrongconnectivity

A directed graph is called ***semi-strongly connected*** if, for every pair of vertices u and v, there is a directed path from either u to v or v to u (or both). Given a graph with $V \leq 100\,000$ vertices and $E \leq 300\,000$, determine whether it's semi-strongly connected.

Solution. The condensation graph is the right tool here, seeing as how it preserves pairwise connectivity in a very simple DAG format. Thus, the problem can be reduced to whether an arbitrary DAG is semi-strongly connected or not, a considerably easier problem. By studying how adjacent vertices of the the topological ordering of the DAG can be connected by a direct path, we arrive at the following.

Exercise 14.37. Prove that a DAG is semi-strongly connected if and only if it contains a Hamiltonian path.

Does this mean that you need to explicitly construct the DAG and find its topological order, or use DP to find the longest path in it? No, not this time either. The SCC algorithm conveniently produces the components in reverse topological order, so it's enough to check if there is an edge to the previously generated component whenever a new one is found. □

Problem 14.38.
Proving Equivalences equivalences

2-Satisfiability

One of the most beautiful applications of the DFS is for solving the following problem. Before seeing the solution, it's almost hard to imagine that the problem can be solved in polynomial time at all, much less by a single application of the DFS.

2-Satisfiability – 2sat

Let $v_1, v_2, \ldots, v_n$ be $n \leq 100\,000$ Boolean variables. You are given m statements, each of the form "at least one of 'v_i is true/false' and 'v_j is true/false' holds". For example, "at least one of 'v_2 is true' and 'v_4 is false' holds".

Find an assignment of true/false values to the variables that simulatenously satisfy all statements, or determine that no such assignment exists.

Solution. To find a graph formulation we can use to attack the problem, we start with an elementary logical manipulation: rewriting a conjunctive ("or") clause as an implication. For example, the statement "at least one of 'v_2 is true' and 'v_4 is false' holds" is logically equivalent to the statement "if v_2 is false, v_4 is false". As for any logical statement, its *contrapositive* also holds, i.e. the implication "if v_4 is true, v_2 is true". These implications look very much like directed edges in a graph where each of the $2n$ statements "v_i is true/false" are vertices, so suddenly a graph approach doesn't seem entirely unlikely. Our

next step should then be to analyze the structure of the graphs generated by solvable and non-solvable instances.

For simplicity, we'll denote the vertex "v_i is true" by v_i, and "v_i is false" by $\neg v_i$. Since implications are transitive[1], if a vertex a can reach some vertex b in this graph, then "if a, then b" must hold. A special case now hints directly at the DFS. If both a can reach $\neg a$ and $\neg a$ can reach a, the instance that generated the graph is non-solvable, since either assignment of truth value to a leads to a contradiction. Determining whether this is true is easy – this condition is equivalent to a and $\neg a$ belonging to the same strongly connected component in the generated graph.

SCC's should always trigger a closer look on the DAG generated by collapsing each SCC into a single vertex, as well as the topological sorting of this DAG. Let the rightmost vertex of the sorting (i.e. any without outgoing edges) be v_i. It may be the case that $\neg v_i$ implies v_i, but the opposite can't be true, since v_i can't reach any edges. So, it would seem that letting v_i be true at least has a higher chance of working out than letting $\neg v_i$ be true! This intuition is entirely correct.

Since v_i being true implies nothing, the only problem that can arise is that $\neg v_i$ is implied by some variable that must be true. If an implication of the form $v_j \to \neg v_i$ exists, then the contrapositive edge $v_i \to \neg v_j$ also exists, contradicting that v_i had no outgoing edges! Thus, after letting v_i be true, we can remove v_i and $\neg v_i$ from the topological sort and repeat the process.

What happens if the rightmost vertex of the DAG doesn't actually correspond to any variable v_i, but rather an SCC that we have collapsed into a single vertex? This makes little difference: we can actually pretend that they are actually a single variable by the following exercise.

Exercise 14.39. Let an SCC C in the graph contain some set of vertex, corresponding to some variables (possibly negated). Prove that some other SCC C' in the graph contains exactly those vertices corresponding to the negations of the variables in C.

Remember that the SCC algorithm also lists SCC's in topological order, so the above computations can be done with straightforward changes to the SCC algorithm in linear time. Just remember to check whether any variable and its negation belonged to the same SCC! □

Since 2-SAT is such an abstract problem, it's one of the harder standard algorithms to reduce problems to. There's often a lot less hints that a problem is solved by 2-SAT compared to other graph algorithms.

Tornjevi

Croatian Olympiad in Informatics 2007, Final Exam day 2 – tornejvi

[1]Meaning that if a implies b and b implies c, then a implies c.

On a $R \times S$ ($1 \leq R, S \leq 100$) grid, some of the squares contain a robber, a castle or a wooden tower with two cannons. Each tower can be configured so that its two cannons fire in one horizontal (left or right) and one vertical direction (up or down). Simultaneously, all cannons will fire a cannon ball. A cannon ball that hits a robber will destroy him and continue to fly in the same direction. If a cannon ball hits a castle, it stops without damaging the big and strong castle. However, if a cannon ball hits a tower, the tower would be destroyed.

Find a configuration for all of the towers' cannons such that all robbers are destroyed, and all towers remain undamaged, or determine that no such configuration exists.

Solution. To find the 2-SAT reduction, we need one crucial observation. While it might seem like any given robber may have up to 4 towers that could possibly hit it, there can never be two towers that can hit it horizontally or vertically. If the robber had a tower both above and below it that could hit him, neither can fire on the robber since they would hit the other tower! Thus each robber has at most two options: it can be hit by a ball fired either horizontally or vertically.

This is a typical clue that 2-SAT might be involved, and that robbers might represent clauses (*"at least one of the horizontal or vertical towers must fire at the robber"*). That would imply that the directions in which each tower fires its cannons would be variables. Indeed, each tower can be represented by two Boolean variables, one to describe which horizontal direction it should fire in, and one for the vertical direction. □

Problem 14.40.

Illumination	illumination
Wedding	wedding

14.4 Minimum Spanning Trees

So far we've seen algorithms to generate several kinds of spanning trees in a graph. Dijkstra's algorithm constructed a shortest-path tree, while the DFS constructed a normal spanning tree. The third important type of spanning tree is the one of minimum total edge weight.

Minimum Spanning Tree – minspantree

We say that the weight of a spanning tree is the sum of the weights of its edges. A ***minimum spanning tree*** (MST) is a spanning tree whose weight is minimal. Given a weighted graph, find a minimum spanning tree.

Solution. There are several famous algorithms for finding a minimum spanning tree. We derive most popular one, ***Kruskal's algorithm***, but mention one based on Dijkstra's algorithm called ***Prim's algorithm*** briefly afterwards.

The method is similar to how we solved greedy problems like scheduling incrementally by constructing the solution one part at a time. Assume that we had a magic way of knowing at least one edge $\{a, b\}$ that is present in an MST. Since a and b are then connected in that

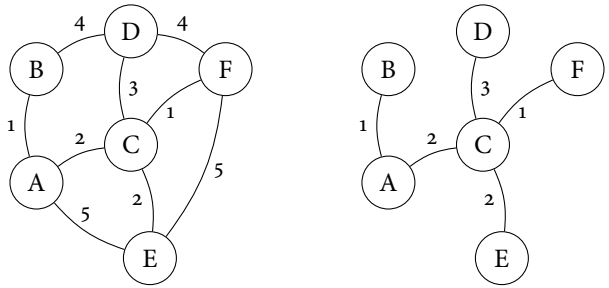


Figure 14.7: A graph with a corresponding minimum spanning tree.

MST, for the remaining edges in the MST it doesn't matter to which of *a* and *b* they are adjacent *as far as connectivity is concerned*. The same situation arose in the union-find example Subway Planning (p. 229). We took advantage of it by contracting the two vertices, replacing *a* and *b* with a new vertex *ab* adjacent to the same edges as the original vertices. Figure 14.8 shows the procedure in action.

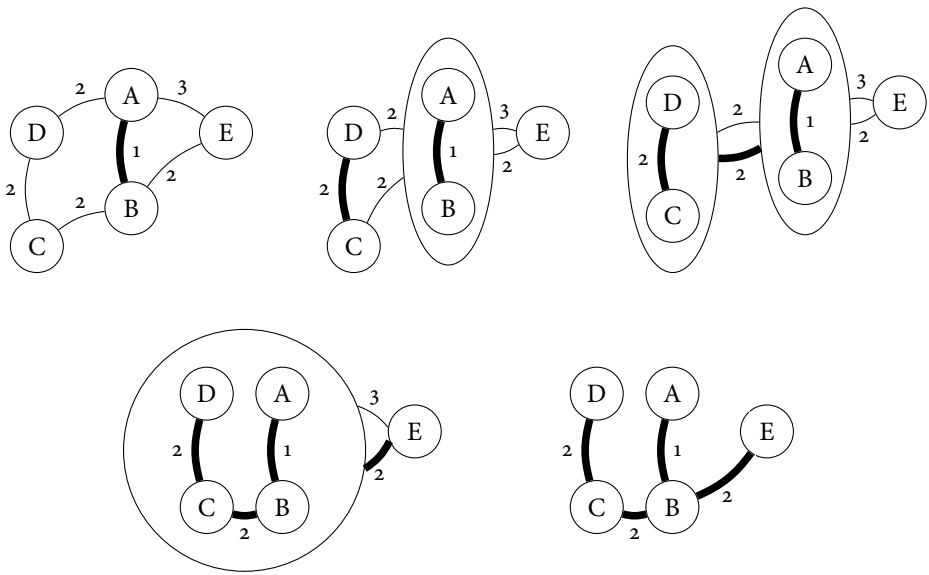


Figure 14.8: Incrementally constructing a minimum spanning tree by merging. The bold edges are the ones we somehow figured out must be present in the MST.

We still need a way to find an edge that we can always pick for the MST. In the scheduling problem, we found an interval that was always optimal by looking at various extremal cases among all the options. For MST we are trying to minimize the sum of all edge weights, so the most natural extremal case is the edge with the smallest weight. This

is optimal, and just as with scheduling can prove it using a swapping argument.

Assume that a minimum-weight edge $\{a, b\}$ with weight w is not part of any minimum spanning tree, so that all edges in the tree have weights $w' \geq w$. Take any minimum spanning tree and append this edge. A cycle then forms in the graph, including this particular edge. Deleting an edge that lies on a cycle never disconnects a graph, so removing a single edge on the cycle again gives us a spanning tree. Since all the other edges on the cycle has a weight $w' \geq w$, the sum of weights after removing one of them is changed by $w - w' \geq 0$, so the new tree is at least as good as the original one. As a contradiction, the edge $\{a, b\}$ was actually part of a minimum spanning tree.

As in Subway planning, edge contractions are not performed explicitly, instead opting for union-find. A neat way of repeatedly finding the minimum-weight edge between two distinct components is to iterate through **all** edges by increasing weight and use union-find to check if its endpoints are already connected or not.

```
1: procedure KruskalsAlgorithm(vertices V, edges E)
2:     uf ← new union-find
3:     for each edge {a, b} ∈ E sorted by increasing weights do
4:         if not uf.sameSet(a, b) then
5:             add (a, b) to the MST
6:             uf.union(a, b)
```

The time complexity is dominated by the $O(E \log E) = O(E \log V)$ sorting. □

Problem 14.41.

Island Hopping	islandhopping
Jurassic Jigsaw	jurassicjigsaw
Driving Range	drivingrange

There are many types of harder MST problems. Sometimes they're after a slight variation of Kruskal's algorithm, other times the underlying graph is given only implicitly and is either too large or a bit tricky to construct. We show one example of each type.

Svemir – svemir

By Goran Žužić. Croatian Open Competition in Informatics 2009/2010, contest #7.

Fourth Great and Bountiful Human Empire is developing a transconduit tunnel network connecting all it's $N \leq 100\,000$ planets, each represented as a point in 3D space. The cost of forming a transconduit tunnel between planets a and b is:

$$\min\{|x_a - x_b|, |y_a - y_b|, |z_a - z_b|\}$$

where $-10^9 \leq x_i, y_i, z_i \leq 10^9$ are the coordinates of planet i. The empire needs to build exactly $N - 1$ tunnels in order to fully connect all planets, either by direct links or chains of links. Compute the

lowest possible cost of successfully completing this project.

Solution. While easily modeled as an MST problem, we can't construct all the 10^{10} of the graph. One possible optimization in cases like this is to prune the graph down to a manageable number of edges before running Kruskal's algorithm, normally by proving that certain edges can never be part of an MST. Consider two planets A and B, with cost w. If there's another path P between A to B that costs at most w the direct edge $\{A, B\}$ never needs to be included in a minimum spanning tree; it could just be replaced with P which connects A, B, and possibly *more* vertices at the same cost. There's a small exception to this: in a connected subgraph of zero-weight edges this is true for any pair of vertices, but at least a spanning tree of edges must be kept.

What does this mean for the edges we need to add for a planet A? Let's decompose the 3D version of the problem into each individual dimension. For uniformity, we can let two planets at (x_a, y_a, z_a) and (x_b, y_b, z_b) have three edges of weights $|x_a - x_b|$, $|y_a - y_b|$ and $|z_a - z_b|$ instead – this can't reduce the weight of an MST. Now, let's plot the X-coordinates of some planets (Figure 14.9).

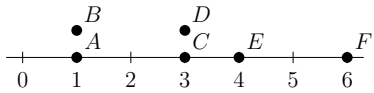


Figure 14.9

It's clear that for example the edge $\{A, E\}$ is unnecessary – it could be replaced by the edges $\{A, D\}$, $\{D, E\}$. In general, when no planets have the same coordinates it's sufficient to only add edges between the $N - 1$ pairs of planets at adjacent coordinates. If there's $k > 1$ planets on the same coordinate, $k - 1$ edges also needs to be added between them so that they are all connected. However, only a single of those planets need an edge to a planet at the adjacent x-coordinates since it's connected to the other planets at the same coordinates through 0-length paths. □

Problem 14.42.

Communications Satellite	communicationssatellite
Grid MST	gridmst

Inventing Test Data – inventing

By Peter Košinár. Internet Problem Solving Contest 2008. CC BY-SA. Shortened.

We are preparing a task for IPSC 2009: "Given a weighted undirected complete graph, find its minimum spanning tree." The only thing left is the input data, but creating it is not as simple as it looks. If the graph has more than one minimum spanning tree, we would have to write a complicated checker program to verify the answers, and we are too lazy to do this. Therefore we want to find an

input data that avoids such cases. If all the other edges were much more expensive, the minimum spanning tree would be obvious. Therefore we want the sum of all the edge weights to be as small as possible.

Given a weighted tree T of $N \leq 15\,000$ vertices, find the minimum possible sum of all edge weights in a complete graph G where T is the only spanning tree of G. All weights must be positive integers.

Solution. Recall the property that allowed us to greedily choose edges for an MST, i.e. that an edge $\{a, b\}$ can be part of an MST if and only if its weight does not exceed the heaviest edge on the path between a and b in the tree. The smallest possible weight we can add for each edge is then the weight of the heaviest edge on the path between its endpoints, plus 1. The heaviest edges can be found for all paths in the tree with a depth-first search from each vertex, providing us with an only slightly too slow $\Theta(N^2)$ algorithm.

For a faster solution we use the union-find technique from *Prominence* (p. 232). Add the edges of T to an empty graph one at a time in order of ascending weight, keeping track of the connected components. If two vertices a and b becomes connected after adding an edge of weight w, that means that the heaviest weight on their path in T has weight w. By keeping count of the size of each component, it's easy to compute the number of pairs that become connected after each edge is added. This takes $\Theta(N \log N)$ time in the worst case, dominated by sorting. □

Problem 14.43.

Landline Telephone Network	Landline Telephone Network

Additional Exercises

Problem 14.44.

Get Shorty	getshorty
Invasion	invasion
Arctic Network	arcticnetwork
Flowery Trails	flowerytrails
Eco-Driving	ecodriving
Texas Summers	texassummers
George	george
Delivery Delays	deliverydelays
Sirni	sirni
A Feast For Cats	cats
Senior Postmen	seniorpostmen
Backpack Buddies	backpackbuddies
Game	game
Muddy Hike	muddyhike
Haunted Graveyard	hauntedgraveyard

Arbitrage?	arbitrage
Slow Leak	slowleak
Self-Assembly	assembly
Grand Opening	grandopening
Firetrucks Are Red	firetrucksarered
Policija	policija
Duplex Connections	duplexconnections
Cantina of Babel	cantinaofbabel
British Menu	britishmenu
Beaming with Joy	beamingwithjoy

Chapter Notes

Almost everything about shortest paths that are relevant for algorithmic problem solving was published in a very active period around the 1960s. The Bellman-Ford algorithm was first published by Shimbel [44] in 1955. Ford's paper was not published until 1956, but made no mention of the Bellman-Ford algorithm as we know it today. Instead, he only noted that shortest paths can be computed by repeatedly finding an edge $e = (u, v)$ and substituting $d(u) \leftarrow \min(d(u), d(v) + w(e))$ until all the $d(u)$ converge. He ignores the matter of how fast – or even if – it converges, which Johnson [25] points out can take exponential time by proving that Dijkstra's algorithm (also based on this type of substitution), while correct in the presence of negative-weight edges, can take exponential time[2]. In 1958 Bellman [5] published almost verbatim the slightly longer version of the Bellman-Ford algorithm that we presented.

Dijkstra [14] published the quadratic version of his famous algorithm in 1959, although it had been conceived of a few years earlier. Interestingly none of the papers at the time considered negative edge weights, which is why Dijkstra simply states that his algorithm is preferable to Bellman-Ford. It was easy for Dijkstra to miss that his algorithm is easily optimized to $O(E \log V)$ with heaps, considering that heaps weren't discovered until 1964. A different Johnson [26] put the pieces together in 1972, to give us the modern version of Dijkstra's algorithm.

Floyd-Warshall has a shorter history. In 1959 Roy [42] published a special case of Floyd-Warshall that determined pairwise reachability in a graph (its "transitive closure"). Warshall [56] was late to the game, waiting until 1962 to publish the same algorithm. Floyd [18] finally referenced Warshall's paper when he extended it to compute pairwise shortest path that same year.

For the component applications of the DFS, Robert Tarjan [52] is the authority. It's no surprise that he was one of the authors first describing the SCC-based 2-SAT algorithm

[2] His paper was in turn written as a counterexample to Edmonds' and Karp's [15] claim that Dijkstra's algorithm only takes $O(VE \log V)$ with negative edge weights.

[3] either, though more complicated linear-time algorithms already existed.

Chapter 15

Maximum Flows

This chapter studies so called *flow networks*, and algorithms we use to solve the so-called *maximum flow* and *minimum cut* problems on such networks. Flow problems are common algorithmic problems, particularly in ICPC competitions (while they are out-of-scope for IOI contests). They are often hidden behind statements which seem unrelated to graphs and flows, especially the *minimum cut* problem.

Finally, we will end with a specialization of maximum flow on the case of bipartite graphs (called *bipartite matching*).

15.1 Flow Networks

Informally, a flow network is a directed graph that models any kind of network where paths have a fixed capacity, or throughput. For example, in a road network, each road might have a limited throughput, proportional to the number of lanes on the road. A computer network may have different speeds along different connections due to e.g. the type of material. These natural models are often use when describing a problem that is related to flows. A more formal definition is the following.

> **Definition 15.1 — Flow Network**
> A ***flow network*** is a special kind of **directed** graph (V, E, c), where each edge e is given a non-negative ***capacity*** $c(e)$. Two vertices are designated the ***source*** and the ***sink***, which we will often abbreviate to S and T.

In Figure 15.1, you can see an example of a flow network.

In such a network, we can assign another value to each edge, that models the current throughput (which generally does not need to match the capacity). These values are what we call flows.

> **Definition 15.2 — Flow**
> A ***flow*** is a function $f : E \to \mathbb{R}_{\geq 0}$, associated with a particular flow network (V, E, c). We call a flow f ***admissible*** if:
>
> - $0 \leq f(e) \leq c(e)$ – the flow does not exceed the capacity
> - For every $v \in V \setminus \{S, T\}$, $\sum_{e \in in(v)} f(e) = \sum_{e \in out(v)}$ – flow is conserved for each

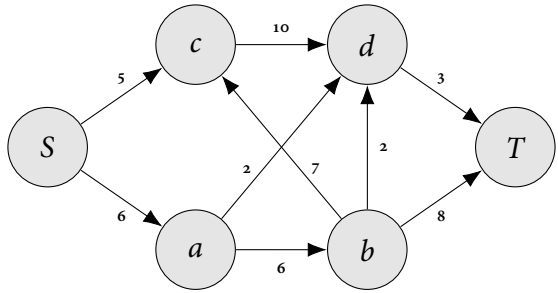


Figure 15.1: An example flow network.

vertex, possibly except the source and sink.

The ***size*** of a flow is defined to be the value

$$\sum_{v \in \text{out}(S)} f(v) - \sum_{v \in \text{in}(S)} f(v)$$

In a computer network, the flows could e.g. represent the current rate of transfer through each connection.

Exercise 15.1. Prove that the size of a given flow also equals

$$\sum_{v \in \text{in}(T)} f(v) - \sum_{v \in \text{out}(T)} f(v)$$

i.e. the excess flow out from S must be equal to the excess flow in to T.

In Figure 15.2, flows have been added to the network from Figure 15.1.

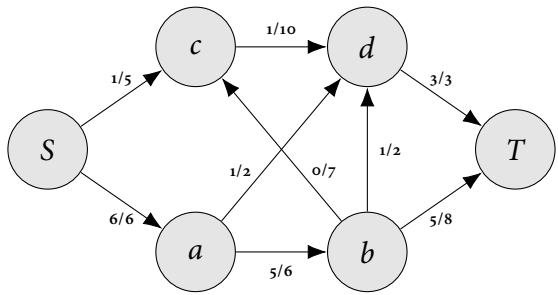


Figure 15.2: An example flow network, where each edge has an assigned flow. The size of the flow is 8.

Given such a flow, we are generally interested in determining the flow of the largest size. This is what we call the *maximum flow* problem. The problem is not only interesting

on its own. Many problems which we study might initially seem unrelated to maximum flow, but will turn out to be reducible to finding a maximum flow.

Maximum Flow

Given a flow network (V, E, c, S, T), find the maximum flow from S to T and how much flow should be assigned to each edge in one possible flow.

Exercise 15.2. The flow of the network in Figure 15.2 is not maximal – there is a flow of size 9. Find such a flow.

Before we study problems and applications of maximum flow, we will first discuss algorithms to solve the problem. We can actually solve the problem greedily, using a rather difficult insight, that is hard to prove but essentially gives us the algorithm we will use. It is probably one of the more complex standard algorithms that is in common use.

15.2 Edmonds-Karp

There are plenty of algorithms which solve the maximum flow problem. Most of these are too complicated to be implemented to be practical. We are going to study two very similar classical algorithms that computes a maximum flow. We will start with proving the correctness of the *Ford-Fulkerson* algorithm. Afterwards, a modification known as *Edmonds-Karp* will be analyzed (and found to have a better worst-case complexity).

Augmenting Paths

For each edge, we define a *residual flow* $r(e)$ on the edge, to be $c(e) - f(e)$. The residual flow represents the additional amount of flow we may push along an edge.

In Ford-Fulkerson, we associate every edge e with an additional *back edge* $b(e)$ which points in the reverse order. Each back edge is originally given a flow and capacity 0. If e has a certain flow f, we assign the flow of the back-edge $b(e)$ to be $-f$ (i.e. $f(b(e)) = -f(e)$. Since the back-edge $b(e)$ of e has capacity 0, their residual capacity is $r(b(e)) = c(b(e)) - f(b(e)) = 0 - (-f(e)) = f(e)$.

Intuitively, the residual flow represents the amount of flow we can add to a certain edge. Having a back-edge thus represents "undoing" flows we have added to a normal edge, since increasing the flow along a back-edge will decrease the flow of its associated edge.

The basis of the Ford-Fulkerson family of algorithms is the *augmenting path*. An augmenting path is a path from S to T in the network consisting of edges $e_1, e_2, ..., e_l$, such that $r(e_i) > 0$, i.e. every edge along the path has a residual flow. Letting m be the minimum residual flow among all edges on the path, we can increase the flow of every such edge with m.

In Figure 15.3, the path S, c, d, b, T is an augmenting path, with minimum residual flow 1. This means we can increase the flow by 1 in the network, by:

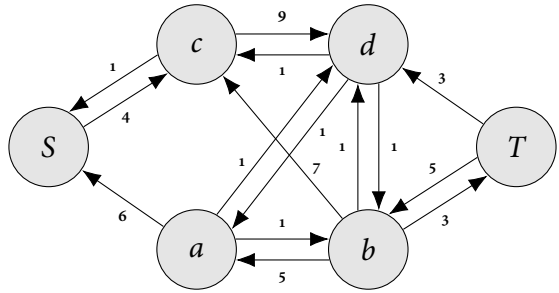


Figure 15.3: The residual flows from the network in Figure 15.2.

- Increasing the flow from S to c by 1
- Increasing the flow from c to d by 1
- Decreasing the flow from b to d by 1 (since (d, b) is a back-edge, augmenting the flow along this edge represents removing flow from the original edge)
- Increasing the flow form d to T

The algorithm for augmenting a flow using an augmenting path is simple:

```
1: procedure AUGMENT(path P)
2:     inc ← ∞
3:     for e ∈ P do
4:         inc ← min(inc, c(e) − f(e))
5:     for e ∈ P do
6:         f(e) ← f(e) + inc
7:         f(b(e)) ← f(b(e)) − inc
8:     return inc
```

Performing this kind of augmentation on an admissible flow will keep the flow admissible. A path must have either zero or two edges adjacent to any vertex (aside from the source and sink). One of these will be an incoming edge, and one an outgoing edge. Increasing the flow of these edges by the same amount conserves the equality of flows between in-edges and out-edges, meaning the flow is still admissible.

This means that a flow can be maximum only if it contains no augmenting paths. It turns out this is also a necessary condition, i.e. a flow is maximum if it contains no augmenting path. Thus, we can solve the maximum flow problem by repeatedly finding augmenting paths, until no more exists.

Finding Augmenting Paths

The most basic algorithms based on augmenting paths is the *Ford-Fulkerson* algorithm. It uses a simple DFS to find the augmenting paths:

```
1:  procedure AugmentingPath(flow network (V, E, c, f, S, T))
2:      bool[] seen ← new bool[|V|]
3:      Stack stack ← new Stack
4:      found ← DFS(S, T, f, c, seen, stack)
5:      if found then
6:          return stack
7:      return Nil
8:  procedure Dfs(vertex at, sink T, flow f, capacity c, path p)
9:      p.push(at)
10:     if at = T then
11:         return true
12:     for every out-edge e = (at, v) from at do
13:         if f(e) < c(e) then
14:             if DFS(v, T, f, c, p) then
15:                 return true
16:     p.pop()
17:     return false
```

For integer flows, where the maximum flow has size m Ford-Fulkerson may require up to $O(Em)$ time. In the worst case, a DFS takes $\Theta(E)$ time to find a path from S to T, and one augmenting path may contribute only a single unit of flow. For non-integral flows, there are instances where Ford-Fulkerson may not even terminate (nor converge to the maximum flow).

An improvement to this approach is simply to use a BFS instead. This is what is called the *Edmonds-Karp* algorithm. The BFS looks similar to the Ford-Fulkerson DFS, and is modified in the same way (i.e. only traversing those edges where the flow $f(e)$ is smaller than the capacity $c(e)$. The resulting complexity is instead $O(VE^2)$ (which is tight in the worst case).

15.3 Applications of Flows

We will now study a number of problems which are reducible to finding a maximum flow in a network. Some of these problems are themselves considered to be standard problems.

Maximum-Flow with Vertex Capacities

In a flow network, each vertex v additionally have a limit C_v on the amount of flow that can go through it, i.e.

$$\sum_{e \in in(v)} f(e) \leq C_v$$

Find the maximum flow subject to this additional constraint.

This is nearly the standard maximum flow problem, with the addition of vertex capacities. We are still going to use the normal algorithms for maximum flow. Instead, we will

make some minor modifications to the network. The additional constraint given is similar to the constraint placed on an edge. An edge has a certain amount of flow passing through it, implying that the same amount must enter and exit the edge. For this reason, it seems like a reasonable approach to reduce the vertex capacity constraint to an ordinary edge capacity, by forcing all the flow that passes through a vertex v with capacity C_v through a particular edge.

If we partition all the edges adjacent to v into incoming and outgoing edges, it becomes clear how to do this. We can split up v into two vertices v_{in} and v_{out}, where all the incoming edges to v are now incoming edges to v_{in} and the outgoing edges instead become outgoing edges from v_{out}. If we then add an edge of infinite capacity from v_{in} to v_{out}, we claim that the maximum flow of the network does not change. All the flow that passes through this vertex must now pass through this edge between v_{in} and v_{out}. This construction thus accomplish our goal of forcing the vertex flow through a particular edge. We can now enforce the vertex capacity by changing the capacity of this edge to C_v.

Maximum Bipartite Matching

Given a bipartite graph, a ***bipartite matching*** is a subset of edges in the graph, such that no two edges share an endpoint. Determine the matching containing the maximum number of edges.

The maximum bipartite matching problem is probably the most common reduction to maximum flow in use. Some standard problems additionally reduce to bipartite matching, making maximum flow even more important. Although there are others ways of solving maximum bipartite matching than a reduction to flow, this is what how we are going to solve it.

How can we find such a reduction? In general, we try to find some kind of graph structure in the problem, and model what it "means" for an edge to have flow pushed through it. In the bipartite matching problem, we are already given a graph. We also have a target we wish to maximize – the size of the matching – and an action that is already associated with edges – including it in the matching. It does not seem unreasonable that this is how we wish to model the flow, i.e. that we want to construct a network based on this graph where pushing flow along one of the edges means that we include the edge in the matching. No two selected edges may share an endpoint, which brings only a minor complication. After all, this condition is equivalent to each of the vertices in the graph having a vertex capacity of 1. We already know how to enforce vertex capacities from the previous problem, where we split each such vertex into two, one for in-edges and one for out-edges. Then, we added an edge between them with the required capacity. After performing this modification on the given graph, we are still missing one important part of a flow network. The network does not yet have a source and sink. Since we want flow to go along the edges, from one of the parts to another part of the graph, we should place the source at one side of the graph and the sink at the other, connecting the source to all vertices on one side and all the vertices on the other side to the sink.

Minimum Path Cover

In a directed, acyclic graph, find a minimum set of vertex-disjoint paths that includes every vertex.

This is a difficult problem to derive a flow reduction to. It is reduced to bipartite matching in a rather unnatural way. First of all, a common technique must be used to get introduce a bipartite structure into the graph. For each vertex, we split it into two vertices, one in-vertex and one out-vertex. Note that this graph still have the same minimum path covers as the original graph.

Now, consider any path cover of this new graph, where we ignore the added edges. Each vertex is then adjacent to at most a single edge, since paths are vertex-disjoint. Additionally, the number of paths are equal to the number of in-edges that does not lie on any path in the cover (since these vertices are the origins of the paths). Thus, we wish to select a maximum subset of the original edges. Since the subgraph containing only these edges is now bipartite, the problem reduces to bipartite matching.

Exercise 15.3. The minimum path cover reduction can be modified slightly to find a *minimum cycle cover* in a directed graph instead. Construct such a reduction.

Fly Again

By Pär Söderhjelm. Swedish Olympiad in Informatics 2012, Online Qualifiers.

Additional Exercises

Notes

The Edmonds-Karp algorithm was originally published in 1970 by Yefim Dinitz The paper by Edmonds and Karp

Chapter 16

Game Theory

In ordinary life, most of us are familiar with the concept of a game. We play video games, sports, card games, board games and many other kinds of games. Algorithmists primarily focus on non-real time games with well-defined rules, where determining who won is simple. Games such as chess, poker, tic-tac-toe or Yahtzee belong to this category, unlike soccer (running humans and the behavior of rolling balls are not sufficiently well-defined) or most video games where reaction speed counts. The mathematical area analyzing such games is called ***game theory***[1]. In algorithmic problem solving we further narrow the field of study into a small subset of this category: the so-called *combinatorial games*. Most problems ask us to determine when winning strategies of games exist against opponents playing perfectly.

16.1 Combinatorial Games

Combinatorial games are a category of mathematical games with some important properties that make them appropriate for algorithmic problems. First, players alternate in taking ***turns*** to perform what we call a ***move***. This is in contrast to games where players simultaneously pick what action to play in the game, as in rock-paper-scissor. We call the state of the game between these turns the ***positions*** of the game. Secondly, the game has so-called ***perfect information***. This means that both players are at all times aware of everything that could influence the future of the game. Most card games are disqualified by this criterion as players tend to have hidden cards unknown to their opponent. Finally, we ignore games which have any kind of random elements. Some examples of combinatorial games are:

- tic-tac-toe,
- chess,
- connect four,
- A and B take turns marking a square in an infinite grid with their respective letters; a player wins by making a 2×2 square of their letter.

The standard algorithmic problem is to determine if a player can force a win from some starting position in a game where both sides play perfectly, i.e. always choosing the best moves. Positions in the game are classified as *won* (there is a forced win), *lost* (the

[1] This also includes many economical constructs not widely thought of as games, e.g. auctions.

opponent has a forced win no matter how we play), or *drawn* (no player has a forced win). If optimal play from a starting position causes a game to go on forever, we classify it as drawn[2]. For example, the initial position of tic-tac-toe is drawn as neither player can win if the opponent plays perfectly. For the four example games above:

- tic-tac-toe is drawn,
- chess is not known to be winning, losing or drawn,
- connect four is known to be a forced win for the first player,
- the square game has no forced win for either player, so it continues forever, which we consider a draw.

Positions where no move is possible are called ***terminal*** positions. Depending on the game, different terminal positions may be considered won, lost or drawn. However, in most games all terminal positions have the same outcome (either won or lost). This does not restrict the types of games available through the following simplifications. Won terminal positions can be made non-terminal by adding a move to a losing terminal position. Similarly, since we consider infinite games to be drawn, a drawn terminal position can be made non-terminal by adding a move to itself, forcing an infinite game. Games are typically distinguished on whether they are are guaranteed to end, called a ***finite game***[3] or not, and whether terminal positions are lost (the most common type, called ***normal games***) or won (called ***misère games***).

Sometimes we must play the game optimally as an interactive problem, providing actual winning moves. The difference between finding the winner and playing the game is usually small. For games where we mathematically prove what positions are winning, the proof often constitutes an optimal strategy. When games require more computation to solve, a winning strategy can routinely be constructed with backtracking. Therefore, you can generally focus only on determining the winner of a game, rather than figuring out the winning strategy.

Problem 16.1.

Interactive Tic-Tac-Toe interactivetictactoe

16.2 Mathematical Techniques

In this section, we study some techniques that are typically used to solve games in mathematical problem solving. They are highly useful in many algorithmic problems as well.

The theory of games have a principal theorem that tells us two things: the problem we are trying to solve is well-defined, and there is a simple algorithm to do so.

[2]This can happen when making a move to break the loop would cause either player to lose, not an uncommon occurrence in chess.

[3]Note that a finite game may still possess an infinite number of positions!

Theorem 16.1 — Zermelo's Theorem
All positions in a finite game are either winning or losing.

Proof. We prove this by induction on the length $L(P)$ of the longest path from a position P to a terminal position. The claim is obvious for all terminal positions (i.e. with $L(P) = 0$). Now, assume that this is true for all positions P with $L(P) < l$ for some l. Consider any position P that has $L(P) = l$. This position can only reach positions P' with $L(P') < l$ by definition of $L(P')$. By the induction hypothesis, all those positions are either winning or losing. If one of them, P', is losing, P has the forced win $P \to P'$. Otherwise, all moves $P \to P'$ leads to a position that is winning for the opponent, so P must be losing. Thus P wins or loses too, completing the induction. □

By sorting positions such that if there is a move $P \to P'$, P' comes before P, we can use the inductive step of the theorem to solve the game. Specifically, we iterate through all positions in this order and mark each of them as winning if and only if it can move to a losing position. We shall call this method the ***induction algorithm***.

Periodicity

The first step in solving most games is to use the induction algorithm to solve small cases and look for patterns. One of the more common patterns is periodicity, which we can often prove holds in the general case using induction.

Alex and Barb – alexandbarb

By Dante Bencivenga. Calgary Collegiate Prog. Contest 2020. CC BY-SA 3.0. Shortened.

Alex and Barb are playing a card game. There is a stack of $1 \le k \le 10^9$ cards. They take turns removing from m to n ($1 \le m \le n \le 10^9$) cards from the stack, beginning with Alex. The first player with no valid moves left loses. Determine which player wins the game provided that both play with an optimal strategy.

Solution. The solution method when using induction is highly formulaic. First, we must determine the terminal positions. Since at least m cards must be removed in a move, they are those with 0 to $m - 1$ cards. The game is in normal form, so the terminal positions are all losing.

Next, we identify winning positions that have a valid move to one of these losing terminal positions. A position x can be reached from the positions $[m + x, n + x]$. Extrapolating this to an interval of targets $[x, y]$ tells us that the positions with moves into this interval are $[m + x, n + y]$. This means that the positions that can reach any of the terminal losing positions $[0, m - 1]$ are the n positions $[m, m + n - 1]$.

The other aspect of determining positions is identifying losing non-terminal positions. They are the ones that can only reach winning positions. For example, we see that $m + n$ must be a losing position; it can only reach positions $[m, n]$, all of which are losing. In

fact, all positions in the interval $[m+n, 2m+n-1]$ can reach only the losing positions $[m, m+n-1]$. Notice that this new interval of losing positions $[m+n, 2m+n-1]$ has length m, as did the interval $[0, m-1]$ of losing positions. All the following n positions, $[2m+n, 2m+2n-1]$ are winning too by the same argument as for $[m, m+n-1]$.

A pattern has emerged. The first m positions lost, the next n won, the next m lost, and the next n won. That it holds in general is easily proven by induction. Assume that it holds for all positions up to $a(m+n)-1$ cards, so that the positions $[am+(a-1)n, a(m+n)-1]$ win. The next m positions $[a(m+n), (a+1)m+an-1]$ can only reach those winning positions, so they lose. The following n positions $[(a+1)m+an, (a+1)(n+m)-1]$ can all reach one of those losing positions, meaning they win. Therefore, the pattern holds up to $(a+1)(m+n)-1$ as well.

Since the outcome of the came has period $n+m$, Alex loses if and only if $k \bmod (n+m)$ is in $[0, m-1]$. □

Problem 16.2.
Ninety-Nine ninetynine

Invariants

A typical solution method is that of finding a ***winning invariant***. Imagine that we could find a property X of the positions in a finite, normal game, such that a position that is X can always move into a non-X position, while positions that are not X can only move into X positions. Then, the winning positions are exactly those with the property X. The idea is that if a player has an X position, they can force the opponent into a non-X position, who is forced to give the first player an X position back. Since the first player can keep the property X invariant for their own position, and an X position have at least one move by definition, an X position can not be losing (and so, by Zermelo's theorem, must win).

Note that this is just a reformulation of what it means to be a winning or losing position. It is mostly a method of thinking that is useful when you are familiar with it, and it can help shorten some proofs. As with most invariance solutions in algorithmic problems, the hardest part is finding the actual invariant.

The Board Game – bradspelet

By Erik Odenman. Swedish IOI Selection 2014.

Ann-Charlotte and Berit are playing a new board game. The game is played using an $n \times m$ rectangular wooden board and a chain saw. The two players alternate taking turns, where each turn is divided into two phases. The player to move starts with sawing the board into two new rectangular pieces with integer dimensions. In the second phase, the opponent chooses one of the pieces and throws it away. The game continues with the remaining piece. A player can not make a move if it is their turn to saw the board and it has dimensions 1×1. When this happens, that player loses the game. Determine if the first player wins or loses.

	1	2	3	4	5	6	7	8	9	10	11	12
1	L	W	L	W	L	W	L	W	L	W	L	W
2	W	L	W	W	W	L	W	W	W	L	W	W
3	L	W	L	W	L	W	L	W	L	W	L	W
4	W	W	W	L	W	W	W	W	W	W	W	L
5	L	W	L	W	L	W	L	W	L	W	L	W
6	W	L	W	W	W	L	W	W	W	L	W	W
7	L	W	L	W	L	W	L	W	L	W	L	W
8	W	W	W	W	W	W	W	L	W	W	W	W
9	L	W	L	W	L	W	L	W	L	W	L	W
10	W	L	W	W	W	L	W	W	W	L	W	W
11	L	W	L	W	L	W	L	W	L	W	L	W
12	W	W	W	L	W	W	W	W	W	W	W	L

Table 16.1: Winning and losing positions for all $1 \leq n, m \leq 12$.

Solution. We first solve some small instances with $n = 1$ by hand using the induction algorithm to see if some pattern emerges. Solving e.g. the first 10 values of m suggests that odd m are losing positions and even m are winning positions. Can we prove this by finding a strategy where a player can keep the evenness of their position invariant? If Ann-Charlotte has a $1 \times 2k$ board, she can split the board into widths 1 and $2k - 1$ no matter what k is. Berit must pick the $1 \times (2k - 1)$ board (odd width!) and split it into one board of even width and one of odd width. Ann-Charlotte can then keep the board of even width. As Ann-Charlotte could keep evenness invariant and always has a possible move, the evenness is a winning invariant.

Finding the correct invariant for general n is harder and requires either mathematical ingenuity or solving a lot of small cases and pattern matching. Since we are programmers, filling in a table of all winning positions for $n, m \leq 12$ using the induction algorithm is the simpler option (Table 16.1). One thing should stand out: the rows for 1, 3, 5, 7, 9, 11 are the same, as are the rows for 2, 6, and 10 and those of 4 and 12. Only $n = 8$ is not identical to another row. What unites the numbers with identical rows? If we look at the smallest number in each, we notice that they are the powers of two 1, 2, 4, 8. This is a strong hint, and drawing some inspiration from number theory, we might notice that the numbers in each group are those exactly divisible by the same power of two[4]: 2^0, 2^1, 2^2, and 2^3 respectively. The next step should then be to compress the table and only focus on a single of these unique categories of answers.

After studying Table 16.2, the hypothesis presents itself: a position is losing when n and m are exactly divisible by the same power of two. The winning player can easily keep this invariant. Assume that for some winning position, $2^k \mid\mid n$ and $2^l \mid\mid m$, with $k < l$. To restore the invariant, the winning player can make the vertical split $(2^k, m - 2^k)$.

[4] An integer n is exactly divisible by 2^k when $2^k \mid n$ but $2^{k+1} \nmid n$, as we learn in Chapter 17 on number theory. We use the notation $2^k \mid\mid n$ for this.

	1	2	4	8
1	L	W	W	W
2	W	L	W	W
4	W	W	L	W
8	W	W	W	L

Table 16.2: Winning and losing positions only for powers of 2.

Both options that the losing player can choose are now exactly divisible by 2^k. In a losing position with $k = l$, every split along one dimension, e.g. n into $(a, n-a)$ results in at least one piece not exactly divisible by 2^k. Assume to the contrary that they both are. Factoring out 2^k from the numbers we get that $a = x2^k$ and $n - a = y2^k$, for some odd x and y. Then $n = (y + x)2^k$ where $y + x$ is even, so $2^{k+1} \mid n$, contradicting our assumption that $2^k \parallel n$. □

Competitive Tip

The popular C++ compilers support many convenient builtin functions for competitive programming. For example, the function `__builtin_ctz(x)` counts the number of trailing zeroes of an `int x` when written in binary. This is equal to the exponent of the power of two that exactly divides x (unless $x = 0$, for which the result is undefined), giving us a one-liner for the previous problem.

Problem 16.3.
Breaking Branches breakingbranches

Symmetry

When children first starts to play chess, an early attempted strategy is playing symmetrically. If white moves the pawn on the e file two squares forward, the symmetric opponent moves black's e file pawn two squares forward, and so on. Of course, such a strategy is eventually doomed to fail in chess. White can at some point make a move to which black has no symmetric response, such as a capture of the symmetrically placed piece or a check. In many mathematically inspired chess problems, this strategy works better.

Knight Packing – knightpacking

On an $n \times n$ chess board, two players alternate placing a *knight* on the board. A knight can only be placed on a square if there is no other knight placed either 1 row and 2 columns or 2 rows and 1 column away from it on the board. The first player who cannot place a knight on the board loses. Given $n \leq 10^9$, determine if the first or second player to move wins.

Solution. In all problems of this kind, where one is supposed to make a move by e.g. choosing a square to place something at, the first question should be: if my opponent makes the first move, is there some symmetric move I can always make? For grids, this

symmetry can manifest in several ways, such as mirroring a move along an axis or rotating it 180° around a center.

In the case where n is even, the second player can always copy the opponent's move be reflecting it vertically along the middle of the board (see Figure 16.1). By always playing

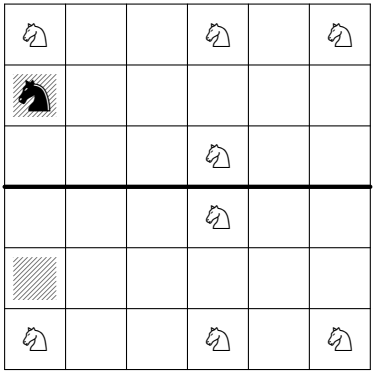


Figure 16.1: An $n = 6$ game where the first player just played their fifth move (the black knight), with the corresponding symmetrical position marked.

vertically symmetrically, the upper and lower halves of the board is identical after each of the second player's moves. This means that the moves available on each half of the board must be the same too. Thus, if a move is possible to make, its symmetrical move is also possible to make. There is caveat though – it could be the case that the last placed piece of the first player makes the symmetrical move impossible. Since a square and its vertical reflection is always on the same column, this can never happen in this problem.

When n is odd, this strategy fails. Reflecting any square along the middle row gives us back that same square, so we can not always play symmetrically. Does the other mirroring strategy we suggested work, i.e. rotation 180° around the center? It nearly does (prove this!), except for a single square – the center. In this case, the first player can place a knight in the center on their first move, and wins by copying the second player's move for all following moves. Thus, the first player wins when n is odd, and the second for even n. □

This is typical in symmetry problems. For certain positions, there is a simple symmetry that the second player can use, while other positions have some kind of non-symmetry (such as a central square) which the first player removes during their first move, simultaneously passing the move to their opponent. After the second player moves, the first player adopts the symmetric strategy and wins. This is often one of the factors that makes the outcome of games dependent on e.g. the parity of grid dimensions or a sequence length.

Exercise 16.4. On a square table, two players alternate in placing coins of the same size. Coins may not overlap or extend outside of the edge of the table. The first player who is unable to place a coin loses. Does the first or second player win?

Problem 16.5.
Chocolate Division chocolatedivision

Before moving on to the next technique, we show an example where a symmetric strategy is not as obviously relevant.

The Ice Cream Game – glasspelet

By Fredrik Ekholm and Nils Gustafsson. Swedish Olympiad in Informatics, Online Qualifiers 2020

Kirderf and Slin have a long table containing $1 \leq N \leq 10^6$ buckets of ice cream organized in a single row. Ice cream can have one of $M \leq N$ flavors. They take turns eating a bucket subject to the following constraints. First, they may only take either the leftmost or the rightmost of the remaining buckets. Secondly, they may never eat the last bucket of any ice cream flavor. If a player can't select a bucket of ice cream, they lose. Determine who has a winning strategy.

Solution. The game is quite complicated, so a good initial step is finding all the terminal positions to see if they give us insight. A terminal interval must contain a bucket of every flavor according to the rules, but also be a minimal such interval. Finding these is a standard application of the two pointers technique. The same player always have the turn at a given terminal interval, since this is only determined by the parity of the moves played, and thus the parity of the interval's length. If N is even, then even-length terminal positions means that the second player wins, and vice versa.

The next step in arriving at a solution is to consider what simple strategies are possible at all – here, symmetry should be one of your first candidates. For games on intervals, symmetry generally means reflection in the midpoint of the interval. In this case, that translates to eating ice cream buckets from the end opposite of where your opponent last ate. The intuition for why this strategy is something that we should investigate here is that **both** players can adopt it, and no matter which of them do, the resulting terminal state is almost the same. If both players can force the game to approximately the same state, odds are that the strategy benefits one of the players more than the other, and thus it would be optimal for them to adopt it.

So, which player benefits the most from symmetric play? Since such a strategy would remove equally many ice creams from both ends of the line, it is the terminal positions whose endpoints are closest to the center that become relevant. Formally, for each player, we consider the terminal interval where the endpoint furthest from the center is closest to the center.

Let us assume that Kirderf's best interval is k buckets in from the left edge of the table (and even further away from the right edge), while Slin's best interval is s buckets away from the right edge[5] (and even more buckets in from the left). If $s > k$, Slin wins by playing symmetrically. After playing $k + 1$ symmetric rounds, there can be no more

[5]The case where both intervals have their worst endpoint on the same side is uninteresting, as one of the intervals is then strictly closer to the center and easily winnable for that player.

winning intervals for Kirderf in the remaining interval, since the one closest to the center was less than $k+1$ buckets from the edges. Thus, the only terminal states left are wins for Slin. If instead $s \leq k$, Kirderf wins. Kirderf eats one bucket from the right on his first move. His interval is still k steps away from the left endpoint, while Slin's is $s-1$. It is then Slin's turn, and since $k > s-1$, Kirderf now wins after $s-1$ rounds of symmetric play. □

Passing the Move

A neat trick using the fact that any position either wins or loses is that of *passing the move*. Assume that there is some position P in the game that either we can move to, or force our opponent to move to. If P is losing, we can win the game by playing a move that takes the game to P. If P wins, we can instead try passing the move to our opponent and force them to take the game to P.

Block Game – blockgame2

By Mees de Vries, BAPC Preliminaries 2016. CC-BY SA 3.0. Shortened.

You have challenged a toddler to the following game. In front of you are two towers of $1 \leq n, m \leq 10^{18}$ blocks. You and the toddler take turns removing blocks from the larger of the two towers. The number of blocks removed must be a positive multiple of the number of blocks in the other tower. The first person to remove the last block from one of the towers wins. If you make the first move, who wins the game?

Solution. We represent positions in the game as pairs of the number of blocks (n, m) in the two towers. To simplify analysis slightly, we start by recognizing the similarity between the problem and the Euclidean algorithm – the operations performed are exactly the same. This means that n and m are always multiples of their greatest common divisor d, so we can without loss of generality divide d away from n and m without changing the game.

After this simplification, there is now only a single terminal position: $(0,1)$. Let us solve the cases where the smallest tower has height 1, starting with the position $(1,1)$. It has a *single possible move*, going to $(0,1)$. Furthermore, any position $(n,1)$ can reach both of these two positions. This is the archetypical pass the move opportunity. We have two positions $P = (1,1)$ and $P' = (0,1)$. P can only move to P', and all other positions $P'' = (n,1)$ can move to both of them. By using the pass the move technique, we can conclude that P'' must be winning. If P loses, we move to P, giving our opponent a losing position. If P wins, we pass the move by moving to P', forcing our opponent to move to P instead and giving us the win.

Does this trick work when the smallest tower has size 2 too? Consider the positions $P = (1,2)$ and $P' = (3,2)$. P' can only move to P, and all other moves $(5,2), (7,2), \ldots$ can reach both P' and P. Thus, they are all winning, with the correct strategy depending on whether P is winning or not.

The reasoning generalizes to all m: for a position (n, m) where $m \geq 2n$, we can always give the position $(n, m \bmod n)$ to our opponent, or get it ourselves by passing the move

$(n, m \bmod n + n)$ to our opponent.

How do we figure out if a position (n, m) is winning or losing if $n < m < 2n$? These positions only have a single possible move, namely to $(n, m - n)$, so we recursively use the same algorithm to determine of that position is winning or not instead. We already know from the Euclidean algorithm that the number of such steps we can be forced to take before arriving at $(0, 1)$ (or $m \geq 2n$) is logarithmic in n and m, so this is fast enough. □

16.3 Game Graphs

An abstraction of combinatorial games is that of the ***game graph***. We visualize a game by letting all positions be vertices in a graph and drawing directed edges $v \to u$ if it is valid to move from position v to position u. Two players (♖ and ♜) then alternate in making moves, where a move consists of moving a game piece placed at one of the vertices along an edge in the graph. When a player is unable to make a move, they lose.

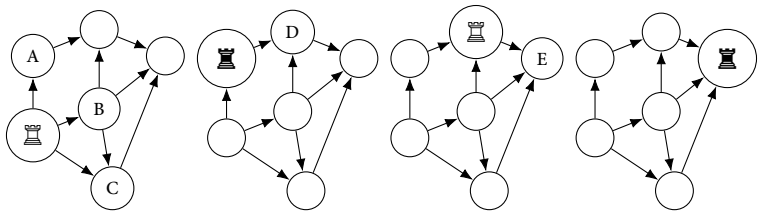


Figure 16.2: An example of a game graph with 6 positions. Player ♖ starts and has three possible moves *A*, *B* and *C*. ♖ chooses to move to *A*, whereupon ♜ responds with the only available move *D*. Finally, ♖ ends the game with the move *E*, leaving ♜ with no possible moves who therefore loses the game.

The finite games are exactly those with acyclic game graphs. Earlier, we described a solution to them – the induction algorithm. We will briefly revisit it in the context of solving a general game without patterns, followed by a nice optimization trick.

The induction algorithm is essentially dynamic programming. Whether a position P with moves P_i is winning or not is computed recursively by the rule

$$\text{Wins}(P) = \begin{cases} \text{true} & \text{if Wins}(P_i) \text{ is false for any move } P_i \\ \text{false} & \text{otherwise} \end{cases}$$

For simple games, a bottom-up solution can give us very short code.

Bachet's Game – bachetsgame

By Piotr Rudnicki. UofA Programming Contest 2002. CC-BY SA 3.0. Shortened.

Stan and Ollie play a game using $n < 10^6$ stones on a table. They take turns removing stones from the table, with Stan moving first. The player to take the last stone wins. The number of stones a

player can remove during a single turn must be one of $m < 10$ given numbers $a_1, \ldots, a_m$ (one of which is always 1, so there is always a possible move). Determine who wins the game.

Solution. Since the number of stones on the table always decreases after a move, the recursion can be solved bottom-up in increasing order of stones. The base case is the terminal position 0 which is losing, since it means the opponent just took the last stone and won. Then, for each other position x, we consider the available moves $x - a_1$, $x - a_2$, and so on. If any of them is a losing position, we mark x as won. Otherwise, x is marked as lost. This takes $O(m)$ time per state, for a total complexity of $O(nm)$, which is fast enough. □

A possible implementation of the bottom-up induction in the previous problem is very short:

```
wins[0] = true;
for (int i = 0; i < N; i++)
for (int ai : moves)
if (ai >= i)
wins[i] |= wins[i - ai];
```

Whether bottom-up or top-down is best depends a lot on the problem at hand. For very complicated game graphs, where a topological ordering of all positions is annoying to determine, top-down is of course simplest. Typically, the top-down version is coded as follows.

```
1: procedure IsWinning(position v)
2:     if v ∈ memo then
3:         return memo[v]
4:     for every move v → u do
5:         if not IsWinning(u) then
6:             return memo[v] = true
7:     return memo[v] = false
```

Problem 16.6.
A Multiplication Game amultiplicationgame

Another benefit of top-down dynamic programming is that it can automatically prune some unnecessary, unreachable states from computation, while bottom-up processing usually explicitly visit every imaginable state in a recursion. In this particular recursion, we can sometimes prune *a lot* of states. The trick is the loop on line 4, which short-circuits once a move to a losing position is found, rather than recursing into all possible moves.

The Bit Game – bitgame

The two siblings Alice and Bob play the following game. First, their mother picks a subset of non-negative integers less than 2^{35}. Alice then writes down either 0 or 1 on a piece of paper. Starting with

Bob, they now alternate appending either 0 or 1 to the number currently on the paper. Once the paper has 35 digits on it (meaning Bob writes down the last digit), it is read as a base 2 integer. If the resulting number is in their mother's subset, Alice wins. Otherwise, Bob wins. Your task is to determine who would win with optimal play.

Since the subset their mother picked can be very large, you are not given all the numbers in it. Instead, you can ask her if different integers are part of the subset.

Solution. The induction algorithm would recursively try all possible moves in the game. Since no two sequences of moves leads to the same position, memoization would not help us at all here. We can not even hope for the worst-case deterministic complexity to improve much on the total number of 2^{35} terminal positions in the game since the game is completely arbitrary. Therefore, we must put our hopes to expected time complexities instead. The key is that the IsWinning recursion short-circuits if the first move we try in a position wins for us. If at least one of the two moves available at a position wins for us, we sometimes get to skip checking the other one. By randomly choosing what move to try first, we are guaranteed to choose it with some probability.

This is unfortunately not always possible. For a losing position P, both of the moves P_0 and P_1 lose, so we have to recurse through both of them no matter which we pick first. However, P_0 in turn has two moves, say P_0' and P_1'. Since P_0 is a winning position, at least one of these two are losing. By randomly picking what moves to check first at P_0, we will find the winning move with probability 0.5 first. This means that we have to recurse on average 1.5 times. The same applies to P_1. The expected branching factor over two consecutive turns is thus 3, rather than the branching factor of 4 we would get if not short-circuiting. In total, the expected time complexity is then $O(3^{\frac{n}{2}}) = O(1.74^n)$ for n moves which is fine. □

A more careful analysis is possible to get a tighter bound for this randomized algorithm, which we leave as an exercise.

Exercise 16.7. Prove that the worst-case time complexity is $\Theta(\frac{1+\sqrt{33}}{4}^n)$ (about 1.69^n) in expectation.

This trick is a result of a more general optimization. As we shall see in Section 16.5, the induction algorithm is a special case of an algorithm called minimax that can solve games with real-valued rather than binary outcomes. This algorithm has an optimization called ***alpha-beta pruning*** (see the chapter notes for further references on this). For binary games, the alpha-beta optimization reduces to the short-circuit behavior of the top-down induction algorithm.

16.4 Cyclic Games

Games with cycles in their graphs are not necessarily finite. This does not mean that we cannot determine whether certain positions are forced wins or losses. It is in fact possible

to do so for every non-drawn position in the game. Using the induction algorithm as-is unfortunately fails. It depends on finding a topological ordering of the game graph, but such an ordering does not exist for game graphs containing cycles. The solution can be found in using graph tools we are already familiar with.

Cop and Robber – copandrobber

By Vytautas Gruslys. Baltic Olympiad in Informatics 2014

A cop and a robber are playing a game in a city, represented as an undirected graph with $2 \leq N \leq 500$ vertices. The cop starts by choosing a vertex to guard. Knowing where the cop is, the robber picks a vertex to rob a bank at. They now alternate moving, always aware of where the other player is before moving. The cop moves by either going to a neighboring vertex, or staying at the current vertex. The robber on the other hand must always move to a neighboring vertex – staying still is too risky for a wanted robber! If the two players occupy the same vertex after either player's move, the cop wins. Write a program that, given the graph, determines if the cop can win, and if so, plays the role of the cop in the game.

Solution. Before all else, we check whether the problem reasonably can be solved by using the game graph approach, or if the graph would be too big and requires smarter insights. A position in the game can be identified by three things: the position of the cop, the position of the robber, and which player moves next. This sums up to roughly 500 000 positions in total. Furthermore, each position has up to 500 possible moves, so the game graph has up to 250 000 000 edges. A solution linear in the number of edges should be fine, although we must take care not to represent the entire game graph explicitly. Instead, generating the moves for a given vertex when needed makes sure we do not exceed available memory. Now, we need to answer two questions. How do we play the game if we know what positions are winning for the cop, and how do we perform this classification?

When choosing the initial vertex for the cop, we pick any one where the cop has a winning position no matter what vertex the robber starts at. If there is no such vertex, the cop can not win the game.

Making optimal moves during the game is trickier. In the finite game case, we could pick any of the moves that leads to a losing position for our opponent. In the cyclic case, this may fail. Choosing arbitrary winning moves could cause us to go back and forth between winning positions forever. For example, consider the case where the city is just a line of vertices. All positions are winning for the cop in this case (just move towards the robber). Picking arbitrary winning moves could then lead to the cop always choosing the "wait" move and thus never catching the robber, despite always having a winning position. The standard way to solve this is to choose the move that allows the cop to win in as few moves as possible against a robber playing perfectly (the ***depth to win***, or DTW metric). With this move the robber can never force us back to a previous position, since our positions must have strictly decreasing DTW.

Now, how do we actually solve the game graph? Let us step through our induction

algorithm and see where it fails. We can still mark any terminal positions, i.e. those where the cop catches the robber, as winning or losing. For those terminal positions where the robber has the turn, the position is losing, and where the cop has the turn, the position is winning. Both of these are possible, since the robber can be forced to move into the cop's vertex if that is their only neighboring vertex. Any position adjacent to one of those losing positions can then be marked as winning, and any positions with only adjacent winning positions as losing. Since this can result in new positions being determined, we repeat this process until all remaining positions either have no losing neighbor or at least one non-winning neighbor. This step is typically implemented as a breadth-first search similar to topological sorting *with all edges in the game graph reversed*, like in the following pseudo code:

```
1:  procedure CyclicGame(list of positions P)
2:      movesLeft ← new array of size |P|
3:      status ← new array of size |P|
4:      winningMove ← new array of size |P|
5:      q ← new queue
6:      for every v ∈ P do
7:          status[v] ← undetermined
8:          movesLeft[v] ← the number of moves going out from v
9:          if v has no moves then
10:             status[v] ← the result of the terminal position v
11:             q.push(v)
12:     while q is not empty do
13:         v ← q.pop()
14:         for every move u → v do                ▷ Note that moves go to v here.
15:             if status[u] = undetermined and status[v] = losing then
16:                 status[u] ← winning
17:                 winningMove[u] ← v
18:                 q.push(u)
19:             else
20:                 decrease movesLeft[u] by 1
21:                 if movesLeft[u] = 0 then
22:                     status[u] ← losing
23:                     q.push(u)
```

When it comes to choosing what move to make at a winning position, the algorithm picks the first losing position that was processed in the queue. This is correct, thanks to the following claim we leave as an exercise:

Exercise 16.8. Prove that the *CyclicGame* algorithm processes all states v (both winning and losing) in increasing order of DTW.

We claim that all positions left undetermined by the algorithm are drawn. This is

based on two observations: all undetermined positions have an undetermined neighbor, and the only determined neighbors of an undetermined position are losing. A player in an undetermined position is faced with exactly two choices: moving into another undetermined position, or moving into something that is a determined loss. No player wants to lose, so they instead keep moving between undetermined positions, drawing the game. □

Problem 16.9.
Grid Volleyball gridvolleyball

16.5 Minimax

In many games, especially those in real life, it's important not only to win, but to maximize your score doing so. We can solve them in the general case using a variant of the induction algorithm called the ***minimax algorithm*** which is just as efficient.

Peg Game for Two – peggamefortwo

By Arup Guha. 2018 North America ICPC Qualifier. CC-BY SA 3.0. Shortened.

Jacquez and Alia have a triangular peg game with 15 holes arranged in 5 rows a triangular pattern.

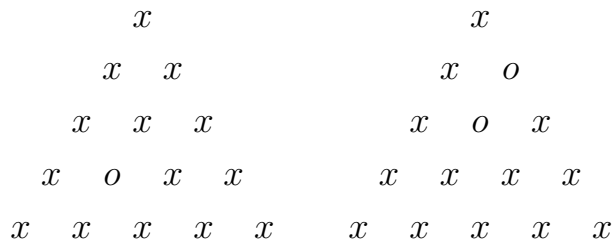


Figure 16.3: The initial state of the game, followed by the state after a diagonal jump was made with the rightmost peg on the second row.

Initially one hole is empty and the remaining 14 are filled with pegs (see Figure 16.3). The player moves by picking up a peg to "jump" it over an adjacent peg, landing on an empty hole adjacent to the jumped peg. Jumps must be in straight lines (horizontally or diagonally). The peg that is jumped over is removed. The players alternate making a jump on the board.

Each peg is assigned a positive point value. The score for a jump is the product of the point values for the two pegs involved in the jump. The total score of a player is the sum of the scores of their jumps. The game ends when a player has no possible jumps to make. Each player's goal is to maximize their own total score minus their opponent's total score (at the end of the game).

Write a program that, assuming that Jacquez starts, outputs the value of his score minus Alia's score if both players move optimally.

Solution. For a given position P, let us call the best possible difference in Jacquez and Alia's scores $S(P)$. What "best" means of course depends on which of the two players are

to move. Jacquez would prefer larger $S(P)$, while Alia wants to minimize it.

Deriving an algorithm to compute $S(P)$ follows intuition. If we are Jacquez at the starting position $S(P)$, which of the possible first moves $P_1, \ldots, P_k$ do we prefer to make? Clearly, the one that gives us the best result no matter how Alia can play. Formulating it differently, we want to pick the move where the sum of the score of the move, which we can call s_k, and the value of $S(P_k)$ is the highest. This gives us a recursion for the best score Jacquez can get at a position,

$$S(P) = \max_{P_k}(S(P_k) + s_k)$$

and, symmetrically, a recursion for when it is instead Alia's turn, who prefers to minimize this value instead:

$$S(P) = \min_{P_k}(S(P_k) - s_k).$$

Note the difference in sign on s_k since increasing Alia's score decreases $S(P)$. These recursions can be computed using dynamic programming over the less than 2^{15} game positions. What remains is generating all possible moves as a given position, which is a small implementation problem. □

The computation of those two recursions is the minimax algorithm in its entirety. The induction algorithm for binary games can be seen as a special case of minimax where each terminal state is assigned e.g. ∞ for first-player wins and $-\infty$ for second-player wins, with the first player maximizing the score and the second player minimizing it.

Additional Exercises

Problem 16.10.

Cutting Brownies	cuttingbrownies
String Game	stringgame
Joyless Game	joylessgame
Black Out	blackout
The Coloring Game	fargningsspelet
Game Strategy	game

Notes

Combinatorial games have a rich theory which we barely scraped the surface of in this chapter. For algorithmic problem solvers, most of it tends to be too complex to be of use. The Sprague-Grundy theorem [21, 48] from the 1930's is essentially the latest development in regular problem solving use.

John H. Conway's contributions to the topic of combinatorial game theory is among the most well-known. In particular *On Numbers and Games* [10] and the series *Winning*

Ways for your Mathematical Plays [6] discuss more advanced games that seldom appear in algorithmic problems.

The game theory we studied is also limited to solving general games only when they have small game graphs. For larger games, there are many optimization techniques for faster evaluation that we did not discuss here, such as *alpha-beta pruning* (although we mentioned it briefly in the context of binary games), or the relatively recent *Monte Carlo tree search*. The latter is famously used in the highly successful *AlphaZero* program which drove much of the development in chess and go computer engines when first developed. For additional reading, consider a book on AI such as *Artificial Intelligence: A Modern Approach* [43].

We have not talked about the general, non-combinatorial game theory, much of which has great economical implications. Von Neumann *Theory of Games and Economic Behavior* [55] is considered the seminal work on founding game theory as a distinct mathematical field. Most textbooks on game theory focuses on these more general games.

Chapter 17

Number Theory

Number theory is the study of certain properties of integers. It makes an occasional appearance within algorithmic problem solving, in the form of its sub-field *computational number theory*. It is within number theory topics such as divisibility and prime numbers belong.

In competitions, number theory problems range from simple applications of the main theorems you learn in the chapter, to harder tasks where you must combine number theoretical insights with other algorithmic techniques. The latter can even require solving difficult mathematical number theory problems in order to at the very least prove correctness of a solution, if not to find it. Most content in this chapter is learning efficient methods of computing the standard number theoretical objects, such as primes, modular inverses, divisors, and becoming well acquainted with time complexities and other asymptotic approximations that tend to arise in number theoretical problems.

A word of caution before you proceed. This chapter is mathematically heavy, and teaches you number theory from the ground up. You might find some example problems on the way obvious because of things you have previously learned, but might not have rigorously proven. Make sure to carefully follow the solutions even to those problems, because they include some fundamental theorems you might take for granted if you have not studied number theory.

17.1 Divisibility

All of the number theory in this chapter relate to a single property of integers, divisibility.

> **Definition 17.1 — Divisibility**
> An integer n is ***divisible*** by an integer d if there exists an integer q such that $n = dq$. We then call d a ***divisor*** of n. We use the notation $d \mid n$ to state that d is a divisor of n, and $d \nmid n$ when it is not.

Dividing both sides of the equality $n = dq$ with d gives us an almost equivalent definition, namely that $\frac{n}{d}$ is an integer. The difference is that the first definition admit the divisibility of 0 by 0, while the second one does not (zero division is undefined). When we speak of the divisors of a number in most contexts (as in Example 17.1), we generally consider only the non-negative divisors. Since d is a divisor of n if and only if $-d$ is a divisor of n, this sloppiness loses little information.

Example 17.1 — Divisors of 12
The number 12 has 6 divisors – 1 ($1 \cdot 12 = 12$), 2 ($2 \cdot 6 = 12$), 3 ($3 \cdot 4 = 12$), 4 ($4 \cdot 3 = 12$), 6 ($6 \cdot 2 = 12$) and 12 ($12 \cdot 1 = 12$).

12 is not divisible by e.g. 5 – we have $\frac{12}{5} = 2 + \frac{2}{5}$, which is clearly not an integer.

Exercise 17.1. Compute the divisors of 7, 18 and 39.

The concept of divisibility raises many questions. First and foremost – how do we check if a number is divisible by another? This question has one short and one long answer. For small numbers, i.e. those that fit inside the native integer types of a language, checking for divisibility is as simple as using the modulo operator (`%`) of your favorite programming language. An integer n is divisible by d if and only if `n % d == 0`, since this means $\frac{n}{d}$ has no remainder and is therefore an integer.

For large numbers, checking divisibility is more difficult. Some programming languages, such as Java and Python, have built-in support for dealing with large integers, but e.g. C++ does not.

Dual Divisibility – dualdivisibility

Given two positive integers a and b with the same number of digits ($1 \leq b \leq a \leq 10^{18}$), compute the number of divisors of a that have b as a divisor.

For example, with $a = 96$ and $b = 12$, there are 5 such numbers: 12, 24, 36, 48 and 96.

Solution. Assume that c is such a number. The solution falls out from some applications of the definition of divisibility. We have $a = cq$ and $c = bq'$ for some positive integers q, q'.

The value of q' is at most 9 by the following argument. If $q' \geq 10$, we have $a = cq \geq c \geq 10b$, but then a has more digits than b, a contradiction. Thus, we can simply test all the values of c by letting $q' = 1, 2, \ldots, 9$ and verifying that the two conditions hold using the modulo operator. □

Problem 17.2.

Meow Factor	meowfactor	
Evening Out 1	eveningout1	
Multiplication Table	multtable	*(for 1 points)*
Divisor Shuffle	divisorshuffle	

We now look at our first constructive problem – computing all divisors of an integer n.

Positive Divisors – positivedivisors

Given an integer $1 \leq n \leq 10^{15}$, compute all the positive divisors of n.

Solution. Every integer has at least two particular divisors called the ***trivial divisors***, namely 1 and n itself. If we exclude the divisor n, we get the ***proper divisors***. To find the remaining divisors, we can use the fact that any divisor d of n must satisfy $|d| \leq |n|$ (why?). This means that we can limit ourselves to testing whether the integers between 1 and n are divisors of n, a $\Theta(n)$ algorithm. We can do a lot better by exploiting a nice symmetry.

Hidden in Example 17.1 lies the key insight to speeding this up. It seems that whenever we had a divisor d, we were immediately given another divisor q. For example, when claiming 3 was a divisor of 12 since $3 \cdot 4 = 12$, we found another divisor, 4. This is not a surprise, given that the definition of divisibility (Definition 17.1) – the existence of the integer q in $n = dq$ – is symmetric in d and q, meaning divisors come in pairs $(d, \frac{n}{d})$.

Exercise 17.3. Prove that a positive integer has an odd number of divisors if and only if it is a perfect square.

Since divisors come in pairs, we can limit ourselves to finding one member of each such pair. Furthermore, one of the elements in each pair must be bounded by $\sqrt{n}$. Otherwise, we would have that $n = d \cdot \frac{n}{d} > \sqrt{n} \cdot \sqrt{n} = n$, a contradiction (again, 0 is a special case here where we always have $\frac{0}{d} = 0$). This limit helps us reduce the time it takes to find the divisors of a number to $\Theta(\sqrt{n})$, which allows us to solve the problem sufficiently fast.

```
1: procedure Divisors(N)
2:     divisors ← new list
3:     for i from 1 up to i² ≤ N do
4:         if N mod i = 0 then
5:             divisors.add(i)
6:             if i ≠ N/i then
7:                 divisors.add(N/i)
8:     return divisors
```

□

Problem 17.4.

Pascal	pascal
Almost Perfect	almostperfect
Candy Division	candydivision

Let us look at an application of this algorithm.

Subcommittees – subcommittees

In a parliament of $P \leq 10^{16}$ people, the speaker wants to divide the parliament into (at least two) disjoint subcommittees of equal size. Of course, the chair of such a subcommittee furthermore wants to divide their subcommittee into (at least two) sub-subcommittees of equal size, and so on,

until no further divisions can be performed.

What is the maximum number of levels of subcommittees can be created?

Solution. What different sizes may the first level of subcommittees have? Well, if we perform a split into groups of size k, we get $\frac{P}{k}$ such groups. Of course, this must be an integer – i.e. k must be a divisor of P. This means we are looking for a sequence of numbers $c_0, c_1, c_2, \ldots, c_n$ such that $c_0 = P$, $c_{i+1} \mid c_i$ and $c_n = 1$.

A simple solution would be to generate all divisors of P (the possible values of c_1), attempt a split into those group sizes, and then recursively solve the problem for them. However, this would be too slow. As an example, if we take $P = 8\,086\,598\,962\,041\,600$, the sum of the square roots of its divisors is 6 636 882 083, so even finding only the ways to split the parliament into 2-level committees would be too expensive.

Instead, we will use the following lemma:

Exercise 17.5. Prove that divisibility is a transitive relationship; if $b \mid a$ and $c \mid b$, then $c \mid a$.

This means that the possible values of c_i, i.e. the transitive closure of divisibility of P, are the divisors of P. The problem reduces to finding the longest sequence of divisors of P such that each divisor is also a divisor of the previous divisor. By constructing the directed graph of all the divisors a with edges from a to its own divisors, we reduce the problem to finding the longest path in a DAG. Unfortunately, this too is slow – the previously mentioned P has 41 472 divisors, leaving us with about $\binom{41472}{2} = 859\,942\,656$ modulo operations to construct the graph.

What if we instead try to look at the entire sequence at once? We have $P = c_0$, $c_0 = c_1 q_1$, $c_1 = c_2 q_2, \ldots, c_{n-1} = c_n q_n$, $c_n = 1$ where $q_i > 1$. Inserting every equation into the previous one gives us $P = q_1 q_2 \cdots q_n$. Conversely, for each such choice of q_i, we can construct a valid sequence of c_i. Note that every q_i must have only trivial divisors, or we could replace it with two numbers a, b with $ab = q_i$, yielding a longer sequence with the same product.

Now comes the key insight. Let k be the smallest non-trivial divisor of P. This number only has trivial divisors too; if k had a non-trivial divisor $k' < k$, then $k' \mid P$ by the transitivity of divisibility, and so k' is a smaller divisor of P. We claim that k can always be chosen as one of the q_i by the following argument. Pick an i such that $k \nmid q_1 q_2 \cdots q_{i-1}$ and $k \mid q_1 q_2 \cdots q_i$ (one must exist since $k \nmid 1$ and $k \mid P$). By applying the following theorem to $p = k$, $a = q_1 \cdot q_{i-1}$ and $b = q_i$, we get that $k = q_i$.

Theorem 17.1 — Euclid's Lemma
If $p > 1$ and $b > 1$ have only trivial divisors, $p \mid ab$ and $p \nmid a$, then $p = b$.

Proof. We prove a slightly stronger statement (called Euclid's Lemma) instead; if $p > 1$ have only trivial divisors and $p \mid ab$, then either $p \mid a$ or $p \mid b$. This implies the original statement, since if $p \mid b$ and b only have trivial divisors, $p = 1$ or $p = b$ (but $p > 1$).

Consider the smallest p for which there exists a (smallest) a for which there exists

a b where the theorem is false. We now prove that this minimal counterexample gives rise to an even smaller counterexample.

First, a lacks non-trivial divisors. Otherwise, we can pick n, m such that $a = nm$ where $0 < n \leq m < a$. Substitution gives us $p \mid (nm)b = n(mb)$. Since $n < a$, we have either $p \mid n$ or $p \mid mb$, otherwise we find a smaller counterexample. We know that $p \nmid n$. Otherwise, as $n \mid a$, we get $p \mid a$ which we have assumed to be false. Therefore, $p \mid mb$. Because $m < a$, we again find that $p \mid m$ or $p \mid b$ (or we have a smaller counterexample). As $p \nmid b$ (by assumption), we get $p \mid m$. Again, as $m \mid a$ we get the contradiction $p \mid a$. This proves that a lacks non-trivial divisors.

Next, we assume $a < p$. Otherwise, consider $c = a - p > 0$. Since $p \mid ab$, we have $p \mid ab - pb = cb$ (see Exercise 17.6). Thus, $p \mid c$ or $p \mid b$ (c would otherwise be a smaller counterexample than a). We have assumed $p \nmid b$, so $p \mid c$, i.e. $p \mid a - p$. By the same exercise, this implies $p \mid a$, contrary to our assumption.

Finally, let n be such that $pn = ab$. Since a is smaller than p and have only trivial divisors, $a \mid pn$ implies $a \mid p$ or $a \mid n$ or a would be a smaller counterexample of the theorem. As p lacks non-trivial divisors, the latter must be true. This means there exists m such that $n = ma$. Inserting this gives us that $ab = pma$, or $b = pm$. But this means $p \mid b$, a contradiction.

Since the assumption of a smallest counterexample only lead to contradictions, we find that that no such counterexample exists, meaning the theorem is true. □

Exercise 17.6. Prove that if $a \mid b$ and $a \mid c$ then $a \mid b + c$.

With this, we are nearly there. As the sequence q_i is independent of order, we can let $q_1 = k$, or equivalently, choosing the largest possible divisor of P as c_1. How do we choose the remaining ones? Well, for c_2, the same argument says we should choose greatest possible divisor of $\frac{P}{c_1}$, and so on. Since it must also be a divisor of P, we can iterate through the smaller divisors (in descending order) and pick the first one that was also a divisor of $\frac{P}{c_1}$. Eventually, we reach $c_n = 1$.

A solution could look something like the following.

```
1: procedure Subcommittees(P)
2:     divisors ← Divisors(P)
3:     sort divisors in descending order
4:     ans ← 0
5:     for each d in divisors do
6:         if d divides P then
7:             ans ← ans + 1
8:             P ← d
9:     return ans
```

This solution only requires computing and iterating through all divisors of P, giving us a

$\Theta(\sqrt{P})$ solution. □

Problem 17.7.

Evening Out 2	eveningout2	
Multiplication Table	multtable	*(for 2 points)*

This result that divisors comes in pairs happens to give us some help in answering our next question, regarding the plurality of divisors. The above result gives us an upper bound of $2\sqrt{n}$ divisors of an integer n. We can do a little better, with $\approx n^{\frac{1}{3}}$ being a commonly used approximation for the number of divisors when dealing with integers which fit in the native integer types.[1] For example, the maximal number of divisors of a number less than 10^3 is 32, 10^6 is 240, 10^9 is 1 344, 10^{18} is 103 680.[2]

A bound we will find more useful when solving problems concerns the *average* number of divisors of the integers between 1 and n.

Theorem 17.2 — Average Number of Divisors
Let $d(i)$ be the number of divisors of i. Then,

$$\sum_{i=1}^{n} d(i) = \Theta(n \ln n)$$

Proof. There are between $\frac{n-i+1}{i}$ and $\frac{n}{i}$ integers between 1 and n divisible by i, since every i'th integer is divisible by i. Thus, the number of divisors of all those integers is bounded by

$$\sum_{j=1}^{n} \frac{n}{j} = n \sum_{j=1}^{n} \frac{1}{j} = O(n \ln n)$$

from above and

$$\sum_{j=1}^{n} \frac{n-j+1}{j} = n \sum_{j=1}^{n} \frac{1}{j} - n + \sum_{j=1}^{n} \frac{1}{j} \geq n \ln n - n + \ln n = \Omega(n \ln n)$$

from below[a].

□

[a] That $\sum_{i=1}^{n} \frac{1}{i} = \Theta(\ln n)$ is a standard result from single-variable calculus.

This proof also suggest a way to compute the divisors of all the integers $1, 2, \ldots, N$.

[1] In reality, the maximal number of divisors of the interval $[1, n]$ grows sub-polynomially, i.e., as $O(n^{\epsilon})$ for *every* $\epsilon > 0$.

[2] Sequence A066150 from OEIS: http://oeis.org/A066150.

Divisor Counts – divisorcounts

Count the number of positive divisors of every integer between 1 and N.

Solution. Solving the problem with the previous algorithm by computing the divisors for every single integer would yield a $\Theta(N\sqrt{N})$ algorithm. Instead, we flip the problem around. For each integer i, we find all the numbers divisible by i (in $\Theta(\frac{n}{i})$ time), which are $0i, 1i, 2i, \dots \lfloor \frac{n}{i} \rfloor i$. In total, this takes $\Theta(N \ln N)$ time, a quite decent improvement. □

The pseudo code of this method is very short but good to put to memory.

```
1: procedure CountDivisors(N)
2:     counts ← new int[N]
3:     for i from 2 up to i ≤ N do
4:         for j ∈ {2i, 3i, ...} up to j ≤ N do
5:             increment counts[j] by 1
```

The technique is called ***sieving*** and is a quite common number theoretical method. We use it again already in the next section.

Exercise 17.8. Assume that we instead only sieve with integers i up to $\sqrt{n}$ and use the fact that if i is a divisor of n, then $\frac{n}{i}$ is too, so that we can count two divisors at a time. Does this improve the time complexity of the algorithm?

Problem 17.9.
Organizator organizator

17.2 Prime Numbers

We regularly use divisibility as a tool to describe factorizations of an integer in various ways. For example, given the number 12, we could factor it as $2 \cdot 6$, or $3 \cdot 4$, or even further into $2 \cdot 2 \cdot 3$. This last factorization is special, in that no matter how hard we try, it cannot be factored further since 2 and 3 lack non-trivial divisors. It consists only of factors that are *prime numbers*.

Definition 17.2 — Prime Number
An integer $p \geq 2$ is called a ***prime number*** if its only positive divisors are 1 and p. The integers $n \geq 2$ that are *not* prime numbers are called ***composite numbers***. Note that 1 is neither prime nor composite.

Example 17.2 The first 10 prime numbers are $2, 3, 5, 7, 11, 13, 17, 19, 23, 29$.

We alluded to this definition several times in the last chapter, where we instead talked about numbers that only had positive divisors. For example, when referring to prime numbers by name Euclid's lemma becomes simpler: if a prime $p \mid ab$, either $p \mid a$ or $p \mid b$.

Problem 17.10.

Longest Prime Sum	longestprimesum
Shortest Composite Sum	shortestcompositesum

Let us start with the simple questions – how do we determine if a number is a prime? Using the knowledge from the previous section, this is simple. A number is prime if it has only trivial divisors, so we can use the algorithm for counting divisors and instead checking if the number has *any* divisor in $O(\sqrt{N})$ time.

Problem 17.11.

Primality	primality	*(for 1 point)*
Blackboard Numbers	primes2	

Exercise 17.12. Let $\pi(N)$ be the number of primes up to N. Given a list of those primes, show how to determine the primality of any integer up to N^2 in $O(\pi(N))$ time.

The result in Exercise 17.12 suggests that, after some precomputation, we can check primality faster than testing all possible divisors. To know how much faster, we need to know more about the number of primes.

There are an infinite number of primes. Euclid gave a simple proof in *Elements*. If $p_1, p_2, ..., p_q$ are the only primes, then $P = p_1 p_2 \dots p_q + 1$ is not divisible by any prime number p_i, since it must then also divide $P - p_1 p_2 \dots p_q = 1$ (and by extension has no divisors but the trivial ones), so it is not composite. This means P is a prime, but it is greater than all the primes in the list, a contradiction. More relevant is instead the density of primes, since that is what determines how $\pi(N)$ relates to N.

Theorem 17.3 — Prime Number Theorem

$$\pi(N) \sim \frac{N}{\ln N}$$

i.e. the density of prime numbers in the interval $[1 \dots N]$ is $\sim \frac{1}{\ln N}$. A consequence is that the n'th prime number is approximately $n \ln n$.

The proof requires a lot of deep number theory, so we will not show it here. The bound means by that precomputing primes and then using that list to check primality you only gain a logarithmic factor. For values relevant in practice the approximation is close to reality: the number of primes below 10^3 is 168, below 10^6 is 78 498, and below 10^9 is approximately $51 \cdot 10^6$.

Based on the Prime Number Theorem, one might have the reasonable suspicion that prime numbers shouldn't be that far apart. The Prime Number Theorem states that the

average distance between primes is $\frac{1}{\ln N}$, but of course the maximum gap may be longer. For all integers up to 10^9, the maximum gap is 282, and for 10^{18} it is 1442.

Problem 17.13.
Enlarging Hash Tables enlarginghashtables

A very lax upper bound on the gaps that is occasionally useful is the following one.

> **Theorem 17.4 — Bertrand's Postulate**
> For all $n \geq 2$, there exists a prime p where $n < p < 2n$.

We omit the elementary but technically complex proof, since it is of little interest to us.

Prime Time – primetime

By Jon Marius Venstad. Nordic Collegiate Prog. Contest 2011. CC BY-SA 3.0. Shortened.

Odd, Even and Ingmariay are playing a game. They start with an arbitrary positive integer and take turns either adding 1 or dividing by a prime (assuming the result is still an integer). Once they reach 1, they each gets points corresponding to the smallest of the numbers their move resulted in. If a player could make no move, their score is instead equal to the starting integer. They all play such that they minimize their own score. If several possible moves would result in the same score for the player, they have agreed to make the one that produces the smallest integer. They play in the order Odd → Even → Ingmariay → . . . , but alternate who starts the round.

In total, they play $n \leq 1000$ rounds of the game. Given the starting integers (between 1 and 10 000) for all rounds, output the final scores of the three players.

Solution. The game theoretic solution of the problem would be to construct the game graph of all the integers with edges between possible transitions. One could then compute the score a player would get for moving to a certain integer in the graph in the way described in Section 16.5. Unfortunately, the game graph contains possible loops such as $2 \to 3 \to 4 \to 2$. We can eliminate those loops with some number theoretical insights together with the tie-breaking rules the players use when picking moves.

Let us investigate the behavior of the players more closely. If a player is presented with a prime number, they clearly will divide with it to end the game and get 1 point. In other cases, they may either add 1 or divide away a prime. This means we never want to move from a prime p to $p + 1$.

It turns out that removing these moves makes the game acyclic. Assume to the contrary that we currently are at a number a between two consecutive primes p_k and p_{k+1} and have a sequence of moves that takes us back to a. The next sequence of moves will be to add 1 until we hit an integer $p_k < b \leq p_{k+1}$ (remember we never want to go to $p_{k+1} + 1$) and dividing some prime p_i away. But $\frac{b}{p_i} \leq \frac{b}{2} \leq \frac{p_{k+1}}{2}$. By Bertrand's Postulate, $\frac{p_{k+1}}{2} < p_k$, so the new result will be less than p_k. Since we never make the transition $p_k \to p_k + 1$ we can never reach a again.

What remains is to compute the transitions from each integer $1 \leq a \leq N$. By precomputing all the primes up to 10 000 (check the primality of each one in $O(\sqrt{k})$ time), we can afford to test whether all of them are divisors of each a. In total, this is on the order of $\frac{10^4 \cdot 10^4}{\ln 10^4} \approx 10^7$ edges in the game graph, which is a reasonable number. □

Factorizations

Since the prime numbers have no other divisors besides the trivial ones, a factorization consisting only of prime numbers is special.

Definition 17.3 — Prime Factorization

The ***prime factorization*** of a positive integer n is a factorization of the form

$$p_1^{e_1} \cdot p_2^{e_2} \cdot \cdots \cdot p_k^{e_k}$$

where p_i are all distinct primes.

Example 17.3 The prime factorization of 228 is $2 \cdot 2 \cdot 3 \cdot 19$.

Note that in the definition, we spoke of *the* prime factorization. This factorization is indeed unique, except for a reordering of the p_i. It may be "intuitively obvious" that this is the case, but that is misguided notion. A proof can be constructed using Euclid's Lemma from the previous section (p. 308).

Theorem 17.5 — Existence and Uniqueness of Prime Factorizations

There exists a unique factorization of a positive integer N into prime numbers.

Proof. The existence part is a simple proof by induction. Assume that all integers up to $N-1$ has a prime factorization. If N is a prime, then N is a prime factorization of itself. Otherwise, it has a non-trivial divisor, so we can write $N = ab$ with $1 < a \leq b < N$. By the induction hypothesis, a and b have prime factorizations. Concatenating the two factorizations gives us a prime factorization of N. Thus, by induction, all positive integers have prime factorizations.

Next, the uniqueness. We prove this too by induction. Our base case is $N = 1$ which has the empty product as unique prime factorization. Assume that N has two distinct prime factorizations $N = p_1 p_2 \cdots p_k = q_1 q_2 \cdots q_l$, but all integers up to $N-1$ has only one. Consider the prime p_k. Since it divides the left side, it must also divide the right side by the following argument. Let i be such that $Q = q_1 \cdots q_{i-1}$ is not divisible by p_k, but Qq_i is. Such an i exists since when $i = 1$, the product Q is 1 (the empty product) which is not divisible by p_1, and with $i = l$ we get $Q = N$ which is divisible by p_1.

Then, by our version of Euclid's Lemma, since p_1 and q_i are primes, $p \mid Qq_i$ but $p \nmid Q$, we have $p_1 = q_i$. Without loss of generality, we can assume that $i = 1$. If we divide away this factor, we get that $p_2 \cdots p_k = q_2 \cdots q_l$ are both prime factorizations of $\frac{N}{p_1} < N$, so by the induction hypothesis they are the same. That means the original prime factorizations were also the same, a contradiction. Thus, N too has a unique prime factorization, completing the proof. □

Factorization

Given an integer N, compute its prime factorization.

Solution. The simplest solution is to extend the method used to test primality. An integer N can have at most one prime in its factorization that exceeds $\sqrt{N}$, since their product otherwise would exceed N. Looping over all possible prime divisors up to $\sqrt{N}$ and factoring them out from N is sure to find all prime factors, except for possibly a single one that was larger than $\sqrt{N}$. The algorithm is called ***trial division***.

```
1: procedure TrialDivision(N)
2:     primes ← new list
3:     for i from 2 up to i^2 ≤ N do
4:         while N mod i = 0 do
5:             primes.add(i)
6:             N ← N/i
7:     if N ≠ 1 then
8:         primes.add(N)          ▷ N may have had a single prime factor > √N
9:     return primes
```

□

Exercise 17.14. In the TrialDivision algorithm, N is being modified in the loop when a new prime is found. Is it a problem to use the new, updated N in the $i^2 \leq N$ check in the loop? Is the time complexity the same?

Problem 17.15.

A List Game listgame

The multiplicity of a prime number is a recurring topic of problems. To discuss it more succinctly, the following notation is often used.

Definition 17.4 For integers a, b, k, we say that a^k ***divides*** b ***exactly*** if $a^k \mid b$ but $a^{k+1} \nmid b$. We use the notation $a^k \parallel b$ for this.

Factorial Power – factorialpower

Given integers $2 \leq n, m \leq 10^{14}$, determine the k for which $n^k \mid\mid m!$.

Solution. To start, we first connect divisibility with prime factorizations. Let n have the prime factorization $n = p_1^{e_1} \cdot p_2^{e_2} \cdots p_l^{e_l}$. That d is a divisor of n is the same as having a prime factorization $d = p_1^{e_1'} \cdot p_2^{e_2'} \cdots p_l^{e_l'}$, where $0 \leq e_i' \leq e_i$. It follows from the uniqueness of the prime factorization and the fact that $n = dq$ for some integer q. That the converse – any number of this form is a divisor of n – is true, is also useful.

The point of this little sidebar is that we can now decompose the problem into only looking at prime values of n. The exponent laws gives us that $n^k = p_1^{ke_1} \cdot p_2^{ke_2} \cdots p_l^{ke_l}$, so we are looking for the largest k such that ke_i does not exceed the power of p_i in $m!$. Thus, after factoring n the problem is reduced to determining how many times all the p_i divides $m!$. This equals

$$\left\lfloor \frac{m}{p_i} \right\rfloor + \left\lfloor \frac{m}{p_i^2} \right\rfloor + \left\lfloor \frac{m}{p_i^3} \right\rfloor + \dots$$

Exercise 17.16. Prove the above formula.

Since n can have at most $\log_2(n)$ prime factors, and all terms after the $\log_2(m)$ first ones are zero, the complexity of the computation is $O(\sqrt{n} + \log(n)\log(m))$. □

Problem 17.17.
Perfect Pth Powers perfectpowers

For the next problem, we will show a version of the divisor sieve that can factor integers.

Product Divisors – productdivisors

Given a sequence of $n \leq 1\,000\,000$ integers $a_1, a_2, \dots a_n \leq 10^6$, compute the number of divisors of $A = \prod_{i=1}^{n} a_i$ modulo $10^9 + 7$.

Solution. Again, we use the prime factorization interpretation of divisors. Let $A = \prod_{i=1}^{k} p_i^{e_i}$ where p_i are distinct primes. A simple combinatorial argument lets us count the number of divisors of A. A divisor of A is of the form $\prod_{i=1}^{k} p_i^{e_i'}$ where $e_i' \leq e_i$. All the e_i' can take any integer value between 0 and e_i to fulfill this condition. This gives us $e_i + 1$ choices for the value of e_i'. Since each e_i' is independent, there are a total of $(e_1+1)(e_2+1)\dots(e_k+1)$ numbers of this form, and thus divisors of A by the multiplication principle.

We are left with the problem of determining the prime factorization of A. This is tantamount to computing the prime factorization of *every* integer between 1 and 10^6, since we could have $a_i = i$ for $i = 1 \dots 10^6$. Once this is done, we can go through the sequence a_i and tally up all primes in their factorization. Since an integer m has at most $\log_2 m$ prime factors, this step is bounded by approximately $n \log_2 10^6$ operations. Then, how

do we factor all integers in $[1 \dots 10^6]$? The solution is yet another variation of the sieving method.

The divisor counting sieve can easily be modified to keep track only of whether integers are prime or not. In fact, it only needs to sieve over primes this, since any non-prime has prime divisors.

```
1: procedure FindPrimes(N)
2:     isPrime ← new list[N + 1] of true
3:     for i from 2 up to N do
4:         if isPrime[i] then
5:             for j ∈ {2i, 3i, ...} up to j ≤ N do
6:                 isPrime[j] ← false
```

This particular sieve is called the *Sieve of Eratosthenes*.

Adapting it to also factoring integers is simple. The inner loop iterates over all integers that have i as prime factor, so we just need to also count the multiplicity of i in j in the loop (don't forget to factor the primes themselves).

The last thing that remains is combining these steps – finding the factors of all integers up to N, counting the number of times each prime appears in the numbers a_i, and then using the divisor formula count the number of divisors.

It is not as obvious what the complexity of this solution is. Clearly it is at most $O(n \log n)$, since it is linear in the number of prime factors (with multiplicity) of all integers in an interval, and this is less than the number of divisors. We do a bit less work though, since we only sieve using primes rather than all integers. If we count the total work done (i.e. the number of prime factors with multiplicity over an entire interval), we get

$$\sum_{p \leq n} \left(\frac{n}{p} + \frac{n}{p^2} + \dots \right) = n \sum_{p \leq n} \left(\frac{1}{p} + \frac{1}{p^2} + \dots \right)$$

per the previous problem. The sum can be bounded using the formula for sums of infinite geometric series

$$\sum_{i=0}^{\infty} \frac{1}{p^i} - 1 = \frac{1}{1 - \frac{1}{p}} - 1 = \frac{p}{p-1} - 1 = \frac{1}{p-1} = \Theta\left(\frac{1}{p}\right)$$

so that the total work is bounded by

$$\Theta\left(n \sum_{p \leq n} \frac{1}{p} \right) = \Theta(n \ln \ln n)$$

by the well-known number theoretic bound[3] $\sum_{p \leq n} \frac{1}{p} = \Theta(\ln \ln n)$. A small difference, but nonetheless asymptotically better than the divisor sieve! The version of Eratosthenes sieve that does not outright factor integers have the same time complexity as well. □

[3] This can be proven by a slightly better approximation of the Prime Number Theorem and basic calculus.

Problem 17.18.

Prime Count	primecount	*(for 2 points)*
Non-Prime Factors	nonprimefactors	
Flower Garden	flowergarden	
Jazz it Up!	jazzitup	

Competitive Tip

When using the Sieve of Eratosthenes, we can save quite a bit of memory by using a bitset instead since we only store a boolean state per number (whether it is prime or not). This gives us slightly better cache behavior, improving the performance in real terms.

Problem 17.19.

Prime Count	primecount	*(for 3 points)*

17.3 The Euclidean Algorithm

The Euclidean algorithm is one of the oldest known algorithms, dating back to Greek mathematician Euclid who wrote of it in his mathematical treatise *Elements*. It regards those numbers which are divisors of two different integers, and extends into integer equations of the form $ax + by = c$.

Definition 17.5 — Greatest Common Divisor

We call an integer d dividing both of the integers a and b a ***common divisor*** of a and b.

The greatest such integer is called the ***greatest common divisor***, or *GCD* of a and b. This number is denoted $\gcd(a, b)$, or with the shorthand (a, b) when clear from the context.

Example 17.4 12 and 42 have divisors 1, 2, 3, 4, 6, 12 and 1, 2, 3, 6, 7, 14, 21, 42. Their shared divisors are 1, 2, 3, 6, so their greatest common divisor is 6.

We warm up with some simple properties of the GCD.

Theorem 17.6 — Properties of the GCD

Let a, b, c be non-negative integers. Then

$$(a, 0) = a \tag{17.1}$$

$$(a, a) = a \tag{17.2}$$

$$(a, b) \leq \max(a, b) \tag{17.3}$$

$$(ac, bc) = c \cdot (a, b) \tag{17.4}$$

$$(a, b) \mid (a, bc) \tag{17.5}$$

When $(a, c) = 1$ we call them ***relatively prime***, which we denote as $a \perp c$. If $a \perp c$,then

$$(a, bc) = (a, b) \tag{17.6}$$

Proof. We give a proof for the last equation – the others are good exercises to get acquainted with the GCD.

It is sufficient to prove the equation for prime c, since that is enough for the following proof by contradiction for composite c. Let c be the smallest counterexample to the equation for some a, b. Since c is composite, $c = pc'$ for some prime p and $c' > 1$. Since $p \mid pc' = c$ and $(a, c) = 1$, Eq. 17.5 gives that $(a, p) = 1$. Together with the assumption that the equation holds for primes, we have $(a, bc) = (a, (bc')p) = (a, bc')$. In a similar manner we can show that $(a, c') = 1$. Since we assume that c was the smallest counterexample of the equation for a, b, we can apply it to get $(a, bc') = (a, b)$. Combining the two equality chains shows us that $(a, bc) = (a, b)$.

For the prime case, let $d = (a, b)$. Then, $(a, bc) = d(\frac{a}{d}, \frac{bc}{d})$ by Equation 17.4. Proving the equation is now equivalent to $(\frac{a}{d}, \frac{bc}{d}) = 1$. Assume that the equality is false. Then there must be some prime p that divides $\frac{a}{d}$ and $\frac{bc}{d}$. The first condition implies $p \mid a$. However, since $(a, c) = 1$ we can not have $p \mid c$, or else they would share a divisor greater than 1. This means $p \mid \frac{b}{d}$. Then pd is an even greater common divisor of a and b, a contradiction. □

Exercise 17.20. Prove Equations 17.1-17.5.

Before we start looking into how to actually compute the greatest common divisor, we take a detour into the land of number theoretic sums to also get some practice and understanding of what the GCD actually means.

GCD Sum – gcdsum

Compute

$$\sum_{i|N} \sum_{j|N} \gcd(i, j)$$

where $N \leq 10^{14}$ is a given integer.

Solution. Now and then problems consist of computing some number theoretic sum. There is a number of different techniques involved in this, so we show two different solutions.

Let us first try to transform the sum into something simpler. We don't even know how to compute the gcd of two numbers quickly yet, so it makes sense to attempt to simplify that term. This approach is also supported by $\gcd(i, j)$ being a non-trivial term

that requires computation to figure out, but that we actually *know all the values it will assume*. Just from the definition, we understand that $\gcd(i,j) \mid i$, and $i \mid N$, so $\gcd(i,j)$ is a divisor of N too. Picking $i = j = d$, shows us that $\gcd(i,j) = d$ assumes the value of exactly the divisors of N.

This allows a common reformulation of sum problems: if we fix the possible values of $\gcd(i,j) = d$ one at a time, for how many pairs $i \mid N$, $j \mid N$ does the GCD term actually assume this value? For each divisor, we then compute the contribution $d \cdot k_d$ to the sum, where k_d is the number of pairs i, j with $\gcd(i,j) = d$.

Evaluating k_d for a fixed $(i,j) = d$ is easy after reducing away the d. We can let $i' = \frac{i}{d}$, $j' = \frac{j}{d}$, and $N' = \frac{N}{d}$. Now, a small divisibility fact to assist us: divisibility is unaffected by integer scaling, i.e. $a \mid b$ is equivalent to $ac \mid bc$ (a one-line proof using the definition). This fact has two consequences: $(i,j) = d$ is equivalent to $(i',j') = 1$, and $i, j \mid N$ is equivalent to $i', j' \mid N'$. This means that we can instead count the pairs $i', j' \mid N'$ with $(i',j') = 1$.

The remainder of the solution is now a combinatorial argument. Let $\prod_{p_i \mid N'} p_i^{e_i}$ be the prime factorization of N'. Since i' and j' are divisors of N' but themselves share no factor, there are three cases for each p_i: it divides i' to some power, j' to some power, or neither (a consequence of Euclid's Lemma). In the first two cases we get to choose how many of the e_i factors to include in the divisor. In the third case, we only have a single choice. Thus, there are in total $2e_i + 1$ choices for each factor which we multiply together for all primes. The sum has now been transformed to its final, easily computable form:

$$\sum_{d \mid N} \left(d \prod_{p_i^{e_i} \mid\mid \frac{N}{d}} (2e_i + 1) \right).$$

A note on implementation: do not compute the prime factorization of each $d \mid N$ using e.g. trial division – this is too expensive. Instead, factor N and then construct all subsets of prime factors of N recursively, one prime factor at a time.

The second approach involves a quite different way of computing the sum. It involves studying the function we are computing, and figuring out how it is affected if we isolate one of the prime powers dividing the argument of the function (N). This is a common theme in number theory, where many sums and functions are easy to compute for prime powers, with results hopefully easy to combine!

Let p be a prime where $p^k \mid\mid N$ and $N' = \frac{N}{p^k}$. Then, we can rewrite our sum as

$$\sum_{a=0}^{k} \sum_{b=0}^{k} \left(\sum_{i \mid N'} \sum_{j \mid N'} \gcd(i \cdot p^a, j \cdot p^b) \right)$$

By Equation 17.4, we can factor out $\min(p^a, p^b)$ from the innermost term. Assume for the purpose of demonstration that $a \le b$. Then, the sum simplifies to

$$\sum_{a=0}^{k} \sum_{b=0}^{k} \left(\sum_{i \mid N'} \sum_{j \mid N'} p^a \gcd(i, j \cdot p^{b-a}) \right)$$

By assumption $p \nmid i$, so Equation 17.6 tells us that $\gcd(i, j \cdot p^k) = \gcd(i, j)$. Further simplification becomes possible:

$$\left(\sum_{a=0}^{k}\sum_{b=0}^{k} p^{\min(a,b)}\right)\left(\sum_{i|N'}\sum_{j|N'} \gcd(i,j)\right)$$

The observant reader may notice that the left factor in the above product happens to be same sum you'd get if $N = p^k$, since $\gcd(p^a, p^b) = p^{\min(a,b)}$! It's apparently enough to compute the sum for all prime factors of N and multiply answers together. This is not uncommon – in Section 17.5 we study more functions like this.

Now, the only thing that remains is to evaluate this particular sum for each prime divisor. Luckily k and the number of primes in a number are both very small ($\leq \log_2(N)$), so the sums can be evaluate with nested loops after N has been factorized. □

Finally, we get to the big question. How do we compute the greatest common divisor of two integers?

Greatest Common Divisor – gcd

Given two non-negative integers a and b, compute (a, b).

We already know of a $\Theta(\sqrt{a}+\sqrt{b})$ algorithm to compute (a, b), namely to enumerate *all* divisors of a and b. A new identity unlocks the much faster Euclidean algorithm.

$$(a, b) = (a, b - a) \tag{17.7}$$

We can prove the equality by proving an even stronger result – that *all* common divisors of a and b are also common divisors of a and $b - a$. Assume d is a common divisor of a and b, so that $a = da'$ and $b = db'$ for integers a', b'. Then $b - a = db' - da' = d(b' - a')$, with $b' - a'$ being an integer, is sufficient for d also being a divisor of $b - a$. The converse is shown in a similar way. Hence the divisors of a and b are the same as the divisors of a and $b - a$. In particular, their largest common divisor is the same. The application of these identities yield a recursive solution to the problem. If we wish to compute (a, b) where a, b are positive and $a \leq b$, we reduce the problem to a smaller one by instead computing (a, b), we compute $(a, b - a)$. This gives us a smaller problem, in the sense that $a + b$ decreases. Since both a and b are non-negative, this means we must at some point arrive at the situation where $b = 0$. Equation 17.1 tells us the GCD is then a.

One simple but important step remains before the algorithm is useful. Note how computing $(1, 10^9)$ requires about 10^9 steps right now, since we will do the reductions $(1, 10^9 - 1), (1, 10^9 - 2), (1, 10^9 - 3)$... The fix is easy – the repeated application of subtraction of a number a from b while $b \geq a$ is the modulo operation, meaning

$$(a, b) = (a, b \bmod a)$$

This last piece of our Euclidean puzzle completes our algorithm, and gives us a remarkably short algorithm. Note the recursive invocation to $(b \bmod a, a)$ to ensure that $a \leq b$.

```
1: procedure GCD(A, B)
2:     if A = 0 then
3:         return B
4:     return GCD(B mod A, A)
```

Problem 17.21.
Temperature Confusion temperatureconfusion

Competitive Tip

The Euclidean algorithm exists as the built-in function `__gcd(a, b)` for most C++ compilers.

Whenever dealing with divisors in a problem, the greatest common divisor can be a useful tool. This is the case in the next problem, where we also look closer at the prime factorization of the GCD.

Granica – granica

Croatian Open Competition in Informatics 2007/2008, Contest #6

Given integers $2 \leq n \leq 100$ integers $a_1, a_2, ..., a_n$, find all those numbers d such that upon division by d, all of the numbers a_i leave the same remainder.

Solution. What does it mean for two numbers a_i and a_j to have the same remainder when dividing by d? Letting this remainder be r we can write $a_i = dn + r$ and $a_j = dm + r$ for integers n and m. Thus, $a_i - a_j = d(n - m)$ so that d is divisor of $a_i - a_j$! This gives us a necessary condition for our numbers d. Is it sufficient? If $a_i = dn + r$ and $a_j = dm + r'$, we have $a_i - a_j = d(n - m) + (r - r')$. Since d is a divisor of $a_i - a_j$ it must be a divisor of $d(n - m) + (r - r')$ too, meaning $d \mid r - r'$. As $0 \leq r, r' < d$, we have that $-d < r - r' < d$, implying $r - r = 0$ so that $r = r'$ and both remainders were the same after all.

The answer is then the set of common divisors of all numbers $a_i - a_j$. We claim that this set is (even for the case of only two numbers) the *divisors of their greatest common divisor*. Intuitively true for some, but to prove it we take aid in the prime factorizations of divisors. A divisor of some integer

$$n = p_1^{e_1} \cdots p_k^{e_k}$$

is of the form

$$d = p_1^{e_1'} \cdots p_k^{e_k'}$$

where $0 \leq e_i' \leq e_i$. Then, the requirement for d to be a **common** divisor of n and another number

$$m = p_1^{f_1} \cdots p_k^{f_1}$$

is that $0 \leq e_i' \leq \min(f_i, e_i)$. The converse, that a number with this property is indeed a common divisor of n and m should be clear.

The largest such number is attained when $e_i' = \min(f_i, e_i)$ giving us the GCD. This also explains why all common divisors must be divisors of the GCD.

Using this interpretation of the GCD, we can extend the result to finding the GCD d of a sequence $b_1, b_2, \ldots$. Consider any prime p, such that $p^{q_i} \parallel b_i$. Then, we must have $p^{\min(q_1, q_2, \ldots)} \parallel d$. This operation is exactly what the GCD algorithm does for two numbers. Since $\min(q_1, q_2, \ldots) = \min(q_1, \min(q_2, \ldots))$, we can use the recursion formula $d = \gcd(b_1, b_2, \ldots) = \gcd(b_1, \gcd(b_2, \ldots))$, simplest implemented in a loop:

```
1: procedure MultiGCD(sequence A)
2:     gcd ← 0
3:     for each a ∈ A do
4:         gcd ← GCD(gcd, a)
5:     return gcd
```

Finally, we need to find all the divisors of the GCD to arrive at the answer. □

Problem 17.22.
Diagonal Cut diagonalcut

A complementary concept is the *least common multiple*.

Definition 17.6 — Least Common Multiple
The ***least common multiple*** of integers a and b is the smallest positive integer m such that $a \mid m$ and $b \mid m$.

Example 17.5 The multiples of 12 are 12, 24, 36, 48, 60, ... The multiples of 10 are 10, 20, 30, 40, 50, 60, ... The least common multiple of the numbers is 60.

Given a and b, ab is clearly a common multiple, but it doesn't have to be the smallest. Since $a \mid m$, we have that $m = ak$. The question is basically what extra factors we *must* add to a in the form of k in order to have $b \mid m$. Previously, we have determined that the condition for being a divisor is that when $p^k \parallel b$ is one of the primes dividing b, then $p^k \mid m$ has to hold. This basically means that we have to add whatever factors to a that are additionally present in b. For example, since $10 = 2 \cdot 5$ and $12 = 2 \cdot 2 \cdot 3$, we need to add an additional factor 2 and 3 to 10 to make a common multiple – $2 \cdot 2 \cdot 3 \cdot 5 = 60$

To compute the LCM easily, note that a multiple m of an integer a with prime factorization

$$a = p_1^{e_1} \cdots p_k^{e_k}$$

must be of the form

$$m = p_1^{e_1'} \cdots p_k^{e_k'}$$

where $e_i \leq e_i'$.

Thus, if m is to be a common multiple of a and another integer

$$b = p_1^{f_1} \cdots p_k^{f_1}$$

it must hold that $\max(f_i, e_i) \leq e_i'$, with $e_i' = \max(f_i, e_i)$ giving us the smallest such multiple. Since $\max(e_i, f_i) + \min(e_i, f_i) = e_i + f_i$, we get that $\operatorname{lcm}(a, b) \cdot gcd(a, b) = ab$. This gives us the formula $\operatorname{lcm}(a, b) = \frac{a}{\gcd(a,b)} b$ to compute the LCM. The order of operations is chosen to avoid overflows in computing the product ab.

As for the GCD of multiple integers, it holds that

$$\operatorname{lcm}(a, b, c, \dots) = \operatorname{lcm}(a, \operatorname{lcm}(b, \operatorname{lcm}(c, \dots)))$$

Problem 17.23.

Smallest Multiple smallestmultiple

GCD and LCM – gcdandlcm

Given that $\gcd(a, b) = x$ and $\operatorname{lcm}(a, b) = y$, where $1 \leq x, y \leq 10^{14}$ determine the possible pairs (a, b).

Solution. Without loss of generality, assume $a \leq b$. By the LCM formula, we have that $a \mid y$. That means we can test all possibilities of a. Furthermore, the formula gives that $ab = xy$, so that once we fix a we can compute the corresponding b easily. Then, we test whether $\gcd(a, b) = x$, and if so, add it to our list of answers. □

Problem 17.24.

Das Blinkenlights dasblinkenlights

Doodling doodling

The Extended Euclidean Algorithm

Next up is the ***extended Euclidean algorithm***. It is a way to solve certain integer equations.

Linear Diophantine Equation

Given integers a, b, find an integer solution x, y to

$$ax + by = (a, b).$$

It is not obvious that a solution exists. Let $S = \{ax + by \mid x, y \text{ integers}\}$. These numbers are called the ***linear combinations*** of a and b. S is closed under addition and negation (and thus also subtraction and multiplication). As divisibility is also closed under these

operations, all numbers of the form $ax + by$ must be multiples of (a, b). We might then stumble upon the (correct) hypothesis that the set contains (a, b) itself. Assume that d is the smallest multiple of (a, b) in S. Then $a - d\lceil \frac{a}{d} \rceil = a \bmod d \in S$, since it is closed under subtraction and multiplication. Similarly, $b \bmod d \in S$. As $0 \leq a \bmod d < d$, we must have $a \bmod d = 0$ and $b \bmod d = 0$ as d was the smallest element of S. However, this is equivalent to $d \mid a$ and $d \mid b$, so $d \mid (a, b)$, forcing $d = (a, b)$ since d is a positive multiple of (a, b).

This proof might remind you somewhat of the Euclidean algorithm. The proof and the algorithm hide within them a method to write (a, b) as a linear combination of a and b. Remember that during the computation of the GCD, we repeatedly used that $(a, b) = (b, a \bmod b)$. Since $a \bmod b$ is a linear combination of a and b, it seems as if the numbers (a, b) during the computation of the GCD *always* are linear combinations of a and b. The algorithm concludes at $(d, 0)$, at which point $d = (a, b)$. If we only kept track of which linear combination that was equal to d, we would be able to construct a solution to $ax + by = (a, b)$. Let us try this with an example, where we use $[x, y]$ to denote the number $ax + by$.

Example 17.6 — Extended Euclidean algorithm
Consider the equation $15x + 11y = (15, 11) = 1$.

Performing the Euclidean algorithm on these numbers we find that

$$(15, 11) = ([1, 0], [0, 1]) = (11, 15 \bmod 11) =$$

$$(11, 15 - 1 \cdot 11) = ([0, 1], [1, 0] - 1 \cdot [0, 1]) =$$

$$(11, 4) = ([0, 1], [1, -1]) = (4, 11 \bmod 4) =$$

$$(4, 11 - 2 \cdot 4) = ([1, -1], [0, 1] - 2[1, -1]) =$$

$$(4, 3) = ([1, -1], [-2, 3]) = (3, 4 \bmod 3) =$$

$$(3, 4 - 1 \cdot 3) = ([-2, 3], [1, -1] - [-2, 3]) =$$

$$(3, 1) = ([-2, 3], [3, -4]) = (1, 3 \bmod 1) =$$

$$(1, 3 - 3 \cdot 1) = ([3, -4], [-2, 3] - 3[3, -4]) =$$

$$(1, 0) = ([3, -4], [-11, 15])$$

Verifying the results, $15 \cdot 3 + 11 \cdot (-4) = 45 - 44 = 1$.

Exercise 17.25. Find an integer solution to the equation $24x + 44y = 4$.

The solution is easily coded either recursively or iteratively. We show the latter here since it is almost exactly how we showed the algorithm in the example.

```
1: procedure ExtendedEuclidean(a, b)
2:     x_a, y_a = 1, 0
3:     x_b, y_b = 0, 1
4:     while b ≠ 0 do
5:         t = ⌊a/b⌋
6:         x_a, x_b = x_b, x_a − t · x_b
7:         y_a, y_b = y_b, y_a − t · y_b
8:         a, b = b, a − t · b
       return (x_a, y_a)
```

This gives us a single solution. Finding the others is not much harder. Let $a' = \frac{a}{(a,b)}$ and $b' = \frac{b}{(a,b)}$. Given two solutions

$$ax_1 + by_1 = (a, b) \qquad ax_2 + by_2 = (a, b)$$

we can first factor out (a, b) to get

$$a'x_1 + b'y_1 = 1 \qquad a'x_2 + b'y_2 = 1$$

Subtracting the equations from each other gives us that

$$a'(x_1 - x_2) + b'(y_1 - y_2) = 0 \Leftrightarrow a'(x_1 - x_2) = b'(y_2 - y_1)$$

Because $(a', b') = 1$ we have $b' \mid x_1 - x_2$. Then there exists k such that $x_1 - x_2 = kb'$, so $x_1 = x_2 + kb'$. Inserting this gives us

$$a'(x_2 + kb' - x_2)) = b'(y_2 - y_1)$$

$$a'kb' = b'(y_2 - y_1)$$

$$a'k = y_2 - y_1$$

$$y_1 = y_2 - ka'$$

Thus, any solution must be of the form

$$(x_1 + k\frac{b}{(a,b)}, y_1 - k\frac{a}{(a,b)}) \text{ for } k \in \mathbb{Z}$$

It is easily verified that any k also gives us a solution to this. This result is called ***Bezout's identity***.

Exercise 17.26. Assume that $(a, b) = 1$. How can we find all the solutions to $ax + by = c$ for any integer c?

Exercise 17.27. For what a, b and c does $ax + by = c$ not have solutions?

Problem 17.28.

So You Like Your Food Hot?	soyoulikeyourfoodhot
Jug Hard	jughard

Generalized Knights – generalizedknights

A *generalized knight* is a special chess piece that moves around on an infinite chessboard. For two given integers $1 \leq a \neq b \leq 10^9$, it moves by first choosing one of the four cardinal directions and moves a steps, and then chooses one of the two orthogonal cardinal directions and moves b steps (for example first up and then left or right, or first left and then up or down). Compute the minimum number of moves the knight needs to move from $(0, 0)$ to (x, y) where $1 \leq x, y \leq 10^9$.

Solution. A common approach to this kind of problem is to decompose it into the two axes individually, and then resolve any interdependencies afterwards. There are in total 8 possible moves; $(\pm a, \pm b)$ and $(\pm b, \pm a)$. For a single axis, we instead have four moves: $\pm a, \pm b$. Using these moves, the knight can only reach positions equal to $as + bt$ for some s, t in each direction. First of all, this tells us that x and y must be divisible by (a, b), since $as + bt$ always is. For simplicity, we can therefore divide a, b, x and y with (a, b) so that $(a, b) = 1$.

Bezout's identity suggests that we should start by finding any s and t such that $x = as+bt$ using Euclid's extended algorithm. Given such s and t, we know that using $|s|$ a-moves and $|t|$ b-moves is enough for this axis. All possible solutions are $(s + kb, t - ka)$ for integer k.

Let us now combine the axes. The X axis requires $x_a = |s_x + k_x b|$ a moves and $x_b = |t_x - k_x a|$ b moves, while the Y axis needs $y_a = |s_y + k_y b|$ a moves and $y_b = |t_y - k_y a|$ b moves. To be able to pair up the moves on each individual axis, the number of a-moves on one axis must have the same parity as the number of b moves on the axis. This is weaker than what might at first expect, i.e. that the number of moves must be equal. If $x_a > y_b$, we can pair the first y_b moves of these types up into $(\pm a, \pm b)$ moves, and then alternate having $+b$ and $-b$ for the remaining moves. This means that as long as $x_a - y_b \bmod 2 = 0$ and $y_a - x_b \bmod 2 = 0$, we need at most $\max(x_a, y_b) + \max(x_b, y_a)$ moves.

Going forward, we'll show that there is an optimal solution where certain properties hold by modifying an arbitrary solution without making it worse. Assume that $s_x + k_x b \geq 0$ and $t_x - k_x a \leq 0$, so that $x_a = s_x + k_x b$ and $x_b = k_x a - t_x$. Then we can clearly improve a solution by decreasing x_b to the point where $t_x - k_x a \geq 0$. Consequently, it must be optimal for $s_x + k_x b$ and $t_x - kx_a$ to have the same sign (and similarly for $s_y + k_y b$ and $t_y - k_y a$), or one of them being ≈ 0. Furthermore,

Now, assume that in an optimal solution, we have $x_a + b \leq y_b$. Then, increasing k_x until $y_b - b < x_a \leq y_b$ does not change the term $\max(x_a, y_b)$, but it may decrease $\max(x_b, y_a)$, so it's a harmless change. We can apply the same reasoning for all four variables, for the conclusion that we can let $x_a \approx y_b$ and $x_b \approx y_a$.

The case where some of x_a, x_b, y_a and y_b are ≈ 0 remains. Assume that e.g. $x_b \approx 0$, so that the change we want to make to x_a also increases x_b. We could then still increase x_a to x_b, since that does not increase $\max(x_b, y_a)$. At this point, we can either increase or decrease x_b and y_a in conjunction without adding moves, depending on which of x_a and y_b is the larger, until either $x_b, y_a \approx 0$ or $x_a \approx y_b$ (but then we can improve the solution).

Thus there are only a few interesting cases: $x_a, y_b \approx 0$, $x_b, y_a \approx 0$, or $x_a \approx y_b$ and $x_b \approx y_a$. The first two lets us solve for k_x and k_y directly, while the last case gives us systems of two linear equations based on the signs of $s_x + k_x b$ and $t_x - k_x a$. Finally, to make sure get the parities – which only depends on k_x and k_y, and rounding right, we do a small local search around each (k_x, k_y) to find the best solution. □

17.4 Modular Arithmetic

When first taught division, it is often done so using the concept of remainders. For example, when dividing 7 by 3, you would get "2 with a remainder of 1". In general, when dividing a number a with a number n, you would get a ***quotient*** q and a ***remainder*** r. These numbers would satisfy the identity $a = nq + r$, with $0 \leq r < b$.

Example 17.7 — Division with remainders

Consider division (with remainders) by 4 of the numbers $0, \ldots, 6$. We have that

$$\frac{0}{4} = 0, \text{ remainder } 0, \quad \frac{1}{4} = 0, \text{ remainder } 1,$$
$$\frac{2}{4} = 0, \text{ remainder } 2, \quad \frac{3}{4} = 0, \text{ remainder } 3,$$
$$\frac{4}{4} = 1, \text{ remainder } 0, \quad \frac{5}{4} = 1, \text{ remainder } 1,$$
$$\frac{6}{4} = 1, \text{ remainder } 2, \quad \frac{7}{4} = 1, \text{ remainder } 3.$$

Note how the remainder always increase by 1 when the numerator increased. You might remember from Chapter 2 on C++ (or from your favorite programming language) that there is an operator which computes this remainder called the *modulo operator*. Modular arithmetic is computing on integers, where every number is taken modulo some integer n. Under such a scheme, we have that e.g. 3 and 7 are the same if computing modulo 4, since $3 \bmod 4 = 3 = 7 \bmod 4$. This concept, where numbers with the same remainder are treated as if they are equal is called *congruence*.

Definition 17.7 — Congruence

If a and b have the same remainder when divided by n, we say that a and b are ***congruent***

modulo n, written

$$a \equiv b \pmod{n}.$$

An equivalent and in certain applications more useful definition is that $a \equiv b \pmod{n}$ if and only if $n \mid a - b$.

Exercise 17.29. What does it mean for a number a to be congruent to 0 modulo n?

When counting modulo something, the laws of addition and multiplication are somewhat altered:

$+$	0	1	2
0	0	1	2
1	1	2	$3 \equiv 0$
2	2	$3 \equiv 0$	$4 \equiv 1$

$\times$	0	1	2
0	0	0	0
1	0	1	2
2	0	2	$4 \equiv 1$

To perform arithmetic of this form, we use the ***integers modulo*** n rather than the ordinary integers. These has a special set notation as well: $\mathbb{Z}_n$.

While addition and multiplication is quite natural (i.e. performing the operation as usual and then taking the result modulo n), division is a more complicated story. For real numbers, the ***inverse*** x^{-1} of a number x is defined as the number which satisfy the equation $xx^{-1} = 1$. For example, the inverse of 4 is 0.25, since $4 \cdot 0.25 = 1$. The division $\frac{a}{b}$ is then simply a multiplied with the inverse of b. The same definition is applicable to modular arithmetic:

Definition 17.8 — Modular Inverse
The ***modular inverse*** of a modulo n is the integer a^{-1} such that $aa^{-1} \equiv 1 \pmod{n}$, if such an integer exists.

Considering our multiplication table of $\mathbb{Z}_3$, we see that 0 has no inverse and 1 is its own inverse (just as with the real numbers). However, since $2 \cdot 2 = 4 \equiv 1 \pmod{3}$, 2 is actually its own inverse, so all integers are invertible. If we instead consider multiplication in $\mathbb{Z}_4$, the situation is quite different.

$\times$	0	1	2	3
0	0	0	0	0
1	0	1	2	3
2	0	2	0	2
3	0	3	2	1

Now, 2 does not even have an inverse! When does an inverse exist? If a have an inverse, then $aa^{-1} \equiv 1 \pmod{n}$, so that $n \mid aa^{-1} - 1$. By the definition of divisibility, $aa^{-1} - 1 = nk$ for some integer k. A small rearrangement to $aa^{-1} + nk = 1$ brings us to a form known from the extended Euclidean algorithm. If we view a^{-1} and k as unknowns, this is the

same as finding solutions to the Diophantine equation $as + nt = 1$, which is possible if and only if $(a, n) = 1$, i.e. when a is relatively prime to the modulo. This also provides us with a way to find the modular inverse, namely by using the extended Euclidean algorithm. By Bezout's identity, all possible values of a^{-1} differ by a multiple of n, so the inverse is actually unique modulo n.

Just like the reals have a cancellation law for non-zero integers, so does modular arithmetic but for the stronger notation of relatively prime ones.

Theorem 17.7
Assume $(a, n) = 1$. Then $ab \equiv ac \pmod{n}$ implies $b \equiv c \pmod{n}$.

Proof. Since $(a, n) = 1$, there exists an a^{-1} such that $aa^{-1} \equiv 1 \pmod{n}$. Multiplying $ab \equiv ac \pmod{n}$ with a^{-1} results in

$$aa^{-1}b \equiv aa^{-1}c \pmod{n} \Rightarrow b \equiv c \pmod{n}.$$

□

Competitive Tip

In C++, you're sometimes constrained by the range of the built-in integer types. For example, to compute $a \cdot b$ modulo 10^{18} when a, b can also be up to 10^{18}, the intermediate multiplication would overflow a 64-bit `long long`. There are two common ways around this. First, many C++ compilers today support an extension called `__int128_t` that for 128-bit arithmetic, which is often enough. Secondly, multiplication can be reduced to a logarithmic number of additions instead, using a method called *multiplication by doubling*. If you write b in binary as $\sum b_i 2^i$ the product equals $\sum ab_i 2^i$. Each individual term is easy to compute: you start with a and then multiply it by 2 each time. Then you make sure to add together only the sums with $b_i = 1$. No term ever exceeds the modulo plus $2b$ before taking the modulo of the result, so this is fine for moduli up to 10^{18} even when using `long long`.

Exercise 17.30. Let $d(s)$ be the sum of all the digits in s. Prove that $d(s) \equiv s \pmod{9}$.

Problem 17.31.

Modular Arithmetic	modulararithmetic
Candy Distribution	candydistribution
The Magical 3	magical3
Divisibility Shortcut	shortcut

Another common modular operation is exponentiation, i.e. computing $a^m \pmod{n}$. While this can be computed easily in $\Theta(m)$, we can actually do better using a method

called ***exponentiation by squaring***. It is based on the recursion

$$a^m \bmod n = \begin{cases} 1 \bmod n & \text{if } m = 0 \\ a \cdot (a^{m-1} \bmod n) \bmod n & \text{if } m \text{ odd} \\ (a^{\frac{m}{2}} \bmod n)^2 \bmod n & \text{if } m \text{ even} \end{cases}$$

This procedure is clearly $\Theta(\log_2 m)$, since applying the recursive formula for even numbers halve the m to be computed, while applying it an odd number will first make it even and then halve it in the next iteration. It is very important that $a^{\frac{m}{2}} \bmod n$ is computed only once, even though it is squared! Computing it twice causes the complexity to degrade to $\Theta(m \log m)$. This is exactly the same as multiplication by doubling, except for exponentiation.

Problem 17.32.
I Hate the Number Nine nine

Chinese Remainder Theorem

The Chinese Remainder Theorem is an immensely useful theorem in number theoretical problems. It gives us a way of solving systems of modular linear equations.

Theorem 17.8 — Chinese Remainder Theorem
Given an integer system of equations

$$\begin{aligned} x &\equiv a_1 \pmod{m_1} \\ x &\equiv a_2 \pmod{m_2} \\ &\dots \\ x &\equiv a_n \pmod{m_n} \end{aligned}$$

where the numbers $m_1, \dots, m_n$ are pairwise relatively prime (i.e $(m_i, m_j) = 1$.), there is a *unique* integer $x \pmod{\prod_{i=1}^{n} m_i}$ that satisfy the system.

Proof. The theorem is clearly true for $n = 1$, with the unique solution $x = a_1$. Now, assume there are at least 2 equations in the system. Take any ones of these, for example

$$x \equiv a_1 \pmod{m_1} \quad x \equiv a_2 \pmod{m_2}.$$

Let $x = a_1 \cdot m_2 \cdot (m_2^{-1} \bmod m_1) + a_2 \cdot m_1 \cdot (m_1^{-1} \bmod m_2)$, where $m_1^{-1} \bmod m_2$ is taken to be a modular inverse of m_1 modulo m_2. These inverses exist, since $(m_1, m_2) = 1$ by

assumption. We then have that

$$x \equiv a_1 \cdot m_2 \cdot (m_2^{-1} \bmod m_1) \equiv a_1 \pmod{m_1}$$

$$x \equiv a_2 \cdot m_1 \cdot (m_1^{-1} \bmod m_2) \equiv a_2 \pmod{m_2}.$$

Since a solution exist for every a_1, a_2, this solution must be unique – there are $m_1 \cdot m_2$ possible values for a_1, a_2, and $m_1 \cdot m_2$ possible values for x. Thus, the theorem is also true for $n = 2$.

Assume that the theorem is true for $k - 1$ equations. Then, we can replace the equations

$$x \equiv a_1 \pmod{m_1} \quad x \equiv a_2 \pmod{m_2}$$

with another equation

$$x \equiv x^* \pmod{m_1 m_2}$$

where x^* is the solution to the first two equations since they hold for the exact same values of x. This reduces the number of equations to $k - 1$. Repeat this procedure until these is only a single equation left. □

Note that the theorem gave a constructive proof, allowing us to find what the unique solution to such a system is.

Problem 17.33.
Chinese Remainder chineseremainder

Radar – radar

By Erik Aas. KTH Challenge 2014. CC BY-SA 3.0. Shortened.

We say that an integer z is within distance y of an integer x modulo an integer m if

$$z \equiv x + t \pmod{m}$$

where $|t| \leq y$. Find the smallest non-negative integer z such that it is:

- within distance y_1 of x_1 modulo m_1,
- within distance y_2 of x_2 modulo m_2, and
- within distance y_3 of x_3 modulo m_3.

All the y_i are between 0 and 300. All the x_i and m_i are between 0 and 10^6.

Solution. The problem gives rise to three linear equations of the form

$$z \equiv x_i + t_i \pmod{m_i}$$

where $-y_i \leq t_i \leq y_i$. If we fix all the variables t_i, the problem reduces to solving the system of equations using CRT. We could then find all possible values of z, and choose the

minimum one. This requires applying the CRT construction about $2 \cdot 600^3 = 432\,000\,000$ times. Since the modulo operation involved is quite expensive, this approach would use too much time. Instead, let us exploit a useful greedy principle in finding minimal solutions.

Assume that z is the minimal answer to an instance. There are only two situations where $z - 1$ cannot be a solution as well:

- $z = 0$ – since z must be non-negative, this is the smallest possible answer
- $z \equiv x_i - y_i$ – then, decreasing z would violate one of the constraints

In the first case, we only need to verify whether $z = 0$ is a solution to the three inequalities. In the second case, we managed to change an inequality to a linear equation. By testing which of the i this equation holds for, we only need to test the values of t_i for the two other equations. This reduces the number of times we need to use the CRT to $600^2 = 360\,000$ times, a modest amount well within the time limit. □

Competitive Tip

While the choice of

$$x = a_1 \cdot m_2 \cdot (m_2^{-1} \bmod m_1) + a_2 \cdot m_1 \cdot (m_1^{-1} \bmod m_2)$$

as solution the Chinese Remainder Theorem is normal to use in the proof because it explains intuitively how the theorem works, it is problematic in implementations since it computes the product of three large integers. When implementing it on code, you should instead use the computation

```
a1 + ((a2 - a1) % m2) * m1inv) % m2) * m1
```

where `m1inv` is the inverse of m_1 modulo m_2. If you guarantee that $m_2 < m_1$, $0 \le a_1 \le m_1$ and $0 \le a_2 \le m_2$ this handles all cases where $m_1 m_2$ fit in a `long long`.

In particular, the original choice of x fails for the *Radar* problem due to this overflow issue.

The classical CRT problem requires the moduli to be pairwise relatively prime. Clearly this is necessary for the system of equations to always be solvable (with the trivial system $x \equiv 0 \pmod 2$ and $x \equiv 1 \pmod 2$ as counterexample). It's fortunately possible to say something about *when* such a system is solvable, and finding a solution in the general case.

Theorem 17.9 — Chinese Remainder Theorem, general moduli
Given an integer system of equations

$$\begin{aligned} x &\equiv a_1 \pmod{m_1} \\ x &\equiv a_2 \pmod{m_2} \end{aligned}$$

there is a *unique* integer $x \pmod{\operatorname{lcm}(m_1, m_2)}$ that satisfy the system **if and only if**

$a_1 \equiv a_2 \pmod{(\gcd(m_1, m_2))}$.

Proof. We have $x - a_1 = m_1 k_1$ and $x - a_2 = m_2 k_2$. This means that

$$a_1 + m_1 k_1 = a_2 + m_2 k_2$$

$$a_1 - a_2 = m_2 k_2 - m_1 k_2$$

so

$$(m_2 k_2, m_1 k_2) \mid a_1 - a_2.$$

Then $(m_1, m_2) \mid a_1 - a_2$ so the *only if* part holds.

Now, let $y m_1 \equiv (m_1, m_2) \pmod{m_2}$ (this exists by the extended Euclidean algorithm). Pick

$$x = a_1 + \frac{a_2 - a_1}{(m_1, m_2)} \cdot m_1 \cdot y.$$

Clearly $x \equiv a_1 \pmod{m_1}$. Furthermore,

$$x \equiv a_1 + \frac{a_2 - a_1}{(m_1, m_2)} \cdot (m_1, m_2) \equiv a_2 \pmod{m_2}$$

so this is a solution.

Uniqueness is proven using the same bijectivity argument as before. For each of the m_1 possible choices of a_1, there are $\frac{m_2}{(m_1, m_2)}$ choices of a_2 such that $a_1 \equiv a_2 \pmod{\gcd(m_1, m_2)}$. Thus there are $\frac{m_1 m_2}{(m_1, m_2)} = \operatorname{lcm}(m_1, m_2)$ such equation systems for a given m_1, m_2. Since each of them have a solution modulo $\operatorname{lcm}(m_1, m_2)$, there must be exactly one solution per system. □

Problem 17.34.

CRT (non-relatively prime moduli)	generalchineseremainder
Gears in Action	gears
Remainder Reminder	remainderreminder

Miller-Rabin Primality Testing

With the tools from modular arithmetic, we have gained the tools to check primality *in logarithmic time* using the ***Miller-Rabin test***. The mathematical details are not very useful for us. It is motivated by properties of modular exponentiation shown in the next section. The classical Miller-Rabin test is probabilistic, but for integers up to 64 bits a very fast deterministic variant has been found.

The algorithm, which we won't prove the correctness of or motivate in any way, is the following:

```
1: procedure IsPrime(N)
2:     if N = 2 then
```

```
3:          return true
4:      if N = 1 or N is even then
5:          return false
6:      S ← the largest d such that 2^d | (N − 1)
7:      D ← (N−1)/2^d
8:      for each a in {2, 325, 9375, 28178, 450775, 9780504, 1795265022} do
9:          P ← a^D mod N
10:         if P ≠ 0 and P ≠ 1 and P ≠ N − 1 then
11:             for S − 1 times do
12:                 P ← (P · P) mod N
13:                 if P = N − 1 then
14:                     restart the loop on line 8 with the next a
15:             return false
16:     return true
```

Note that the above code requires multiplication of two integers up to N, so be careful about overflows and use 128-bit integers (or multiplication-by-doubling) when performing the multiplications!

Problem 17.35.

Primality	primality	*(all subtasks)*
Divisions	divisions	

17.5 Euler's Totient Function

Now that we have talked about modular arithmetic, we finally have the tools to give numbers which are *not* divisors of some integer n their well-deserved attention. This discussion will start with the ϕ-function.

Definition 17.9 — Euler's totient function
Euler's totient function $\phi(n)$ is defined as the number of integers in $[1, n]$ that are relatively prime (i.e. only shares the trivial divisors with) to n.

Example 17.8 What is $\phi(12)$? The numbers 2, 4, 6, 8, 10 all have the factor 2 in common with 12 and the numbers 3, 6, 9 all have the factor 3 in common with 12. This leaves us with the integers 1, 5, 7, 11 which are relatively prime to 12. Thus, $\phi(12) = 4$.

For prime powers, $\phi(p^k)$ is easy to compute. The only integers which are not relatively prime to $\phi(p^k)$ are the multiples of p, which there are $\frac{p^k}{p} = p^{k-1}$ of, meaning

$$\phi(p^k) = p^k - p^{k-1} = p^{k-1}(p-1).$$

It turns out that $\phi(n)$ has the same property that the *GCD Sum* problem has.

Theorem 17.10
$\phi(n)$ is a ***multiplicative*** function, which means that if $a \perp b$, then

$$\phi(ab) = \phi(a)\phi(b).$$

Proof. We show that there is an invertible mapping between integers $1 \le x \le ab$ coprime to ab, and pairs $1 \le i \le a$ and $1 \le j \le b$ coprime to a and b, respectively. The mapping we use is $x \to (x \bmod a, y \bmod b)$.

First, note that x is coprime to ab if and only if it is coprime to both a and b. Thus, we must have that $x \bmod a$ is also coprime to a, and $y \bmod b$ is also coprime to b, so the mapping does indeed map each x to such a pair (i, j).

Furthermore, this mapping is invertible: for each (i, j) we have that

$$\begin{aligned} x &\equiv i \pmod{a} \\ x &\equiv j \pmod{b} \end{aligned}$$

which has exactly one solution in $1 \le x \le ab$ by the Chinese remainder theorem.

The number of (i, j), of which there are $\phi(a)\phi(b)$, is thus equal to the number of x, of which there are $\phi(ab)$), which proves the theorem. □

For multiplicative functions, we can reduce the problem of computing arbitrary values of the function to finding a formula only for prime powers. Using the multiplicativity of ϕ we get the simple formula

$$\phi(p_1^{e_1} \dots p_k^{e_k}) = \phi(p_1^{e_1}) \cdots \phi(p_k^{e_k}) = p_1^{e_1-1}(p_1 - 1) \cdots p_k^{e_k-1}(p_k - 1).$$

Problem 17.36.
Relatives relatives

Exercise 17.37. Prove that

$$n = \sum_{d|n} \phi(d).$$

Computing ϕ for a single value reduces to factoring the number. If we wish to compute ϕ for an interval $[1, n]$ we can thus use the Sieve of Eratosthenes.

Problem 17.38.
Farey Sums fareysums

This seemingly convoluted function might seem useless, but is of great importance via the following theorem:

Theorem 17.11 — Euler's theorem
If a and n are relatively prime and $n \geq 1$,

$$a^{\phi(n)} \equiv 1 \pmod{n}$$

Proof. The proof of this theorem isn't trivial, but it is number theoretically interesting and helps to build some intuition for modular arithmetic. The idea behind the proof is to consider the product of the $\phi(n)$ positive integers less than n which are relatively prime to n. We will call these $x_1, x_2, \ldots, x_{\phi(n)}$. Since these are all distinct integers between 1 and n, they are incongruent modulo n. We call such a set of $\phi(n)$ numbers relatively prime to n and all incongruent modulo n a ***complete residue system*** (CRS) modulo n. Next, we will prove that $ax_1, ax_2, \ldots, ax_\phi$ also form a CRS modulo n.

We will start with the first property, that they are all relatively prime to n. Since both a and x_i are relatively prime to n, neither number have a prime factor in common with n. This means ax_i have no prime factor in common with n either, meaning the two numbers are relatively prime.

To prove that they are incongruent modulo n we use the cancellation property of modular arithmetic (Theorem 17.7). If $ax_i \equiv ax_j \pmod{n}$, the cancellation law gives us $x_i \equiv x_j \pmod{n}$. Since all x_i are incongruent modulo n, we must have $i = j$, meaning all the numbers ax_i are incongruent as well. Thus, these numbers did indeed form a complete residue system modulo n.

If $ax_1, \ldots, ax_{\phi(n)}$ form a CRS, we know every ax_i must be congruent to some x_j, meaning

$$ax_1 \cdots ax_{\phi(n)} \equiv x_1 \cdots x_{\phi(n)} \pmod{n}$$

Factoring the left hand size turns this into

$$a^{\phi(n)} x_1 \cdots x_{\phi(n)} \equiv x_1 \cdots x_{\phi(n)} \pmod{n}$$

Since all the x_i are relatively prime to n, we can again use the cancellation law, leaving

$$a^{\phi(n)} \equiv 1 \pmod{n}$$

completing our proof of Euler's theorem. □

Competitive Tip

For primes p we get a special case of Euler's theorem since $\phi(p) = p - 1$:

$$a^{p-1} \equiv 1 \pmod{p}$$

when $p \nmid a$, called ***Fermat's Theorem***. Multiplying both sides by a^{-1} gives that $a^{p-2} \equiv a^{-1}$

$(\mathrm{mod}\ p)$, which is an easy way of computing modular inverses modulo primes using exponentiation-by-squaring.

Exponial – exponial

By Per Austrin. Nordic Collegiate Programming Contest 2016. CC BY-SA 3.0. Shortened

Define the *exponial* of n as the function

$$\mathrm{exponial}(n) = n^{(n-1)^{(n-2)^{\cdots^{2^1}}}}$$

Compute $\mathrm{exponial}(n) \pmod m$ where $1 \le n, m \le 10^9$.

Proof. Euler's theorem suggests a recursive approach. Since $n^e \pmod m$ is periodic (in e), with a period of $\phi(m)$, maybe when computing $n^{(n-1)^{\cdots}}$ we could first compute $e = (n-1)^{\cdots}$ modulo $\phi(m)$ and only then compute $n^e \pmod m$? This is of course just the same problem but with $n := n-1$ and $m := \phi(m)$. Alas, this only helps us when $n \perp m$, since that is a necessary precondition for Euler's theorem.

A standard number theoretical transformation comes to the rescue: instead of working modulo some integer m, we can take the prime factorization of $p_1^{e_1} \cdots p_k^{e_k}$ and compute modulo each of its its prime powers $p_i^{e_i}$. The result modulo m is then computed by the Chinese remainder theorem. While some prime powers will share factors with n, they are much easier to handle. When computing $n^e \bmod p_i^{e_i}$ we have two cases. Either $p_i \mid n$, in which case $n^e \equiv 0 \pmod{p_i^{e_i}}$ whenever $e \ge e_i$. Otherwise, $p_i \perp n$, and $n^e \equiv n^{e \bmod \phi(p_i^{e_i})} \pmod{p_i^{e_i}}$ by Euler's theorem.

This suggests that as long as $e \ge \max(e_1, \ldots, e_k)$, n^e is still periodic. Furthermore, e_i is bounded by $\log_2 n$. Otherwise, $p_i^{e_i} > 2^{\log_2 n} = n$, a contradiction.

Since $\log_2(10^9) \approx 30$ and $4^{3^{2^1}} \ge 30$, we know that n^e is periodic whenever $n \ge 5$. For $n = 4$, the exponial equals only 262144, meaning we can compute it naively.

One final insight remains. If we use the recursive formula, i.e. first computing $e = (n-1)^{(n-2)^{\cdots}} \bmod \phi(m)$ and then $n^{\phi(m)+e \bmod \phi(m)} \bmod m$, we still have the problem that n can be up to 10^9. We would need to perform a number of exponentiations that is linear in n, which is slow for such large n. However, our modulus will actually very quickly converge to 1. While the final result is taken modulo m, the first recursive call is taken modulo $\phi(m)$. The recursive call performed at the next level will thus be modulo $\phi(\phi(m))$, and so on. This sequence decreases very quickly based on two facts. For even m, $\phi(m) = \phi(2)\phi(\frac{m}{2}) = \phi(\frac{m}{2}) \le \frac{m}{2}$. For odd m, $\phi(m)$ is even. Any odd m consists only of odd prime factors, but since $\phi(p) = p-1$ (i.e. an even number for odd primes p) and ϕ is multiplicative, $\phi(m)$ must be even. Thus $\phi(\phi(m)) \le \frac{m}{2}$ for $m > 1$ (1 is neither even nor contains an odd prime factor). This means the modulus will become 1 in a logarithmic number of iterations, so the recursive step reaches a simple base case quickly. With this,

we are done. □

Additional Exercises

Problem 17.39.

Joint Attack	jointattack
Inheritance	inheritance
I'm Thinking of a Number	thinkingofanumber
Primal Representation	primalrepresentation
Happy Happy Prime Prime	happyprime
Prime Path	primepath
Three Digits	threedigits
GCD Sum 2	gcdsum2
Repeating Decimal	repeatingdecimal
Cocoa Coalition	cocoacoalition
Inverse Totient	inversetotient
LCM Pair Sum	lcmpairsum
Ternarian Weights	ternarianweights
Rational Sequence	rationalsequence
Factor-Free Tree	factorfree
Guma	guma

Notes

A highly theoretical introduction to classical number theory can be found in Hardy and Littlewood's *An Introduction to the Theory of Numbers* [22]. While devoid of exercises and examples, it is very comprehensive.

Victor Shoup's *A Computational Introduction to Number Theory and Algebra* [45] instead takes a more applied approach, and is freely available under a Creative Commons license at the author's home page.[4].

If you're interested in understanding more about the Miller-Rabin test, see e.g. Miller [37] and Rabin [41].

[4] http://www.shoup.net/ntb/

Chapter 18

Combinatorics

Combinatorics is a mathematical area with as many definitions as it has sub-fields. A very handwavy description is that it, in broad strokes, deals with counting objects, proving that objects exist, constructing objects and sometimes finding the best object. The various topics in combinatorics are unified in that they tend to deal with discrete structures.

A lot (if not most!) of algorithmic problem solving falls under the wide umbrella of combinatorics – for example, graph theory is considered to be a sub-field of combinatorics. In this chapter we mainly study the branch of combinatorics known as *enumerative combinatorics* – the art of counting. For example, we count the number of ways to shuffle a list of N items such that no item is in its original place, the number of shortest axis-aligned paths through a grid and many other things. A large number of combinatorial counting problems are based on a few standard techniques which we learn in this chapter.

We also include some non-counting combinatorics, like a detailed analysis on permutations and further study on invariants, a concept you have come across in a few other chapters.

18.1 The Addition and Multiplication Principles

The ***addition principle*** states that, given a finite collection of **disjoint** sets we can compute the size of the union of all sets by adding up the sizes of our sets, i.e.

$$|S_1 \cup S_2 \cup \cdots \cup S_n| = |S_1| + |S_2| + \cdots + |S_n|.$$

Example 18.1 Assume that we have 5 different types of chocolate bars (the set C), 3 different types of bubble gum (the set G), and 4 different types of lollipops (the set L). These form three disjoint sets, meaning that we can compute the total number of snacks by summing up the number of snacks of the different types. Thus, we have $|C| + |G| + |L| = 5 + 3 + 4 = 12$ different snacks.

Later on, we will see a generalization of the addition principle that handles cases where our sets are not disjoint.

The ***multiplication principle***, on the other hand, states that the size of the Cartesian product $S_1 \times S_2 \times \cdots \times S_n$ equals the product of the individual sizes of these sets, i.e.

$$|S_1 \times S_2 \times \cdots \times S_n| = |S_1| \cdot |S_2| \cdots |S_n|.$$

Example 18.2 How many subsets are there of an N-element set? For each element i there are two choices: either it's included, or it's not. By the multiplication principle, there are 2^N such choices (and thus subsets).

Exercise 18.1. How many pairs of disjoint subsets A, B are there of an N-element set?

Problem 18.2.
Best Compression Ever bestcompression
Character Development character

The addition principle is often useful when we solve counting problems by case analysis.

Example 18.3 How many four letter words consisting of the letters a, b, c and d contain exactly two letters a? There are six possible ways to place the two letters a:

$$aa\_\_ \qquad a\_a\_$$
$$a\_\_a \qquad \_aa\_$$
$$\_a\_a \qquad \_\_aa$$

For each of these ways, there are four ways of choosing the other two letters (bb, bc, cb, cc). Thus, there are $4 + 4 + 4 + 4 + 4 + 4 = 6 \cdot 4 = 24$ such words.

Problem 18.3. *Incognito* incognito

Later counting principles are mostly built from these basic building blocks. However, in many combinatorial counting problems, it's sufficient only to apply these two principles a number of times.

Kitchen Combinatorics – kitchencombinatorics

By Per Austrin. Northwestern Europe Regional Contest 2015. CC BY-SA 3.0. Shortened.

The Swedish Chef is planning a three-course dinner: a starter course, a main course, and a dessert. His cook-book offers a wide variety of choices for each course, though some of them do not go well together (for instance, you cannot serve chocolate moose and sooted shreemp at the same dinner).

Each potential dish has a list of ingredients. Ingredients are in turn available from a few different brands. Each brand is unique, so using a particular brand of an ingredient results in a different dinner experience than using another brand of the same ingredient. Some ingredients may appear in several of the chosen dishes. When an ingredient is used in multiple dishes, Swedish Chef uses

the same brand of the ingredient in all of them.

There are $r \leq 1\,000$ different ingredients, $s, m, d \leq 25$ available starter dishes, main dishes and desserts, respectively, and $n \leq 2\,000$ pairs of dishes that do not go well together. For each ingredient i, there are $b_i \leq 100$ different brands. Each dish may contain up to 20 different ingredients.

How many different dinner experiences are there that he could make, by different choices of dishes and brands for the ingredients? If the number of different dinner experiences Swedish Chef can make is more than 10^{18}, output "`too many`".

Solution. The solution is a addition-multiplication principle combo like the one in Example 18.3. We can simplify the problem considerably by brute forcing over the coarsest component of a dinner experience, namely the courses included. Since there are at most 25 dishes of every type, we need to check up to $25^3 = 15\,625$ choices of dishes. By the addition principle, we can compute the number of dinner experiences for each such three-course dinner, and then sum them up to get the answer.

Some pairs of dishes do not go well together. At this stage in the process we exclude any triple of dishes that include such a pair. We can perform this check in constant time if we save the incompatible dishes in 2D boolean vectors, so that e.g. badStarterMain$[i][j]$ determines if starter i is incompatible with main dish j.

For a given dinner course consisting of starter a, main dish b and dessert c, only the **set** of ingredients of three dishes matters since the chef will use the same brand for an ingredient even if it is part of two dishes. The next step is to compute this set by taking the union of ingredients for the three included dishes. This step takes $\Theta(k_a + k_b + k_c)$ time. After that, the only remaining task is to choose a brand for each ingredient. Assigning brands is an ordinary application of the multiplication principle, where we multiply the number of brands available for each ingredient together. □

Problem 18.4. *Dunglish* dunglish

The objects we are meant to count often have some kind of recursive structure. We already solved several combinatorial counting problems in Chapter 7 on recursion. At first we counted objects by constructing them recursively one at a time (e.g. by backtracking), but not much later we gained the tool of dynamic programming to speed up counting solutions. Next, we look at a typical recursive counting problem.

Booming Business – boomingbusiness

By Mees de Vries. BAPC preliminaries 2017. CC BY-SA. Shortened.

You are an expert in bonsai, the Japanese art of cultivating small trees in small containers. You recently rented a store to sell your creations. In the store's window display, you want to show the most impressive tree possible that is exactly as tall as the window, and exactly as heavy as the display floor allows.

By definition a bonsai tree consists of a single branch, with zero or more smaller bonsai trees branching off from that branch.

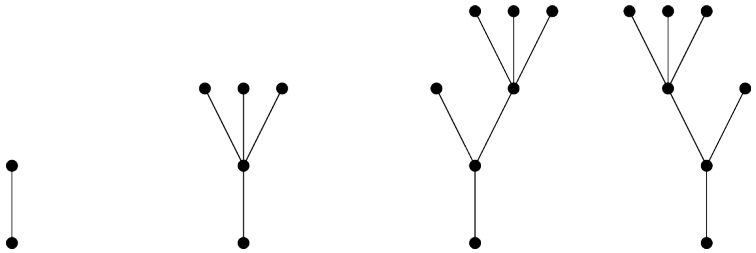


Figure 18.1: Four distinct examples of bonsai trees. Their weights are $1, 4, 6, 6$ and their heights are $1, 2, 3, 3$.

A tree's weight is equal to the number of branches that appear in it, while its height equals the length of the longest chain of branches from the root to the top of the tree.

Compute the number of different trees you could grow that has exactly height $1 \le h \le 300$ and weight $1 \le w \le 300$.

Solution. In combinatorics, it's sometimes easier to count the number of objects where an integer property, such as the height of a tree, is equal to *at most* a given value, rather than *exactly* that value (and sometimes, it's the other way around). Luckily, these two problems can easily be transformed to each other. To find the number of objects where the property is exactly h, you count the number where it's at most h and subtract those where it's at most $h-1$. For the reverse situation, you just add up all those objects with h equal to all the values you're interested in. In this problem, it's easier to count the number of trees of height up to h, and subtract away those of height up to $h-1$.

Let $T(h, w)$ be the number of trees of height $\le h$ and weight equal to w. To progress, we should try to break the tree down recursively. The problem already describes the recursive structure of the tree: start with a branch, and then build some new trees. It is then natural to ask what the properties are of the leftmost smaller bonsai tree of the larger tree. If we know its height (at most $h-1$) and weight w' (arbitrary between 1 and $w-1$), we can simply cut it off from the main tree and count the number of ways that the remaining tree, with height at most h and weight $w-w'$, looks. By trying all choices of w', we find the recursion

$$T(h, w) = \sum_{w'=1}^{w-1} T(h-1, w') \cdot T(h, w-w').$$

This recursion takes $\Theta(hw^2)$ time to evaluate with dynamic programming. The base cases remain, but they are straightforward. □

Problem 18.5.

Card Magic	cardmagic
Bobby's Bet	bobby

18.2 Permutations

We are now going to build upon the simple combinatorial principles of addition and multiplication to study more complex combinatorial objects. One of the most fundamental one is the permutation.

Definition 18.1 — Permutation
A ***permutation*** of a set $\{a_1, \ldots, a_n\}$ is an ordering $\langle a'_1, \ldots, a'_n \rangle$ of all the elements in the set.

Example 18.4 The set $\{1, 2, 3\}$ has 6 permutations:

$$\langle 1,2,3\rangle, \quad \langle 1,3,2\rangle, \quad \langle 2,1,3\rangle,$$
$$\langle 2,3,1\rangle, \quad \langle 3,1,2\rangle, \quad \langle 3,2,1\rangle.$$

Our first task is to count the number of permutations of an n-element set S. We use an iterative procedure that constructs a permutation one element at a time. Assume that the permutation is the sequence $\langle a_1, a_2, \ldots, a_n \rangle$. As the first element a_1, any of the the n elements of S can be chosen. Once this assignment has been made, there are $n-1$ possible choices for a_2 (any element of S except a_1). In general, when we are to select the $(i+1)$'th value a_{i+1} of the permutation, i elements have already been included in the permutation, leaving $n - i$ options for a_{i+1}. By the multiplication principle, we have $n \cdot (n-1) \cdots 2 \cdot 1$ sequences of choices in total. No permutation is constructed in two different way and every permutation can be constructed in this way, so this also counts exactly the number of permutations.

Problem 18.6. *Counting Greedily Increasing Supersequences* countinggis

This number is so useful that it has its own name and notation.

Definition 18.2 — Factorial
The ***factorial*** of a non-negative integer n, denoted $n!$, is defined as the product of the first n positive integers, i.e.

$$n! = 1 \cdot 2 \cdots n = \prod_{i=1}^{n} i.$$

For $n = 0$, we use the convention that the empty product is 1.

This sequence of numbers begins 1, 1, 2, 6, 24, 120, 720, 40 320, 362 880, 3 628 800, 39 916 800 for $n = 0, 1, 2, \ldots, 11$. It is good to know the magnitudes of these numbers, since they are frequent in time complexities when doing brute force over permutations. Asymp-

totically, they grow as $n^{\Theta(n)}$. More precisely, the well-used *Stirling's formula*[1] gives the approximation

$$n! = \sqrt{2\pi n}\left(\frac{n}{e}\right)^n \left(1 + O\left(\frac{1}{n}\right)\right).$$

Exercise 18.7. In how many ways can 8 persons be seated around a round table, if we consider cyclic rotations of a seating to be different? What if we consider cyclic rotations to be equivalent?

Problem 18.8.

Euler's Number	eulersnumber
Name That Permutation	namethatpermutation

Permutations as Bijections

The word *permutation* has roots in Latin, meaning "to change completely". We are now going to look at permutations in a very different light, justifying the etymology of the word.

Given a set such as $\{1, \ldots, 5\}$, we can fix some "base" ordering of its elements such as $\langle 1, 2, 3, 4, 5\rangle$. A permutation $\pi = \langle 1, 3, 4, 5, 2\rangle$ of this set can then be seen as a movement of these elements. Of course, this same movement can be applied to any other 5-element set with a fixed permutation, such as $\langle a, b, c, d, e\rangle$ being transformed to $\langle a, c, d, e, b\rangle$. This suggests that we can consider permutations as rules which describe how to move – *permute* – the elements of a list, rather than as an ordering of any set in particular.

Such a rule can also be described as a function $\pi : [n] \to [n]$, where $\pi(i)$ describes **what element should be placed at position** i. Thus, the permutation $\langle 1, 3, 4, 5, 2\rangle$ would have $\pi(1) = 1$, $\pi(2) = 3$, $\pi(3) = 4$, $\pi(4) = 5$, $\pi(5) = 2$.

i	1	2	3	4	5
	↓	↓	↓	↓	↓
$\pi(i)$	1	3	4	5	2

Since each element is mapped to a different element, the function induced by a permutation is actually a bijection. By interpreting permutations as function, all the theory from functions apply to permutations too.

We call $\langle 1, 2, 3, \ldots, n-1, n\rangle$ the *identity permutation*, since the function given by the identity permutation is actually the identity function. Permutations can also be composed in the same way as functions. Given two permutations α and β, their composition $\alpha\beta$ is also a permutation given by $\alpha\beta(k) = \alpha(\beta(k))$. Composing $\alpha = \langle 5, 4, 3, 2, 1\rangle$ and $\beta = \langle 1, 3, 4, 5, 2\rangle$ would result in the permutation

[1] Named after James Stirling (who have other important combinatorial objects named after him too), but stated already by his contemporary Abraham de Moivre.

i	1	2	3	4	5
	↓	↓	↓	↓	↓
$\beta(i)$	1	3	4	5	2
	↓	↓	↓	↓	↓
$\alpha\beta(i)$	5	3	2	1	4

This is called *multiplying* permutations, i.e. $\sigma\pi$ is the product of σ and π. If we multiply a permutation π by itself n times, we call the resulting product π^n.

An important property of permutation composition follows from its functional properties, namely associativity, i.e. $(\alpha\beta)\gamma = \alpha(\beta\gamma)$. This property is shared with ordinary multiplication of real numbers. Similarly, we take the liberty of dropping the parentheses and write $\alpha\beta\gamma$ instead. Thanks to the associativity, tricks from previous chapters such as exponentiation by squaring and segment trees for computing the product of segments of a list of permutations can be used.

Problem 18.9.

Slom	slom
Parade	parade2

A related concept is that of the ***cycle decomposition*** of a permutation. If we start with an element i and repeatedly apply a permutation on this element (i.e. take $i, \pi(i), \pi(\pi(i)), \ldots$) we will at some point find that $\pi^k(i) = i$, at which point we will start repeating ourselves.

i	1	2	3	4	5
	↓	↓	↓	↓	↓
$\pi(i)$	2	1	4	5	3
	↓	↓	↓	↓	↓
$\pi^2(i)$	1	2	5	3	4
	↓	↓	↓	↓	↓
$\pi^3(i)$	2	1	3	4	5
	↓	↓	↓	↓	↓
$\pi^4(i)$	1	2	4	5	3
	↓	↓	↓	↓	↓
$\pi^5(i)$	2	1	5	3	4
	↓	↓	↓	↓	↓
$\pi^6(i)$	1	2	3	4	5

We call the k distinct numbers of this sequence the *cycle* of i. For π, we have two cycles: $(1, 2)$ and $(3, 4, 5)$. Note how $\pi(1) = 2$ and $\pi(2) = 1$ for the first cycle, and $\pi(3) = 4, \pi(4) = 5, \pi(5) = 3$. It gives us an alternative way of writing it, namely as the concatenation of its cycles: $(1, 2)(3, 4, 5)$.

To compute the cycle decomposition of a permutation π, we repeatedly pick any element of the permutation which is currently not a part of a cycle, and compute the cycle it is in using the method described above. Since every element is processed exactly once, this procedure is $\Theta(n)$ for n-element permutations.

Problem 18.10. *Job Switching* jobbyte

Given a permutation π, we define its ***order***, denoted $\operatorname{ord}\pi$, as the size of the set $\{\pi, \pi^2, \pi^3, \dots\}$. This is the smallest positive integer k such that π^k is the identity permutation. In our example, we have that $\operatorname{ord}\pi = 6$, since π^6 was the first power of π that was equal to the identity permutation.

How can we quickly compute the order of π? The maximum possible order of a permutation happens to grow rather quickly (it is $e^{(1+o(1))\sqrt{n \log n}}$ in the number of elements n). Thus, trying to compute the order by computing π^k for every k until π^k is the identity permutation is too slow. Instead, we can use the cycle decomposition. If a permutation of π has a cycle $(c_1, c_2, \dots c_l)$, we know that

$$\pi^l(c_1) = c_1, \pi^l(c_2) = c_2, \dots, \pi^l(c_l) = c_l$$

by the definition of the cycle composition. Additionally, this means that $(\pi^l)^k(c_1) = (\pi^{lk})(c_1) = c_1$. Any power of π that is a multiple of l acts as the identity permutation *on this particular cycle*.

This fact gives us an upper bound on the order of π. If its cycle decomposition has cycles of length $l_1, l_2, \dots, l_m$, then k can be chosen to be the lowest common multiple[2] of all those cycle lengths. The permutation $\pi = \langle 2, 1, 4, 5, 3\rangle$ had two cycles, one of length 2 and 3. Its order was $\operatorname{lcm}(2, 3) = 2 \cdot 3 = 6$. This is also a lower bound on the order, which follows from a fact left as an exercise:

Exercise 18.11. Prove that if π has a cycle of length l, we must have $l \mid \operatorname{ord}\pi$.

Dance Reconstruction – dance

By Lukáš Poláček. Nordic Collegiate Programming Contest 2013. CC BY-SA 3.0. Shortened.

Marek loves dancing and is due to attend the coming wedding of his best friend Miroslav. For a whole month he worked on a special dance for the wedding. The dance was performed by $N \le 10\,000$ people and there were N marks on the floor. Each mark had an arrow to another mark, and every mark had exactly one incoming arrow. The arrow could be also pointing back to the same mark.

At the wedding, every person first picked a mark on the floor and no two persons picked the same one. Every 10 seconds, there was a loud signal when all dancers had to move along the arrow on the floor to another mark. If an arrow was pointing back to the same mark, the person at the mark just stayed there and maybe did some improvised dance moves on the spot.

A year later, another wedding is coming up, and Marek would like to do a similar dance. He found two photos from exactly when the dance started and when it ended. Marek also remembers that the signal was triggered $1 \le K \le 10^9$ times during the time the song was played, so people moved K times along the arrows. The photos shows the initial and final positions for every dancer.

Given the two photos, can you help Marek reconstruct any possible placement of the arrows on the floor, or determine that no such placement exists?

---

[2] Return to the chapter on number theory, page 17.6 if you do not remember how this is computed.

Solution. The problem can be rephrased in terms of permutations. First of all, the dance corresponds so some permutation π of the dancers, given by where the arrows pointed. This is the permutation we seek in the problem. We are given the permutation a, so we seek a permutation π such that $\pi^K = a$, i.e. the *K'th root* of π.

When given permutation problems of this kind, the cycle decomposition is a great avenue of attack. Since the cycles of π are all independent of each other under exponentiation, it is a good guess that the decomposition can simplify the problem. Assume that π^K has some cycle $(c_1, \ldots, c_l)$. Can we not just find the K'th root of it and assume that is how the cycle looked in π? Sadly not – the cycle may lack a fifth root. This has to do with how cycles are affected by taking powers. For example, a cycle of 10 elements in π would actually decompose into five cycles of length two in π^5, or two cycles of length 5 in π^2. This means that cycles of some length in π^K can not always have had that same length in π. In general, how cycles are decomposed under taking powers involves the divisors of l and K:

Exercise 18.12. Prove that a cycle of length l in a permutation π decomposes into $\gcd(l, K)$ cycles of length $\frac{l}{\gcd(l,K)}$ in π^K.

While the cycles of π^K were not completely independent, at least cycles of different lengths could not have been part of a single cycle in π. This gives us a small step forward: partition all cycles of π^K by their lengths.

Furthermore, we can generate all the possible ways that this decomposition may be reversed. For each possible cycle length l we can compute the number of cycles it decomposed to, and their length. There are N possible lengths, so it takes $\Theta(N \log(N+K))$ time to find them due to the GCD computation.

Given all the ways to combine cycles, a knapsack problem remains for each cycle length of π^K. If we have a cycles of length l in π^K, they must be partitioned into sets of those sizes that allows recombination (given by the computation in the previous paragraph). This step takes $\Theta(ac)$ ways, if there are c ways to combine a-length cycles. Since $\sum a \leq N$ and $c \leq N$, over all a this is bounded by N^2 which is fast enough.

Once it has been decided what cycles are to be combined, only the act of computing a combination of them remains. This is not difficult on a conceptual level, but is a good practice to do on your own (the solution to Exercise 18.12 basically outlines the reverse procedure). □

Problem 18.13. *The Power of Substitution* substitution

18.3 Ordered and Unordered Subsets

A variation of the permutation counting problem is to count the number of ordered sequences containing *exactly k* distinct elements from a set.

Definition 18.3 An ***ordered k-subsets*** of a set $S = \{a_1, \ldots, a_n\}$ is an ordered list of k distinct elements $\langle a'_1, \ldots, a'_k \rangle$ chosen from S.

We can compute the number of ordered k-subsets by taking all the permutations of the entire set of n elements and grouping together those whose k first elements are the same. Taking the set $\{a, b, c, d\}$ as an example, it has the permutations:

abcd	**ba**cd	**ca**bd	**da**bc
abdc	**ba**dc	**ca**db	**da**cb
acbd	**bc**ad	**cb**ad	**db**ac
acdb	**bc**da	**cb**da	**db**ca
adbc	**bd**ac	**cd**ab	**dc**ab
adcb	**bd**ca	**cd**ba	**dc**ba

Once the first k elements in the permutation are fixed, there are $(n-k)!$ ways to order the remaining $n-k$ elements. All the $n!$ permutations are thus split into groups of size $(n-k)!$ so there are $\frac{n!}{(n-k)!}$ such groups. We denote the number of ordered subsets by

$$P(n,k) = \frac{n!}{(n-k)!}.$$

Note that this number can also be written as $n \cdot (n-1) \cdots (n-k+1)$, which hints at an alternative way of computing these numbers. We can perform the ordering and choosing of elements at the same time. The first element of our sequence can be any of the n elements of the set, the next element any but the first (leaving us with $n-1$ choices), and so on. In contrast to permutations, we stop after choosing the k'th element, which we can do in $(n-k+1)$ ways.

Binomial Coefficients

Ordered subsets are far less interesting than their counterpart *unordered* k-subsets. The numbers of unordered subsets of size k taken from an set of size n are called the ***binomial coefficients***, and are one of the most useful combinatorial number there is.

To compute the number of k-subsets of a set of size n, start with all the $P(n,k)$ *ordered* subsets. Any particular unordered k-subset can be ordered in exactly $k!$ different ways. Hence, there must be $\frac{P(n,k)}{k!}$ unordered subsets, by the same grouping argument used when determining $P(n,k)$ itself. For example, consider again the ordered 2-subsets of the

set $\{a, b, c, d\}$, of which there are 12:

ab	ba	ac	ca
ad	da	bc	cb
bd	db	cd	dc

The subset $\{a, b\}$ can be ordered in 2! ways – the ordered subsets ab and ba. Since each unordered subset is responsible for the same number of ordered subsets, we find the number of unordered subsets by dividing 12 with 2!, giving us the 6 different 2-subsets of $\{a, b, c, d\}$: $\{a, b\}$, $\{a, c\}$, $\{a, d\}$, $\{b, c\}$, $\{b, d\}$ and $\{c, d\}$.

Definition 18.4 — Binomial Coefficient
The number of k-subsets of an n-set is called the ***binomial coefficient***, written as

$$\binom{n}{k} = \frac{n!}{k!(n-k)!}.$$

This is read as "n choose k", and sometime written as $C(n, k)$ or C_k^n.

Note that

$$\binom{n}{k} = \frac{(n-k+1)\cdot(n-k+2)\cdots(n-1)\cdot n}{1\cdot 2\cdots(k-1)\cdot k}.$$

They are thus the product of k numbers, divided by another k numbers. With this fact in mind, it does not seem unreasonable that they should be computable in $O(k)$ time. Naively, one might try to compute them by first multiplying the k numbers in the nominator, then the k numbers in the denominator, and finally divide them.

Unfortunately, both of these numbers grow quickly. Indeed, already at 21! we have outgrown a 64-bit integer. Instead, we compute the binomial coefficient by alternating multiplications and divisions. We start with storing $1 = \frac{1}{1}$. Then, we multiply with $n-k+1$ and divide with 1, leaving us with $\frac{n-k+1}{1}$. In the next step we multiply with $n-k+2$ and divide with 2, having computed $\frac{(n-k+1)\cdot(n-k+2)}{1\cdot 2}$. After doing this r times, we are left with our binomial coefficient. This doesn't overflow a 64-bit unsigned integer until computing $\binom{63}{28}$ (i.e., all binomial coefficients with $n \leq 62$ fits).

One big question mark remains – why must the intermediate results always be integer? This must be true if our procedure is correct, or we will at some point perform an inexact integer division, leaving us with an incorrect intermediate quotient. If we study the partial results more closely, we see that they are binomial coefficients themselves, namely $\binom{n-k+1}{1}$, $\binom{n-k+2}{2}, \ldots, \binom{n-1}{k-1}, \binom{n}{k}$. Certainly, these numbers must be integers. As we just showed, the binomial coefficients count things, and counting things tend to result in integers.

As a bonus, we discovered another useful identity in computing binomial coefficients:

$$\binom{n}{k} = \frac{n}{k}\binom{n-1}{k-1}.$$

Exercise 18.14. Prove this identity combinatorially, by first multiplying both sides with k.

Exercise 18.15. Another option is to change the order of multiplication of the numerator to $n, n-1, \ldots, n-k+1$.

1. Is this better, i.e. overflows later?
2. This order of multiplication also works, in that all partial results are integers. They are also binomial coefficients. Prove this by finding a similar binomial coefficient identity as before, and prove it combinatorially in the same way.

We have one more useful trick up our sleeves. Currently, if we want to compute e.g. $\binom{10^9}{10^9-1}$, we have to perform $10^9 - 1$ operations. To avoid this, we exploit a symmetry of the binomial coefficient. Assume we are working with subsets of some n-element set S. Define a bijection from the subsets of S onto itself by taking complements. Since a subset T and its complement $S \setminus T$ are disjoint, we have $|S \setminus T| = |S| - |T|$. This means that every 0-subset is mapped bijectively to every n-subset, every 1-subset to every $(n-1)$-subset, and every r-subset to every $(n-r)$-subset.

However, if we can bijectively map r-subsets to $(n-r)$-subsets, there must be equally many such subsets. Since there are $\binom{n}{r}$ subsets of the first kind and $\binom{n}{n-r}$ subsets of the second kind, they must be equal:

$$\binom{n}{r} = \binom{n}{n-r}.$$

More intuitively, our reasoning is "choosing what r elements to include in a set is the same as choosing what $n-r$ elements to exclude". This is very useful in our example of computing $\binom{10^9}{10^9-1}$, since this equals $\binom{10^9}{1} = 10^9$. More generally, this enables us to compute binomial coefficients in $\Theta(\min(r, n-r))$ time instead of $\Theta(r)$.

Problem 18.16.

Locked Treasure	lockedtreasure
Binomial Coefficients	binomialcoefficients

Sjecista – sjecista

Croatian Olympiad in Informatics 2006/2007, Contest #2

In a convex polygon with N sides, line segments are drawn between all pairs of vertices in the polygon, so that no three line segments intersect in the same point. Some pairs of these inner segments intersect, however.

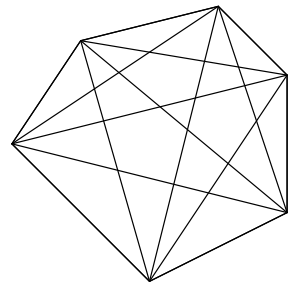


Figure 18.2: A polygon with 6 vertices. There are 15 intersecting segments.

Given $3 \leq N \leq 100$, determine how many pairs of segments intersect.

Solution. A classical counting problem, sometimes given with the alternative formulation to count the number of regions formed in the polygon. If we compute the answer by hand starting at $N = 0$, we get $0, 0, 0, 0, 1, 5, 15, 35$. A quick lookup on OEIS[3] suggests that the answer is the binomial coefficient $\binom{N}{4}$. While searching online certainly is a legit strategy when solving problems, it's not a particularly insightful approach, nor is it useful at contests where access to the Internet is restricted.

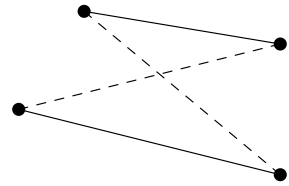


Figure 18.3: Four points taken from Figure 18.2.

Instead, let us find some kind of bijection between the objects we're interested in (intersections of line segments) and something easier to count. This strategy is one of the basic principles of combinatorial counting. An intersection is defined by two line segments, of which there are $\binom{N}{2}$. Does every pair of segments intersect? In Figure 18.3, two segments (the solid segments) do not intersect. However, two other segments which together have the same four endpoints *do* intersect with each other. This suggests that line segments was the wrong level of abstraction when finding a bijection. On the other hand, if we choose a set of four points, the segments formed by the two diagonals in the convex quadrilateral given by those four points will intersect at some point (the dashed segments in Figure 18.3).

Conversely, any intersection of two segments give rise to such a quadrilateral – the one given by the four endpoints of the segments that intersect. This is a bijection between

[3] https://oeis.org/A000332

intersections and quadrilaterals, meaning that there must be an equal number of both. There are $\binom{N}{4}$ such choices of quadrilaterals, and thus equally many intersections. □

Exercise 18.17. Prove that

$$\binom{n}{k} = \binom{n-1}{k-1} + \binom{n-1}{k}.$$

This exercise also demonstrates another way of computing binomial coefficients up to $\binom{n}{k}$ in $\Theta(nk)$ time. It has a better overflow behavior than the previous method since it never needs to compute values larger than the final result. It's commonly used when you need the result of all binomial coefficients with n up to a limit (normally computed modulo something).

Exercise 18.18. Prove that

1. $\sum_{k=0}^{n} \binom{n}{k} = 2^n$
2. $\sum_{k=0}^{n} (-1)^k \binom{n}{k} = 0$ $(n > 0)$
3. $\sum_{k=0}^{n} \binom{n}{k} 2^k = 3^n$
4. $\sum_{i=k}^{n} \binom{i}{k} = \binom{n+1}{k+1}$

Exercise 18.19. By the previous exercise, $\binom{n}{\frac{n}{2}} = O(2^n)$, but what's the lower bound on the growth of the central binomial coefficient? Prove that $\binom{n}{\frac{n}{2}} = \Omega(\frac{2^n}{n})$.

A stronger result is that $\binom{n}{\frac{n}{2}} = \Theta(\frac{2^n}{\sqrt{n}})$. This follows by e.g. Stirling formula.

Exercise 18.20. If k is fixed, how fast does $\binom{n}{k}$ grow asymptotically in terms of n?

Problem 18.21.

Election	election
Gnoll Hypothesis	gnollhypothesis

Dyck Paths

In a grid of width W and height H, we stand in the lower left corner at coordinates $(0, 0)$, wanting to venture to the upper right corner at (W, H). To do this, we are only allowed two different moves – we can either move one unit north, from (x, y) to $(x, y + 1)$ or one unit east, to $(x + 1, y)$. Such a path is called a ***Dyck path***.

As is the spirit of this chapter, we ask how many Dyck paths there are in a grid of size $W \times H$. The solution is based on two facts: a Dyck path consists of exactly $H + W$ moves, and exactly H are northbound (so W are eastbound). Conversely, any path consisting of exactly $H + W$ moves of which H are northbound moves is a Dyck path.

If we consider e.g. the Dyck path in Figure 18.4, we can write down the sequence of moves we made, with the symbol N for northbound moves and E for eastbound moves:

$$EENENNEEENEEN.$$

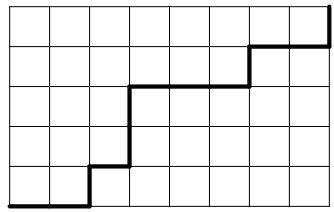


Figure 18.4: A Dyck path on a grid of width 8 and height 5.

Such a sequence must consist of all $H + W$ moves, with H "N"-moves. There are $\binom{H+W}{H}$ such sequences, since this is the number of ways we can choose the subset of positions which should contain the N moves.

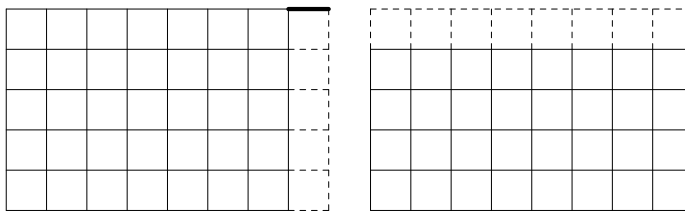


Figure 18.5: The two options for the last possible move in a Dyck path.

If we look at Figure 18.4, we can find another way to arrive at the same answer. Letting $D(W, H)$ be the number of Dyck paths in a $W \times H$ grid, some case work on the last move gives us the recurrence

$$D(W, H) = D(W - 1, H) + D(W, H - 1)$$

with base cases

$$D(0, H) = D(W, 0) = 1.$$

We introduce a new function D', defined by $D'(W + H, W) = D(W, H)$. This gives us the recurrence

$$D'(W + H, H) = D'(W - 1 + H, W - 1) + D'(W + H - 1, H - 1)$$

with base cases

$$D'(0, 0) = D'(H, H) = 0.$$

These relations are satisfied by the binomial coefficients (Exercise 18.17).

Exercise 18.22. Prove that $\sum_{i=0}^{n} \binom{n}{i}\binom{n}{n-i} = \binom{2n}{n}$.

While Dyck paths sometimes do appear directly in problems, they are also a useful tool to find bijections to other objects.

Sums

Given integers $1 \le N, K \le 10^5$, compute the number of solutions to the integer equations

$$a_1 + a_2 + \cdots + a_N = K$$

where $0 \le a_1$. Since the answer may be large, compute it modulo $10^9 + 7$.

Solution. Given a Dyck path such as the one in Figure 18.4, what happens if we count the number of northbound steps we take at each x-coordinate? There are a total of $W + 1$ coordinates and H northbound steps, so we except this to be a sum of $W+1$ (non-negative) variables with a sum of H. This is indeed similar to what we are counting, and Figure 18.6 shows this connection explicitly.

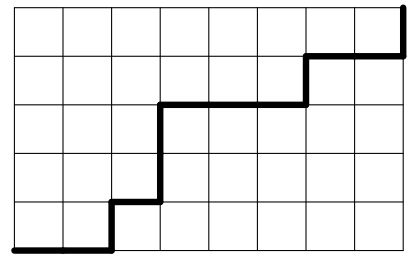


Figure 18.6: A nine-term sum as a Dyck path.

This mapping gives us a bijection between sums of k terms with a sum of n and Dyck paths on a grid of size $(k-1) \times n$. We already know how many such Dyck paths there are: $\binom{n+k-1}{n}$.

What remains is only to compute

$$\binom{n+k-1}{n} = \frac{(n+k-1)!}{n!(k-1)!}$$

modulo $10^9 + 7$. So far, we haven't tried computing binomial coefficients this large. In this particular case, it's easy enough to use the normal method of computing factorials, and then taking modular inverses to with them. Since $10^9 + 7$ is prime and $n + k - 1$ is smaller than that, it is guaranteed that both factorials have inverses. □

We learned that one way of looking at the binomial coefficient $\binom{n}{k}$ is as the number of ways to order an $(a + b)$-character string of a letters A, and b letters B. A different way to view the binomial coefficient formula in this context is that we have a string of $a + b$ letters which can be permuted in $(a + b)!$ ways, but then each identical string appears $a! \cdot b!$ times, since that's the number of ways the A's and B's can be permuted to produce

the same string. In the same way we can count the number of different strings with c_i copies of the letter i. This is called the ***multinomial coefficient***, and is written as

$$\binom{c_1 + c_2 + \cdots + c_n}{c_1, c_2, \ldots, c_n}.$$

Problem 18.23. *Anagram Counting* anagramcounting

Catalan Numbers

A special case of Dyck paths is paths on a square grid that *do not cross the diagonal* of the grid. See Figure 18.7 for an example.

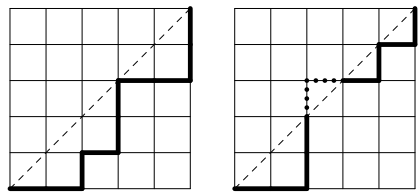


Figure 18.7: A valid path (left) and an invalid path (right).

We are now going to count the number of such paths, the most complex counting problem we have encountered so far. It turns out that there is a straightforward bijection between the *invalid* Dyck paths, i.e. those who do cross the diagonal of the grid, to Dyck paths in a grid of different dimensions. In Figure 18.7, the right grid contained a path that cross the diagonal. If we take the part of the grid just after the first segment that crossed the diagonal and mirror it in the diagonal translated one unit upwards, we get the situation in Figure 18.8.

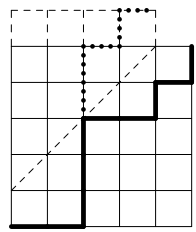


Figure 18.8: Mirroring the part of the Dyck path after its first diagonal crossing.

We claim that when mirroring the remainder of the path in this translated diagonal, we always get a new Dyck path on the grid of size $(n-1) \times (n+1)$. Assume that the first crossing is at the point (c, c). Then, after taking one step up in order to cross the diagonal, the remaining path goes from $(c, c+1)$ to (n, n). This needs $n - c$ steps to the right and $n - c - 1$ steps up. When mirroring, this instead turns into $n - c - 1$ steps

up and $n - c$ steps right. Continuing from $(c, c+1)$, the new path must thus end at $(c + (n-c-1), c+1+(n-c)) = (n-1, n+1)$. This mapping is also bijective.

This bijection lets us count the number of paths that do cross the diagonal: they are $\binom{2n}{n+1}$. The numbers of paths that does not cross the diagonal is then $\binom{2n}{n} - \binom{2n}{n+1}$.

Definition 18.5 — Catalan Numbers

The number of Dyck paths in an $n \times n$ grid is called the n'th ***Catalan number***, equal to

$$C_n = \binom{2n}{n} - \binom{2n}{n+1} = \binom{2n}{n} - \frac{n}{n+1}\binom{2n}{n} = \frac{1}{n+1}\binom{2n}{n}.$$

The first few Catalan numbers[4] are $1, 1, 2, 5, 14, 42, 132, 429, 1430$.

Problem 18.24. *Catalan Numbers* catalan

Catalan numbers count many other objects, most notably the number of *balanced parentheses* expressions. A balanced parentheses expression is a string of $2n$ characters $s_1 s_2 \dots s_{2n}$ of letters (and), such that every prefix $s_1 s_2 \dots s_k$ contain *at least* as many letters (as). Given such a string, like (()())(()) we can interpret it as a Dyck path, where (is a step to the right, and) is a step upwards. Then, the condition that the string is balanced is that, for every partial Dyck path, we have taken at least as many right steps as we have taken up steps. This is equivalent to the Dyck path never crossing the diagonal, giving us a bijection between parentheses expressions and Dyck paths. The number of such parentheses expressions are thus also C_n.

Sometimes, it can be easier to reason about one of these alternatives proving bijections with Catalan numbers.

Fiat – fiat

By Steven Halim. NUS Competitive Programming. CC BY-SA 3.0. Shortened.

The king of Rectangle Land has just received a royal gift, a stairstep-shaped piece of wood with $1 \le N \le 100\,000$ steps. The king accepted the gift out of courtesy, but he actually has a disdain for stairstep-shaped pieces of wood with N steps. He decided to give you, his royal advisor, a royal fiat to divide this stairstep-shaped piece of wood with N steps into more reasonable shapes: N rectangles with integer dimension. In how many ways, modulo $10^9 + 7$, can this be done?

[4] https://oeis.org/A000108

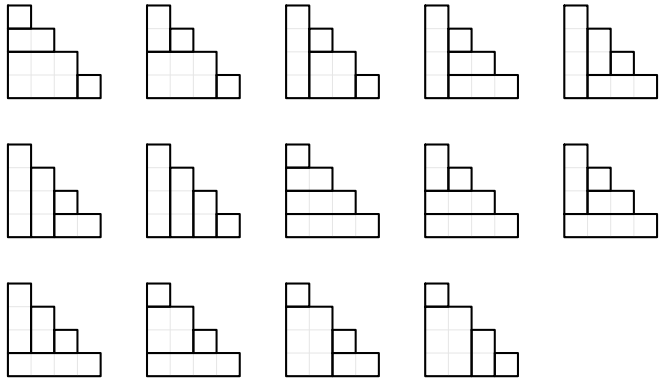


Figure 18.9: All the possible divisions of a 4 step staircase into 4 rectangles.

Solution. First, a solution that comes from a lot of experience. The online judge version of the problem gives the sample cases $N = 4, 7$ with answers 14, 429. For the experienced combinatorist, just seeing those outputs is enough to ring the Catalan bell and immediately submit a solution assuming that the answer is C_n in a contest[5].

If we can not immediately guess that the answer is the Catalan numbers based on this, we must instead come to suspect this based on some intuition. In this case, if one squints really hard, it is possible to map sequences of balanced parenthesis with a rectangle division. The height of the topmost rectangle in the i'th column tells us how many bracket pairs that the i'th bracket pair (counting in order of starting brackets) should contain, including itself. For example, the second division on the last row in Figure 18.9 can be interpreted like this. The first bracket pair contains only itself. The second bracket pair should also contain the third bracket pair. The third bracket pair should contain only itself. The fourth bracket pair should contain only itself. Thus, the bracket sequence is $()(())()$. Similarly, the third division on the last row would represent $()(()())$. It is slightly difficult to see why no division can generate an unbalance parenthesis sequence, and vice versa, why any sequence of rectangle heights corresponding to a bracket sequence in this way even results in a valid division of the staircase. This is good combinatorial practice, however.

Exercise 18.25. Prove that the above construction is a bijection between rectangle divisions and balanced parenthesis sequences.

For completeness, we show a maybe slightly more natural way of solving the problem. We have previously counted combinatorial objects by finding a recursion for the number in terms of smaller objects of the same sort. Here, this would mean finding a recursion

[5]One of the world's best competitive programming coaches, Andrew Stankevich, is famous for creating combinatorial problems where the answers for small N in the sample cases starts out with 1, 2, 5, 14, 42, but then diverge. This has baited many contestants throughout the years to incorrectly submit Catalan number solutions on his contests.

for the number of divisions of an N-step shape $c(N)$ using smaller values of $c(N)$. If the recursion is sufficiently fast, this can allow a dynamic programming solution.

Something that all the examples for $N = 4$ in Figure 18.9 share is that the rectangle starting in the bottom-left corner of the figure always have one of the four diagonal squares as its top-right corner. If this is the case in general, we could definitely create some kind of recursion. Removing this rectangle from the figure would actually divide the shape into to new staircases: one immediately above the rectangle, and one to the right (except for the edge cases where we only get a single staircase). To see that is indeed true we use another observation from the examples. All the squares that form the diagonal are necessarily the top-right corner N of *different* rectangles. Since there are only N rectangles in total, one of must have the bottom-left square as corner.

To form the recursion we apply the addition and multiplication principles. If the rectangle has its corner in the i'th diagonal square counted from the top, the top shape is a staircase with $i - 1$ steps, and the right one with $N - i$ steps. Summing over all such corners, we get the recursion

$$c(N) = \sum_{i=1}^{N} c(i-1)c(N-i).$$

Unfortunately this path was not sufficient to solve the problem without more Catalan knowledge[6]. Evaluating this recursion up to N takes quadratic time, and N is quite large.

Figuring out by intuitive means that this recursion generates the Catalan sequence with the correct base case ($c(0) = 1$, the empty staircase, divided into an empty set of rectangles), is hard. We could perhaps see the similarity to the identity

$$\binom{2n}{n} = \sum_{i=0}^{n} \binom{n}{i}\binom{n}{n-i}$$

from one of the exercises on Dyck paths to guess that a similar interpretation and proof strategy could work (although this is reaching a bit). A proof by Dyck path's is quite straightforward: if $c(N)$ is the number of Dyck paths that do not cross the diagonal, look at which column the path *first touches* the diagonal. Now count the number of paths from that point to the start and end corners in the grid. □

Problem 18.26.

Fiat	fiat
Catalan Square	catalansquare

[6] With a wider toolbox containing e.g. generating functions we could actually have determined a closed form for the recursion by hand and see that it results in the Catalan numbers.

The Binomial Theorem

We end here with a short note on why binomial coefficients are named just so. The background is the expression $(x+y)^n$, and what happens when you multiply it out. For the first few n, you get the expressions

$$\begin{aligned}(x+y)^0 &= 1x^0y^0\\ (x+y)^1 &= 1\cdot x^1y^0 + 1\cdot x^0y^1\\ (x+y)^2 &= 1\cdot x^2y^0 + 2\cdot x^1y^1 + 1\cdot x^0y^2\\ (x+y)^3 &= 1\cdot x^3y^0 + 3\cdot x^2y^1 + 3\cdot x^1y^2 + 1\cdot x^0y^3.\end{aligned}$$

Here we've written out perhaps more coefficients and exponents than necessary to drive home a point, which is that in the expression $(x+y)^n$, the term x^iy^{n-i} has the coefficient $\binom{n}{i}$. This is a well-known theorem:

Theorem 18.1 — Binomial Theorem

For non-negative integers n,

$$(x+y)^n = \sum_{i=0}^{n}\binom{n}{i}x^iy^{n-i}.$$

Proof. In the expression $(x+y)^n$, each term in the multiplied out expression has i factors x, and $n-i$ factors y, depending on which term was chosen within each factor $(x+y)$. There are $\binom{n}{i}$ ways to choose exactly i factors where x should be picked. □

The theorem gives algebraic proofs of some well-known combinatorial identities by inserting, for example, $x=y=1$, $x=1$ and $y=-1$, $x=2$ and $y=1$.

18.4 The Principle of Inclusion and Exclusion

The addition principle tells us how to compute the union of disjoint sets. For the situation that they aren't disjoint, we have the ***principle of inclusion and exclusion***.

The simplest case of the principle is for two sets A and B. To compute the size of their union $|A\cup B|$, we *at least* need to count every element in A and every set in B, i.e. $|A|+|B|$. The problem with this formula is that whenever an element is in *both* A and B, we count it twice. This is easily mitigated: the number of elements in both sets equals $|A\cap B|$ (Figure 18.10). Thus, we see that $|A\cup B| = |A|+|B|-|A\cap B|$.

Similarly, we can determine a formula for the union of three sets $|A\cup B\cup C|$. We begin by including every element: $|A|+|B|+|C|$. Again, we have included the pairwise intersections too many times, so we remove those and get

$$|A|+|B|+|C|-|A\cap B|-|A\cap C|-|B\cap C|.$$

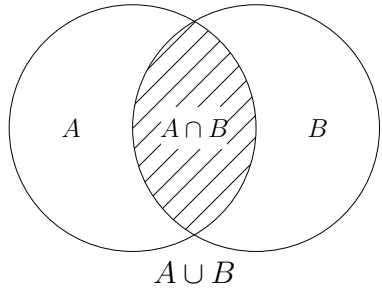


Figure 18.10: The union of two sets A and B.

This time we are not done. While we have counted the elements which are in exactly one of the sets correctly (using the first three terms), and the elements which are in exactly two of the sets correctly (by removing the double-counting using the three latter terms), we currently do not count the elements which are in all three sets at all! Thus, we need to add them back, which gives us the final formula:

$$|A \cup B \cup C| = |A| + |B| + |C| - |A \cap B| - |A \cap C| - |B \cap C| + |A \cap B \cap C|.$$

Exercise 18.27. Compute the number of integers between 1 and 1000 that are divisible by 2, 3 or 5.

From the two examples, you can probably guess formula in the general case, which we write in the following way:

$$\left|\bigcup_{i=1}^{n} S_i\right| = \sum_i |S_i| - \sum_{i<j} |S_i \cap S_j| + \sum_{i<j<k} |S_i \cap S_j \cap S_k| - \cdots + (-1)^{n+1} |S_1 \cap S_2 \cap \cdots \cap S_n|.$$

Exercise 18.28. Prove the general case of the inclusion-exclusion formula.

From this formula, we see the reason behind the naming of the principle. We *include* every element, *exclude* the ones we double-counted, *include* the ones we removed too many times, and so on. For the principle to be useful, a very important assumption must hold – that it is hard to compute unions of but easy to compute their intersections. Whenever this is the case, you might want to consider if the principle is applicable.

Secret Santa – secretsanta

By James Stanley. United Kingdom and Ireland Programming Contest 2016. CC BY-SA

Christmas comes sooner every year. In fact, in one oft-forgotten corner of the world, gift-giving has already started in the form of a Secret Santa syndicate.

The $1 \leq N \leq 10^{12}$ citizens of Haircombe are going to put their name into a hat. This hat will be given a hearty shuffle, and then afterwards everybody will take turns once more in taking a name back from the hat. The name each person receives is the name of the fellow citizen to whom they

will send a gift.

One concern with this strategy is that some unfortunate citizens could wind up giving gifts to themselves. Compute the probability that this will happen to any of the citizens of Haircombe, within an absolute or relative error of at most 10^{-6}.

Solution. A given way of distributing the names to everyone can be modeled as a permutation, representing who's present should be given to whom. We are then looking for a permutation π such that $\pi(i) \neq i$ for all i. This is called a ***derangement***.

Counting derangements is a typical application of inclusion-exclusion. We will use it on those sets of permutations where the condition is false for *at least* a particular index i. If we let these sets be D_i, the set of all permutations where the condition is false for any i is $D_1 \cup D_2 \cup \cdots \cup D_N$. The answer we look for is $N! - |D_1 \cup \cdots \cup D_N|$. To apply the inclusion and exclusion formula, we must be able to compute the size of intersections of a collection of D_i. This task is simplified greatly since the intersection of k such subsets is entirely symmetrical (it does not matter which elements violate the condition, only the amount).

For a permutation that's the intersection of k subsets D_i, there are (at least) k indices i where $\pi(i) = i$. Furthermore, there are $N - k$ other elements, which can be arranged in $(N-k)!$ ways, so the intersection of these sets have size $(N-k)!$. Since we can choose which k elements that should be fixed in $\binom{N}{k}$ ways, the term in the formula where we compute all k-way intersections evaluates to $\binom{N}{k}(N-k)! = \frac{N!}{k!}$. Thus, the formula can be simplified to

$$\frac{N!}{1!} - \frac{N!}{2!} + \frac{N!}{3!} - \dots$$

This gives us a $\Theta(N)$ algorithm to compute the answer. Since we only care about the probability of choosing a derangement, we ignore the $N!$ numerators in the sum. Note that while N is very large, after computing 10^6 terms, you're guaranteed to be within 10^{-6} of the answer.

We'll take a close look on why this is true – the error approximation gives us a neat result. Count the number of permutations that aren't derangements, i.e.

$$N!(1 - 1 + \frac{1}{2!} - \frac{1}{3!} + \dots).$$

From calculus, we know that[7]

$$e^{-1} = 1 - 1 + \frac{1}{2!} - \frac{1}{3!} + \dots$$

For large N we expect that the answer should converge to $\frac{N!}{e}$. It will in fact *always* be $\frac{N!}{e}$ rounded to the nearest integer since the value of the extra terms in the product equals

$$\left| \frac{1}{N+1} - \frac{1}{(N+1)(N+2)} + \dots \right| < \left| \frac{1}{N+1} \right| \leq 0.5.$$

[7] If you don't about e or this fact, don't fret. Just ignore this for now, take a single-variable calculus course, and then come back here and marvel.

To see why the above is true, note that each partial sum starting with the third lies strictly between the previous two partial sums. □

A lot of combinatorial counting problems are more math than programming. So far, we have mostly needed to add DP to some recursions, but that's basically the only programming specific part of any problem. In this next problem, we work through some of the other more algorithmic aspects that can pop up when counting.

Classical Counting – classicalcounting

By Syx Pek. ICPC SG Preliminary Contest 2018. CC BY-SA.

Given integers $1 \leq N, M, K \leq 10^5$, compute the number of solutions to the integer equation

$$a_1 + a_2 + \cdots + a_N = K$$

where $0 \leq a_i \leq M$. Since the answer may be huge, compute it modulo $10^6 + 7$.

Solution. The problem looks deceptively simple. If N, M, K would just have been much smaller, say up to 10^3 instead, the problem could have been solved with DP by adding one term at a time, plus a bit of prefix sum magic (think a bit about this – it too makes for a nice problem).

Back when we learned about Dyck paths we solved *almost* the same problem – the difference being that there was no upper bound on M. This, of course, is the same situation as for Secret Santa. We knew how to compute the number of permutations, so we tried to get rid of the extra restriction that no element remained in place. If we try the same approach here, we should try to see what happens if we make sets of all solutions where a given constraint is violated (i.e. a given variable is $> M$). If $a_i > M$, we can substitute it with a new variable $b_i = a_i - (M+1) \geq 0$ and let the sum instead be $K - (M+1)$. This substitution leaves us with the problem we already know how to solve.

Of course, for inclusion-exclusion to apply, it must also be simple to solve the problem when any given set of constraints are violated. Performing the same substitution on any l variables simply changes the sum to be for $K - l(M+1)$. We can now apply the inclusion-exclusion formula. First, we include all solutions: $\binom{N+1+K}{N+1}$. Then, we must subtract the number of solutions where at least one constraint is violated. There are $\binom{N+1+K-(M+1)}{N+1}$ such solutions, but also $\binom{N}{1}$ ways to pick what constraint is violated. Next, add back the cases where at least two constraints are violated, and so on. In total, the answer will be

$$\sum_{l=0}^{N} \binom{N}{l} \binom{N+1+K-l(M+1)}{N+1}.$$

One final annoyance remains. As in Sums we're supposed to output the answer modulo something – always a hassle when we need to perform divisions. The situation here is much worse however – 10^6+7 is not a prime. In general this is not that bad. By the Chinese

Remainder Theorem, we can solve the problem modulo each prime power of the modulus and combine answers, so this only adds an extra logarithmic factor to the time complexity. Here however, the modulo is the product of two integers much smaller than what we need to compute binomial coefficients for (29 and 34483). A consequence is that we have to divide even by integers that *don't have inverses* modulo our modulus. In reality the result are of course never really divided by this, but we can't use the method of precomputing inverse factorials to use for divisions.

The trick is to factor out the modulus m whenever it appears in the factorials. The idea is that if we have an integer a, we write it as $m^k \cdot a'$, where a' is the m-free part of a. Multiplying and dividing such numbers are easy: two integers $m^{k_1} \cdot a_1$ and $m^{k_2} \cdot a_2$ have the product $m^{k_1+k_2} \cdot a_1 \cdot a_2$ and quotient $m^{k_1-k_2} \cdot a_1 \cdot a_2^{-1}$. Since binomial coefficients only involve factorials, it's enough to factor out m from each factor in the factorial and keep track of the accumulated k values for each of them. Finally, if the binomial coefficient equals $m^k \cdot a$ where $k > 0$, it's simply equal to 0 in the rest of the computation. □

Problem 18.29.
Tom and Jerry tomjerry

A less obvious application of inclusion-exclusion is on the primes or prime powers of an integer. We can derive the formula for $\phi(n)$ in an easy way with inclusion-exclusion, without any need for the Chinese Remainder Theorem. If an integer a is not coprime to n, it must have some set of prime divisors in common with a. Specifically, if those primes are $q_1, \ldots, q_l$, there are $\frac{n}{q_1 q_2 \cdots q_l}$ such integers. Applying inclusion-exclusion where we look at which primes a have in common with $n = \prod_{i=1}^{k} p_i^{e_i}$, we find that

$$\phi(n) = n - \sum \frac{n}{p_i} + \sum_{i<j} \frac{n}{p_i p_j} - \ldots$$

Factoring out n and simplifying this leaves us with

$$n(1 - \frac{1}{p_1})(1 - \frac{1}{p_2}) \cdots (1 - \frac{1}{p_k}).$$

If you split n up into its prime powers and multiply them with their corresponding $(1 - \frac{1}{p_i})$ terms, you get the formula we already know.

Problem 18.30.
Winning Tickets winningtickets

18.5 Invariants

Many problems deal with processes that consist of a series of steps. During a process, we are often interested in certain properties that never change. We call such a property an *invariant*. You have already encountered invariants several times before. They are tightly

attached to greedy algorithms, and is a common tool used in proving the correctness of various greedy algorithms. Also, the binary search algorithm maintains the invariant that the value we are searching for must be contained in some given segment $[lo, hi)$ of the array at any time – a fact that basically constitutes the entire proof of correctness of binary search. They even made an appearance as recently as Chapter 16, when we found winning invariants in games. Hence this section should be quite familiar to you, but hopefully you'll learn some different uses of invariants.

Coin Stacks – coinstacks

By Antti Laaksonen. Nordic Collegiate Programming Contest (NCPC) 2020. CC BY-SA.

A and B are playing a collaborative game that involves $2 \leq n \leq 50$ stacks of coins, with at most 1000 coins in total. Every round of the game, they select a nonempty stack each, but they are not allowed to choose the same stack. They then remove a coin from both the two selected stacks and then the next round begins. The players win the game if they manage to remove all the coins. Is it possible for them to win the game, and if it is, how should they play?

Proof. Our first invariant is also the most obvious one – parity. Since A and B always removes two coins at once, the parity of the number of remaining coins is invariant after making moves. Since the end state – no coins – is an even amount, there must also be an even amount in the beginning.

This is not sufficient, however. If you consider the case $(1, 1, 4)$, you will quickly figure out that there is no way to remove all the coins. No matter what you do, there are too many coins in the third pile to remove them all, with the help of the other stacks. In general, if a stack has a coins, and all the remaining stacks together have fewer than a coins, this fact is invariant under making a move, since you can't remove more coins from the larger stacks than the smaller.

This last condition is sufficient, however. □

Exercise 18.31. Prove that as long as no stack has more coins than the other stacks combined, it is possible to remove all coins, and give a strategy for this.

There are always two conflicting ideas that must be balanced on this kind of problem. On one hand, you are trying to identify sufficient conditions for possibility. This typically means working towards an algorithm that actually solves the problem, i.e. sorts the grid. The other part is finding the necessary conditions for when something is possible, i.e. determining what makes that algorithm fail. In Coin Stacks, once you came up with one of the directions, the other was fairly obvious. In the most difficult problems, you will instead constantly struggle in knowing which of your necessary and sufficient conditions are closest to the truth. Our next problem comes from the land of permutations, and becomes easy only once experience tells you what invariants give you necessary conditions that *usually* are sufficient as well.

Grid Crossings – gridcrossing

Given is an $N \times M$ grid ($3 \leq N, M \leq 1000$), where the integers 1 through NM are written in the cells of the grid, one in each grid square. The grid can be transformed by taking a 2×2 subgrid and swapping the integers in the two diagonals, such that the top-left and bottom-right integers switch places, as well as the top-right and bottom-left. Can the grid be transformed so that all the integers are sorted, when read row-by-row from left-to-right?

Solution. The invariants in this problem are again ones of parity. The first one pops up pretty easily. In all of these permutation-type problems, a good approach is first to try and see if it's easy to construct an algorithm to perform arbitrary transformations, such as finding a sequence of operations that move only a single element one step. Another common approach is trying to correct the grid row-by-row. It's sometimes easy to move elements arbitrarily upwards in the grid to a row, while not disrupting any element further up in that row, often reducing the problem to solving the general problem only for a few rows in the bottom.

If you try to design a way to move e.g. 1 into the right place from an arbitrary position, an obstacle quickly pops up. Integers can only move diagonally in the grid, so they are constrained to either squares with an odd or an even coordinate sum. You can think about this as the grid being colored in a black and white chessboard pattern, and integers only moving between same colored squares. The puzzle can be solved only when the integers that are on the white squares must also have their target positions on their white squares.

Is this sufficient, then? Nope, but it's not obvious how to know except by getting Wrong Answer on an online judge. If you'd brute force count the number of 3×3 grid that can be sorted (a good idea in this kind of problems!), you'd find that there are exactly 1440 such grids. In total, there are $5! \cdot 4! = 2880$ possible grids (permute all integers on the black and the white squares), i.e. exactly half are reachable. No matter what you do to one of the other 1440 grids, you'll notice that there is always one pair of out-of-order integers on either the white or the black squares.

To understand why, we bring out some additional combinatorial theory, namely that of the *parity* of a permutation. The basic idea is the following. Take a permutation and repeatedly swap the positions of two elements an arbitrary **even** number of times. Not all permutations can be reached through this process. For example $\langle 1, 2 \rangle$ can never turn into $\langle 2, 1 \rangle$, and $\langle 1, 2, 3 \rangle$ can never turn into $\langle 3, 2, 1 \rangle$. It is these two classes we are after: the permutations that can or can't be obtained. To characterize them, we use one more concept.

Definition 18.6 — Inversions and parity of a permutation

In a permutation π, we call a pair of indices $i < j$ an ***inversion*** if $\pi(i) > \pi(j)$, i.e. the corresponding elements are out-of-order in relation to each other.

The ***parity of a permutation*** is the parity of the number of of inversions in the

> permutation. We call the classes of permutations ***odd*** and ***even*** permutations.

To tie this tangent together, we need a hard to come up with fact: the parity of a permutation changes when swapping any two elements.

Exercise 18.32. Prove that the parity of a permutation changes when two elements are swapped.

Exercise 18.33. Give a simple $\Theta(N \log N)$ time algorithm to compute the the number of inversions of a permutation (and thus also the parity of the permutation).

This concept allows us to formulate a new necessary condition for when a grid can be sorted. The integers on the black and white squares can be considered as two permutations of their sorted positions. Note now that each move performs exactly one swap on both of those permutations. This means that their parity is changed simultaneously each time. The end state of both permutations is the identity permutation which is always even, so they must have the same parity at every point during the process – more specifically, in the beginning. This explains why only half of the 3×3 grids can be solved. In the other half, the parity of the two permutations don't match.

If you think that this must surely be a sufficient condition, you'd be absolutely right. Proving this, however, is left as an exercise. □

Exercise 18.34. Prove that if all integers in the grid are on a square of the same color as its target square, and the permutation formed by all the integers on the white and black squares respectively, the grid can be sorted.

Exercise 18.35. Prove that a permutation is even if and only if it has an even number of even-length cycles in its cycle decomposition. Give a linear-time algorithm to compute the parity of a permutation.

Exercise 18.36. Prove that there are as many odd permutations as there are even permutations when the permutation has at least 2 elements.

Problem 18.37.

Bread Sorting	breadsorting
Grid Transpositions	gridtranspositions

Monovariants

Another similar tool is the ***monovariant***. While invariants are used to prove that some property doesn't change during a process, a monovariant instead aims to show that some property of the process always changes in the same direction. As an example, consider the Euclidean algorithm for computing the GCD. It is a step-by-step process on non-negative integers a, b, that terminates when once is zero. The proof that it terminates essentially amounts to showing that the value $a + b$ strictly decreases with every move operation

$(a, b) \to (b, a \bmod b)$. Since $a + b$ is bounded downwards by 0, this procedure could not continue forever, so it must at some point terminate.

This idea can be generalized to many situations where we can assign some kind of value $p(v)$ to the state v of an iterative process. We try to choose p such that it always strictly increases or decreases after each move. If we can prove that p only assumes values from a finite set (in the GCD case, integers between 0 and $a + b$), it follows that the process terminates, or else p, taking only a finite number of different values, must take some value twice. That contradicts p's monotonicity.

Majority Graph Bipartitioning

Given is a graph G with at most $1 \le V \le 100$ vertices. Find a bipartition of this graph into parts A and B, such that every vertex v has at most $\frac{|N(v)|}{2}$ neighbors in the same part as v itself.

Solution. A first approach might be to attempt something greedy. For example, one might try to construct a greedy algorithm based on the degrees of the vertices, or something like that. Unfortunately, the problem does not have enough structure for any simple greedy idea to work.

Instead, we will attempt to use the most common monovariant attack. Roughly, the process follows these steps:

1. Start with any arbitrary state s.
2. Look for some kind of modification to this state, which is possible if and only if the state is not admissible. In this problem, admissible means that no vertex is in the same part as a strict majority of its neighbors. Generally, the goal of this modification is to "fix" whatever makes the state inadmissible.
3. Prove that there is some value $p(s)$ that must decrease/increase whenever such a modification is done.
4. Prove that this value cannot decrease/increase infinitely many times.

Using these four rules, we prove the existence of an admissible state. If (and only if) s is not admissible, by step 2 we can perform some specified action on it, which by step 3 will decrease the value $p(s)$. Hence, by performing finitely many such actions, we must (by rule 4) reach a state where no such action is possible. This happens only when the state is admissible, meaning such a state must exist. The process might seem a bit abstract, but will become clear once we walk you through the bipartitioning step.

Our algorithm works as follows. First, consider any invalid bipartition of the graph such as in Figure 18.11. Since the bipartition is invalid, there must exist a vertex v which has more than $\frac{|N(v)|}{2}$ vertices in the same part as v itself. Move v to the other side of the partition. See Figure 18.12 for the result of the this process.

One question remains – why does this move guarantee a finite process? We now have a general framework to prove this, suggesting that we should look for a value function

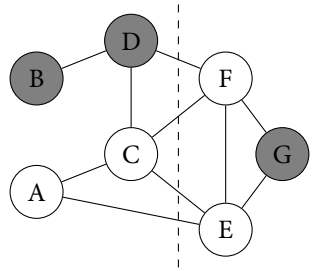


Figure 18.11: An invalid bipartitioning, where vertices B, D, G break the condition.

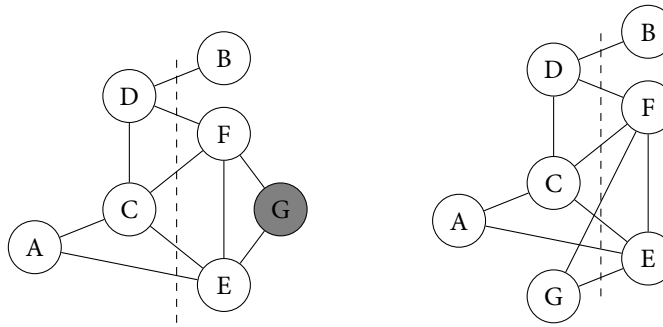


Figure 18.12: The results after of the algorithm, which brings the graph to a valid state.

$p(s)$ which is either strictly increasing or decreasing as we move a vertex to the other side. By studying the algorithm in action in Figure 18.12 we might notice that more and more edges tend to go between the two parts. This number always increased in our example. As we shall see, this is true no matter which vertex v is moved.

If a vertex v has a neighbors in the same part, b neighbors in the other part, and violates the neighbor condition, this means that $a > b$. When we move v to the other part, the b edges from v to its neighbors in the other part is no longer between the two parts, while the a edges to its neighbors in the same part are. The number of edges between the parts then changes by $a - b > 0$, making this a good choice for the value function. Since this is an integer function with the obvious upper bound of $\frac{N(N-1)}{2}$, we complete step 4 of our proof technique and conclude that the final state must be admissible. We also got a $O(N^2)$ upper bound on the number of steps needed. □

In mathematical problem solving, monovariants are usually used to prove that an admissible state exists. However, monovariant problems are really algorithmic problems in disguise since they tend to provide an algorithm for constructing an admissible state.

Water Pistols

N girls and N boys ($N \leq 200$) stand on a field, with no three children on the same line. Each girl is equipped with a water pistol, and wants to pick a boy to fire at. While the boys probably won't appreciate being drenched in water, at least the girls are fair – they will only fire at a single boy each. Sometimes, it may be the case that two girls fire on the boys in a way that makes the water from their pistols cross. If this happens, the beams collide mid-air and cancel each other out, never hitting their targets.

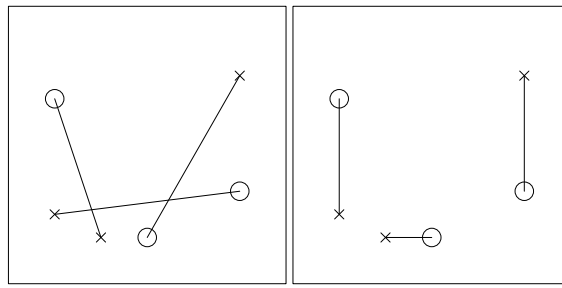


Figure 18.13: An assignment where some beams intersect (left), and an assignment where no beams intersect (right).

Help the girls choose which boys to fire at, in such a way that no two girls fire at the same boy, and the water fired by two girls will not cross.

Solution. After the last problem, the solution should not come as a surprise. Start by arbitrarily pairing up the boys with the girls. While the pairing has two girls whose water beams cross, swap their targets.

Unless you have great geometrical intuition, it may be hard to figure out an appropriate value function. A naive attempt is to count how many pairs of water beams cross. This doesn't necessarily decrease after a move – it might even increase.

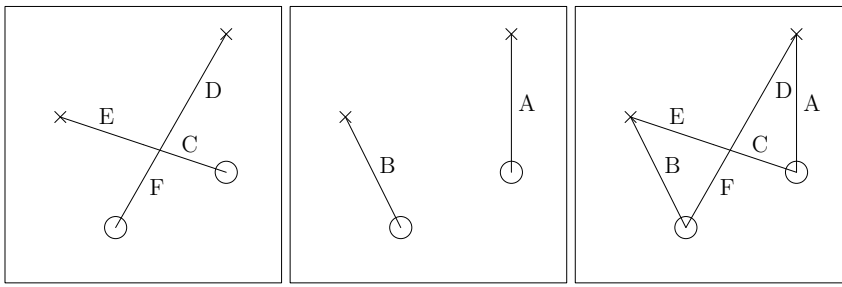


Figure 18.14: Swapping the targets of two intersecting beams.

Instead, let us look closer at what happens when we switch the targets of two girls. In Figure 18.14, we see the before and after of such an example, as well as the two situations

interposed. If we consider the sum of the two lengths of the water beams before the swap $C + E + D + F$ versus the lengths after the swap $A + B$, we see that the latter must be less than the first. Indeed, we have $A < C + D$ and $B < E + F$ by the triangle inequality, which by summing the two inequalities gives the desired result.

As students of algorithmics, we can make the additional note that, if we construct the complete bipartite graph of the girls and boys with edges between them weighted with their distance, the bipartite matching with smallest total weight is a valid assignment. If this was not the case, we would be able to swap two targets and decrease the weight of the matching, contradicting the assumption that it was minimum-eight. This matching can be efficiently computed with min-cost max-flow. □

The last part shows another way to work with monovariants, by directly looking at the state that minimizes the value function. Since there's a transition to another state with a lower value if the state is inadmissible, it must be that the state is actually admissible (or else it wasn't of minimal value). This should remind you of the extreme value principle from Chapter 10 on greedy algorithms.

Problem 18.38.

Army Division	armydivision
Non-Negative Matrix Sums	nonnegmatrixsums

Additional Exercises

Problem 18.39.

Dejavu	dejavu
Digit Division	digitdivision
Dice Betting	dicebetting
Neighborhood Watch	neighborhoodwatch
Hamming Ellipses	hammingellipses
Perica	perica
Gwen's Gift	gwensgift
Permutation Descent Count	permutationdescent
The Sock Pile	thesockpile
Counting Heaps	countingheaps
Lucky Draw	luckydraw
Birthday Paradox	birthdayparadox
Genijalac	genijalac
Dance Reconstruction (hard)	dancehard
Code Permutations	codepermutations
Yule	yule
Cycles (Hard)	cycleshard
King's Colors	kingscolors

Boss Battle	bossbattle
Domino Tiling	dominotiling

Notes

While a long chapter, this was just a short foray into the huge area of combinatorics. For a wider survey of many combinatorial topics, we can recommend *A Course in Combinatorics* [54] by van Lint et al. An amazing collection of problems of a wide variety is László Lovász' *Combinatorial Problems and Exercises* [30] which teaches a large number of useful combinatorial techniques purely through posing problems.

To go deeper into the area of enumerative combinatorics, Stanley's two-volume series *Enumerative Combinatorics* [49] is a great resource. To focus more on the computational aspects, it is possible that Donald Knuth wrote everything there is to say in the relevant volume of *The Art of Computer Programming* [28].

If you at some point decide that computer science isn't for you, there are plenty of purely mathematical ways to count complex objects too. Aside from *Analytic Combinatorics* [16] mentioned earlier, Wilf's *generatingfunctionology* [58] is a good reference works on analyzing counting problems using analytical methods[8].

[8] If you thought the sudden appearance of e^{-1} in combinatorics was the coolest thing you've seen this millennia, wait until the trigonometry appears.

Chapter 19

Strings

In computing, text is one of the most prevalent data types that programs process. Everything from searching for files on your computer to quickly routing Internet packages to their right destination involves text processing in some sense. Therefore, it should not come as a surprise that a great effort has been made in computer science research to develop efficient algorithms and data structures for common text tasks.

For simplicity, we adopt the mathematical definition of a string in this chapter, i.e, a ***string*** is a sequence of ***characters*** chosen from a given, finite alphabet. When implementing string algorithms on a computer, we assume that any alphabet of size N is represented by the set of integers 0 through $N - 1$.

Conversely, sequences of bounded non-negative are strings. In particular, a single integer can be viewed as a string, since its representation in some base b is a string over the alphabet $\{0, 1, ..., b - 1\}$. In this chapter, we'll use this fact to apply string algorithms and data structures outside of traditional text processing use cases.

19.1 Tries

How does one sort strings? The straightforward method is to use your favorite comparison-based algorithm. Determining the time complexity of this is less straightfoward, since it depends on the length of your strings and how long their pairwise common prefixes are. For a simplified model, assume there are N strings, each of length L, all of which are almost equal so that any comparison takes $\Theta(L)$ time. Then this sorting takes $\Theta(NL \log N)$ time in the worst case.

If the total input size is N and there are The complexity of this is bounded by the maximum string length

We'll start by looking at three different ways of representing Representing a set of strings can be done in many ways. For example, a sorted list, as a sorted set using a balanced tree, or a hash map. The first two of these are

Representing a set of strings can be done in many ways. For example, a sorted list, as a sorted set using a balanced tree, or a hash map. The first two of these are

The ***trie*** (also called a ***prefix tree***) is the most common string data structure. It's used to store a set of words in the form of a rooted tree, where every prefix of a word is a vertex. Edges are added from one prefix P to all other prefixes Pc where c is a single character. If two words share a prefix, it only appears once as a vertex. The root of the tree is the

empty string, which all words share as a prefix. The trie is very useful when we want to associate some information with prefixes of strings and quickly get the information from neighboring strings.

Tries are normally implemented as trees, with the vertex for the prefix P containing a map *children* mapping a character c to the child prefixes Pc. Depending on the requirements of the problem that the trie is used for, these vertices may also be augmented with additional data. For example it's common for a prefix P to store a *count* of how many inserted strings were exactly equal to P. Inserting a word into a trie with this count can look like this:

```
1: procedure InsertWord(trie node T, string W, int idx)
2:     if idx = |W| then
3:         increase T.count by 1
4:         return
5:     child ← T.children[W[idx]]
6:     InsertWord(child, W, idx + 1)
```

This procedure is linear time in the length of the string W being inserted.

Many problems essentially can be solved by very simple usage of a trie, such as the following old IOI problem.

Type Printer

By Richard Peng. International Olympiad in Informatics 2008.

You need to print $N \leq 25\,000$ distinct words on a movable type printer, each with up to 20 letters `a-z`. Movable type printers are those old printers that require you to place small metal pieces (each containing a letter) in order to form words. A piece of paper is then pressed against them to print the word. The printer you have allows you to do any of the following operations:

- Add a letter to the end of the word currently in the printer.
- Remove the last letter from the end of the word currently in the printer. You are only allowed to do this if there is at least one letter currently in the printer.
- Print the word currently in the printer.

Initially, the printer is empty; it contains no metal pieces with letters. At the end of printing, you are allowed to leave some letters in the printer. Also, you are allowed to print the words in any order you like. As every operation requires time, you want to minimize the total number of operations. Find a sequence of operations that prints all the words using the minimum number of operations needed.

Solution. Let us start by solving a variation of the problem, where we aren't allowed to leave letters in the printer at the end. Are there any actions that never make sense? For example, what sequences of letters will ever appear in the type writer during an optimal sequence of operations? Clearly we should never have a sequence that is not a prefix of a word we wish to type. Conversely, every prefix of a word we wish to print must at some

point appear on the type writer, or there would be a word we could never print. Therefore, the partial words on the type printer are exactly the prefixes of the words to be printed, strongly hinting at a trie-based solution.

The edges of the trie containing all the input words has a nice property. The type writer can only add or remove a single letter at the end of the word, and this corresponds to moving along a single edge in the trie – to the parent when removing a letter, or to a child when adding one. The goal is then to construct the shortest possible tour starting at the root of the trie and passing through all the vertices. A tour of this kind must pass through each edge at least twice: the first time when visiting a vertex, and then again when moving back to the root. This is also what a depth-first search of a tree does, so the DFS tour of the trie represent an optimal sequence of operations.

The problem is subtly different once we are allowed to leave some letters in the printer at the end. The only difference for sequence of operations is that we are allowed to skip an arbitrary number of removals at the end of the sequence. If the last word printed is S, that difference is $|S|$ removals. An optimal solution should therefore print the longest word last, in order to skip as many removal operations as possible. The DFS traversal can be made to visit the longest word in the input last by always traversing into the subtree containing it last for each vertex.

The trie only needs to store whether a prefix represents the word or not. Additionally, we must perform a DFS on it.

```
1:  procedure DFS(T, longest, i)
2:      if T.count > 0 then
3:          print the current word
4:      for child c → T′ of T do
5:          if longest = nil or c ≠ longest[i] then          ▷ don't DFS into the non-longest word
6:              add the letter c to the printer
7:              DFS(T′, nil, −1)
8:              remove the last letter from the printer
9:      if longest ≠ nil then          ▷ if we're on the path to the longest word, DFS into it last
10:         c ← longest[i]
11:         add the letter c to the printer
12:         DFS(T.children[c], longest, i + 1)                                              □
```

Problem 19.1. *Bing it In* bing

Generally, the uses of tries are not this simple, where we only need to construct the trie and fetch the answer through a simple traversal. We often need to augment tries with additional information about the prefixes we insert. This is when tries start to become really powerful. The next problem is very difficult and requires several new useful techniques, including a commmon type of trie augmentation.

Klasika – klasika

By Ivan Paljak. Croatian Open Competition in Informatics 2019-2020, contest # 4.

In the beginning there was a vertex denoted as 1 and it represented the root of a tree. Your task is to support $Q \leq 200\,000$ queries of the form:

- Add x y – add a new vertex to the tree as a child of vertex x. The newly added vertex and vertex x are connected with an edge of weight $0 \leq y \leq 2^{30}$. Newly added vertices are denoted by 2, 3, 4, ... in the order that they are added.
- Query a b – find the highest *value* of a path starting at vertex a and ending in some vertex from the subtree of vertex b (which itself is considered to be in its own subtree). The value of a path is the exclusive or[a] (XOR) of the weights of all edges on the path.

[a] The exclusive or $\oplus$ for two binary digits is defined as $0 \oplus 0 = 1 \oplus 1 = 0$ and $0 \oplus 1 = 1 \oplus 0 = 1$. For integers a and b, $a \oplus b$ is computed by performing the operation for each pair of bits on the same position.

Solution. Our strategy for solving the problem will be to start with a much simpler version that we then build upon to the full problem. The easiest version that still makes sense would be where the entire graph is given beforehand, and queries are for paths from a to any vertex in the tree (i.e. $b = 1$).

To solve this easier problem, we need to know three basic facts about the XOR: it's associative, i.e. $a \oplus (b \oplus c) = (a \oplus b) \oplus c$, it's commutative, i.e. $a \oplus b = b \oplus a$ and every element is its own inverse, i.e. $a \oplus a = 0$. A consequence of this is that a path from vertex a to b can be reduced to two paths from the root to a and to b, in a very similar way to how LCA can be used to compute arbitrary distances in a tree.

Exercise 19.2. Let $V(a, b)$ be the value of the path from a to b. Prove that $V(a, b) = V(root, a) \oplus V(root, b)$.

The queries are then, given an integer x, find the vertex v that maximizes $x \oplus V(root, v)$ (where $x = V(root, a)$). This allowed us to remove the vertex a from the equation other than in the form of the value x. To remove the underlying graph completely from the problem, let S be the set of all $V(root, v)$, so that we're trying to find the maximum of $x \oplus y$ with $y \in S$. The set S can be computed with a single DFS from the root.

We've now simplified the problem enough to attack it directly. Since the XOR operation operates on a bit by bit basis, that's also how most XOR problems are solved. First, note that all values can have only the lowest 31 bits set. If possible we want $x \oplus y$ to have the highest possible bit (2^{30}) set, since that maximizes the value no matter what the lower bits are. This happens when exactly one of x and y has the 2^{30} bit as 1. The set of possibly y can thus be partitioned into two subsets: those where $x \oplus y$ has the bit set, A, and those where it doesn't, B. If A is non-empty, we know that the right y is in A, otherwise it's in B. In either case the top bit is now fixed, so we can repeat the procedure focusing only maximizing the top 30 bits instead, and repeat this until there's only a single y left. This is where the trie enters the problem. We have two types of operations. After fixing some

prefix p of y, we want to know how many numbers in S have that prefixes $p0$ and $p1$, as well as appending one bit at a time to that prefix. A trie efficiently supports both of those operations if we interpret the binary representation of each number as a string. For free, we've also gained the possibility of solving the problem where vertices are added since the trie can be updated at any time.

What remains is supporting queries with $b \neq 1$, i.e. where S consists of only values $V(root, v)$ where v is in a specific subtree. □

The problem required several separate steps such as, reducing the problem to only paths from the root, using a trie to maximize the XOR, pre-processing all the queries to perform some precomputation, and using the Euler tour to deal with queries on subtrees. Still, the problem could have been solved quickly in a contest by a seasoned competitor who would have seen several of these ideas in other problems. Hopefully, this drives home the point that solving many problems is the best way of practicing problem solving. Many problems become nothing more than the combination of things you have seen before.

19.2 String Matching

A common problem on strings – both in problem solving and real life – is that of searching. Not only do we need to check whether e.g. a set of strings contain some particular string, but also if one string contains another one as a substring. This operation is ubiquitous; operating systems allow us to search the contents of our files, and our text editors, web browsers and email clients all support substring searching in documents. It should come as no surprise that string matching is part of many string problems.

String Matching – stringmatching

Find all occurrences of the pattern P as a substring in the string W.

We can solve this problem naively in $O(|W| \cdot |P|)$. If we assume that an occurrence of P starts at position i in W, we can compare the substring $W[i...i + |P| - 1]$ to P in $O(|P|)$ time by looping through both strings, one character at a time:

```
1: procedure StringMatching(pattern P, string W)
2:     answer ← new vector
3:     for outer: i from 0 to |W| − |P| do
4:         for j from 0 to |P| − 1 do
5:             if P[j]! = W[i + j] then
6:                 start next iteration of outer
7:         answer.append(i)
8:     return answer
```

Intuitively, we should be able to do better. With the naive matching, our problem is basically that we can perform long stretches of partial matches for every position. Searching

for the string $a^{\frac{n}{2}}$ in the string a^n takes $O(n^2)$ time, since each of the $\frac{n}{2}$ positions where the pattern can appear requires us to look ahead for $\frac{n}{2}$ characters to realize we made a match. On the other hand, if we manage to find a long partial match of length l starting at i, we *know* what the next l letters of W are – they are the l first letters of P. With some cleverness, we should be able to exploit this fact, hopefully avoiding the need to scan them again when we attempt to find a match starting at $i + 1$.

For example, assume we have $P = bananarama$. Then, if we have performed a partial match of $banana$ at some position i in W but the next character is a mismatch (i.e., it is not an r), we know that no match can begin at the next 5 characters. Since we have matched $banana$ at i, we have that $W[i + 1...i + 5] = anana$, which does not contain a b.

As a more interesting example, take $P = abbaabborre$. This pattern has the property that the partial match of $abbaabb$ actually contains as a prefix of P itself as a suffix, namely abb. This means that if at some position i get this partial match but the next character is a mismatch, we can not immediately skip the next 6 characters. It is possible that the entire string could have been $abbaabbaabborre$. Then, an actual match (starting at the fifth character) overlaps our partial match. It seems that if we find a partial match of length 7 (i.e. $abbaabb$), we can only skip the first 4 characters of the partial match.

For every possible partial match of the pattern P, how many characters are we able to skip if we fail a k-length partial match? If we could precompute such a table, we should be able to perform matching in linear time, since we would only have to investigate every character of W once. Assume the next possible match is l letters forward. Then the new partial match must consist of the last $k - l$ letters of the partial match, i.e. $P[l \dots k - 1]$. But a partial match is just a prefix of P, so we must have $P[l \dots k - 1] = P[0 \dots l - 1]$. In other word, for every given k, we must find the longest suffix of $P[0 \dots K - 1]$ that is also a prefix of P (besides $P[0 \dots k - 1]$ itself, of course).

We can compute these suffixes rather easily in $O(n^2)$. For each possible position for the next possible match l, we perform a string matching to find all occurrences of prefixes of P within P:

```
1:  procedure LongestSuffixes(pattern P)
2:      T ← new int[|P| + 1]
3:      for l from 1 to |P| − 1 do
4:          matchLen ← 0
5:          while l + matchLen ≤ |W| do
6:              if P[l]! = P[matchLen] then
7:                  break
8:              matchLen ← matchLen + 1
9:              T[l + matchLen] = matchLen
10:     return T
```

A string such as $P = bananarama$, where no partial match could possibly contain a

new potential match, this table would simply be:

P	b	a	n	a	n	a	r	a	m	a
T	0	0	0	0	0	0	0	0	0	0

When $P = abbaabborre$, the table instead becomes:

P	a	b	b	a	a	b	b	o	r	r	e
T	0	0	0	1	1	2	3	0	0	0	0

With this precomputation, we can now perform matching in linear time. The matching is similar to the naive matching, except we can now use this precomputed table to determine whether there is a new possible match somewhere within the partial match.

```
1:  procedure StringMatching(pattern P, text W)
2:      matches ← new vector
3:      T ← LongestSuffixes(P)
4:      pos ← 0, match ← 0
5:      while pos + match < |W| do
6:          if match < |P| and W[pos + match] = P[match] then
7:              match ← match + 1
8:          else if match = 0 then
9:              pos ← pos + 1
10:         else
11:             pos ← pos + match − T[match]
12:             match ← T[match]
13:         if match = |P| then
14:             matches.append(match)
15:     return matches
```

In each iteration of the loop, we see that either *match* is increased by one, or *match* is decreased by $match - T[match]$ and *pos* is increased by the same amount. Since *match* is bounded by P and *pos* is bounded by $|W|$, this can happen at most $|W| + |P|$ times. Each iteration takes constant time, meaning our matching is $\Theta(|W| + |P|)$ time.

While this is certainly better than the naive string matching, it is not particularly helpful when $|P| = \Theta(|W|)$ since we need an $O(|P|)$ preprocessing. The solution lies in how we computed the table of suffix matches, or rather, the fact that it is entirely based on string matching itself. We just learned how to use this table to perform string matching in linear time. Maybe we can use this table to extend itself and get the precomputation down to $O(|P|)$? After all, we are looking for occurrences of prefixes of P in P itself, which is exactly what string matching does. If we modify the string matching algorithm for this purpose, we get what we need:

```
1:  procedure LongestSuffixes(pattern P)
2:      T ← new int[|P| + 1]
```

```
3:    pos ← 1, match ← 0
4:    while pos + match < |P| do
5:        if P[pos + match] = P[match] then
6:            T[pos + match] ← match + 1
7:            match ← match + 1
8:        else if match = 0 then
9:            pos ← pos + 1
10:       else
11:           pos ← pos + match − T[match]
12:           match ← T[match]
13:       if match = |P| then
14:           matches.append(match)
15:   return T
```

This string matching algorithm is called the ***Knuth-Morris-Pratt*** (KMP) algorithm.

Using the same analysis as for the improved string matching, this precomputation is instead $\Theta(|P|)$. The resulting string matching then takes $\Theta(|P| + |W|)$.

Competitive Tip

Most programming languages have functions to find occurrences of a certain string in another. However, they mostly use the naive $O(|W||P|)$ procedure. Be aware of this and code your own string matching if you need it to perform in linear time.

Clock Pictures

Nordic Collegiate Programming Contest 2014

You have two pictures of an unusual kind of clock. The clock has $2 \leq n \leq 200\,000$ hands, each having the same length and no kind of marking whatsoever. Also, the numbers on the clock are so faded that you can't even tell anymore what direction is up in the picture. So the only thing that you see on the pictures, are n shades of the n hands, and nothing else.

You'd like to know if both images might have been taken at exactly the same time of the day, possibly with the camera rotated at different angles.

Given the description of the two images, determine whether it is possible that these two pictures could be showing the same clock displaying the same time. An description of an image consists of a set of n angles, given in thousandths of a degree.

19.3 Hashing

Hashing is a concept most familiar from the hash table data structure. The idea behind the structure is to compress a set S of elements from a large set to a smaller set, in order to quickly determine memberships of S by having a direct indexing of the smaller set into an array (which has $\Theta(1)$ look-ups). In this section, we are going to look at hashing in a

different light, as a way of speeding up comparisons of data. When comparing two pieces of data a and b of size n for equality, we need to use $\Theta(n)$ time in the worst case since every bit of data must be compared. This is fine if we perform only a single comparison. If we instead wish to compare many pieces of data, this becomes an unnecessary bottleneck. We can use the same kind of hashing as with hash tables, by defining a "random" function $H(x) : S \to \mathbb{Z}_n$ such that $x \neq y$ implies $H(x) \neq H(y)$ with high probability. Such a function allows us to perform comparisons in $\Theta(1)$ time (with linear preprocessing), by reducing the comparison of arbitrary data to small integers (we often choose n to be on the order of 2^{32} or 2^{64} to get constant-time comparisons). The trade-off lies in correctness, which is compromised in the unfortunate event that we perform a comparison $H(x) = H(y)$ even though $x \neq y$.

FriendBook

By Arash Rouhani. Swedish Olympiad in Informatics 2011, Finals.

FriendBook is a web site where you can chat with your friends. For a long time, they have used a simple "friend system" where each user has a list of which other users are their "friends". Recently, a somewhat controversial feature was added, namely a list of your "enemies". While the friend relation will always be mutual (two users must confirm that they wish to be friends), enmity is sometimes one-way – a person A can have an enemy B, who – by plain animosity – refuse to accept A as an enemy.

Being a poet, you have lately been pondering the following quote.

> A friend is someone who dislike the same people as yourself.

Given a FriendBook network with $2 \leq N \leq 5000$ friends, you wonder to what extent this quote applies. More specifically, for how many pairs of users is it the case that they are either friends with identical enemy lists, or are not friends and does not have identical enemy lists?

Input

The network is described using N lines, each with N characters. N lines follow, each containing n characters. The c'th character on the r'th line $S_{r,c}$ species what relation person r has to person c. This character is either

`V` – in case they are friends.

`F` – if r thinks of c as an enemy.

`.` – r has a neutral attitude towards c.

S_{ii} is always `.`, and S_{ij} is `V` if and only if S_{ji} is `V`.

This problem lends itself very well to hashing. It is clear that the problem is about comparisons – indeed, we are to count the number of pairs of persons who are either friends and have equal enemy lists or are not friends and have unequal enemy lists. The first step is to extract the enemy lists E_i for each person i. This will be a N-length string, where the j'th character is `F` if person j is an enemy of person i, and `.` otherwise. Basically, we remove all the friendships from the input matrix. Performing naive comparisons on

these strings would only give us a $O(N^3)$ time bound, since we need to perform N^2 comparisons of enemy lists of length N bounded only by $O(N)$ in the worst case. Here, hashing comes to our aid. By instead computing $h_i = H(E_i)$ for every i, comparisons of enemy lists instead become comparisons of the integers h_i – a $\Theta(1)$ operation – thereby reducing the complexity to $\Theta(N^2)$.

Alternative solutions exist. For example, we could instead have sorted all the enemy lists, after which we can perform a partitioning of the lists by equality in $\Theta(N^2)$ time. However, this takes $O(N^2 \log N)$ time with naive sorting (or $O(N^2)$ if radix sort is used, but it is more complex) and is definitely more complicated to code than the hashing approach. Another option is to insert all the strings into a trie, simplifying this partitioning and avoiding the sorting altogether. This is better, but still more complex. While it would have the same complexity, the constant factor would be significantly worse compared to the hashing approach.

This is a common theme among string problems. While most string problems can be solved without hashes, solutions using them tend to be simpler.

The true power of string hashing is not this basic preprocessing step where we can only compare two strings. Another hashing technique allows us to compare arbitrary substring of a string in constant time.

Definition 19.1 — Polynomial Hash
Let $S = s_1 s_2 \dots s_n$ be a string. The *polynomial hash* $H(S)$ of S is the number

$$H(S) = (s_1 p^{n-1} + s_2 p^{n-2} + \cdots + s_{n-1} p + s_n) \bmod M$$

As usual when dealing with strings in arithmetic expressions, we take s_i to be some numeric representation of the character, like its ASCII encoding. In C++, `char` is actually a numeric type and is thus usable as a number when using polynomial hashes.

Polynomial hashes have many useful properties.

Theorem 19.1 — Properties of the Polynomial Hash
If $S = s_1 \dots s_n$ is a string and c is a single character, we have that

1. $H(S||c) = (pH(S) + H(c)) \bmod M$
2. $H(c||S) = (H(S) + H(c)p^n) \bmod M$
3. $H(s_2 \dots s_n) = (H(S) - H(s_1)p^{n-1}) \bmod M$
4. $H(s_1 \dots s_{n-1}) = (H(S) - H(s_n))p^{-1} \bmod M$
5. $H(s_l s_{l+1} \dots s_{r-2} s_{r-1}) = (H(s_1 \dots s_{R-1}) - H(s_1 - S_{L-1})p^{R-L}) \bmod M$

Exercise 19.3. Prove the properties of Theorem 19.1

Exercise 19.4. How can we compute the hash of $S||T$ in $O(1)$ given the hashes of the strings

S and T?

Properties 1-4 alone allow us to append and remove characters from the beginning and end of a hash in constant time. We refer to this property as polynomial hashes being *rolling*. This property allows us to String Matching problem with a single pattern (Section 19.2) with the same complexity as KMP, by computing the hash of the pattern P and then rolling a $|P|$-length hash through the string we are searching in. This algorithm is called the ***Rabin-Karp*** algorithm.

Property 5 allows us to compute the hash of any substring of a string in constant time, provided we have computed the hashes $H(S_1), H(s_1s_2), \ldots, H(s_1s_2 \ldots s_n)$ first. Naively this computation would be $\Theta(n^2)$, but property 1 allows us to compute them recursively, resulting in $\Theta(n)$ precomputation.

Radio Transmission

Baltic Olympiad in Informatics 2009

Given is a string S of length at most 10^6. Find the shortest string L, such that S is a substring of the infinite string $T = \ldots LLLLL \ldots$. If there are several possible L of the shortest length, output any of them.

Assume that L has a particular length l. Then, since T is periodic with length l, S must be too (since it is a substring of T). Conversely, if S is periodic with some length l, can can choose as $L = s_1s_2 \ldots s_l$. Thus, we are actually seeking the smallest l such that S is periodic with length l. The constraints this puts on S are simple. We must have that

$$s_1 = s_{l+1} = s_{2l+1} = \ldots$$

$$s_2 = s_{l+2} = s_{2l+2} = \ldots$$

$$\ldots$$

$$s_l = s_{2l} = s_{3l} = \ldots$$

Using this insight as-is gives us a $O(|S|^2)$ algorithm, where we first fix l and then verify if those constraints hold. The idea is sound, but a bit slow. Again, the problematic step is that we need to perform many slow, linear-time comparisons. If we look at what comparisons we actually perform, we are actually comparing two substrings of S with each other:

$$s_1s_2 \ldots s_{n-l+1} = s_{l+1}s_{l+2} \ldots s_n$$

Thus we are actually performing a linear number of substring comparisons, which we now know are actually constant-time operations after linear preprocessing. Hashes thus gave us a $\Theta(N)$ algorithm.

Radio Transmission

```
H lh = 0, Rh = 0;
int l = 0;
for (int i = 1; i <= n; ++i) {
  Lh = (Lh * p + S[i]) % M;
  Rh = (S[n - i + 1] * p^(i - 1) + Rh) % M;
  if (Lh == Rh) {
    l = i;
  }
}
cout << n - l << endl;
```

■

Polynomial hashes are also a powerful tool to compare something against a large number of strings using hash sets. For example, we could actually use hashing as a replacement for Aho-Corasick. However, we would have to perform one pass of rolling hash for each different pattern length. If the string we are searching in is N and the sum of pattern lengths are P, this is not $O(N+P)$ however. If we have k different pattern lengths, their sum must be at least $1+2+\cdots+k=\Theta(k^2)$, so $k=O(\sqrt{P})$.

Substring Range Matching

Petrozavodsk Winter Training Camp 2015

Given $N \leq 50\,000$ strings $s_1, s_2, \ldots, s_N$ and a list of $Q \leq 100\,000$ queries of the form L, R, S, answer for each such query the number of strings in $s_L, s_{L+1}, \ldots, s_R$ which contain S as a substring.

The sum of $|S|$ over all queries is at most 20 000. The lengths $|s_1| + |s_2| + \cdots + |s_N|$ is at most 50 000.

Let us focus on how to solve the problem where every query has the same string S. In this case, we would first find which of the strings s_i that S is contained in using polynomial hashing. To respond to a query, could for example keep a set of all the i where s_i was an occurrence together with how many smaller s_i contained the string (i.e. some kind of partial sum). This would allow us to respond to a query where $L = 1$ using a upper bound in our set. Solving queries of the form $[1, R]$ is equivalent to general intervals however, since the interval $[L, R]$ is simply the interval $[1, R]$ with the interval $[1, L-1]$ removed. This procedure would take $\Theta(\sum |s_i|)$ time to find the occurrences of S, and $O(Q \log N)$ time to answer the queries.

When extending this to the general case where our queries may contain different S, we do the same thing but instead find the occurrences of all the patterns of the same length p simultaneously. This can be done by keeping the hashes of those patterns in a map, to allow for fast look-up of our rolling hash. Since there can only be at most $\sqrt{20\,000} \approx 140$ different pattern lengths, we must perform about $140 \cdot 50\,000 \approx 7\,000\,000$ set look-ups, which is feasible.

Substring Range Matching

```
int countInterval(int upTo, const set<pii>& s) {
  auto it = s.lower_bound(pii(upTo + 1, 0));
  if (it == s.begin()) return 0;
  return (--it)->second;
}

int main() {
  int N, Q;
  cin >> N >> Q;
  vector<string> s(N);
  rep(i,0,N) cin >> s[i];

  map<int, set<string>> patterns;

  vector<tuple<int, int, string>> queries;
  rep(i,0,Q) {
    int L, R;
    string S;
    cin >> L >> R >> S;
    queries.emplace_back(L, R, S);
    patterns[sz(s)].insert(S);
  }

  map<H, set<pii>> hits;
  trav(pat, patterns) {
    rep(i,0,N) {
      vector<H> hashes = rollHash(s[i], pat.first);
      trav(h, hashes)
        if (pat.second.count(h))
          hits[h].emplace(i, sz(hits[h]) + 1);
    }
  }

  trav(query, queries) {
    H h = polyHash(get<2>(query));
    cout << countInterval(R, hits[h]) - countInterval(L-1, hits[h]) << endl;
  }
}
```

■

Exercise 19.5. Hashing can be used to determine which of two substrings are the lexicographically smallest one. How? Extend this result to a simple $\Theta(n \log S + S)$ construction of a suffix array, where n is the number of strings and S is the length of the string.

Problem 19.6.
Palindrome Substring palindromesubstring

The Parameters of Polynomial Hashes

Until now, we have glossed over the choice of M and p in our polynomial hashing. These choices happen to be important. First of all, we want M and p to be relatively prime. This

ensures p has an inverse modulo M, which we use when erasing characters from the end of a hash. Additionally, $p^i \bmod M$ have a smaller period when p and M share a factor.

We wish M to be sufficiently large, to avoid hash collisions. If we compare the hashes of c strings, we want $M = \Omega(\sqrt{c})$ to get a reasonable chance at avoiding collisions. However, this depends on how we use hashing. p must be somewhat large as well. If p is smaller than the alphabet, we get trivial collisions such as $H(10) = H(p)$.

Whenever we perform rolling hashes, we must have $(M-1)p < 2^{64}$ if we use 64-bit unsigned integers to implement hashes. Otherwise, the addition of a character would overflow. If we perform substring hashes, we instead need that $(M-1)^2 < 2^{64}$, since we perform multiplication of a hash and an arbitrary power of p. When using 32-bit or 128-bit hashes, these limits change correspondingly. Note that the choice of hash size depends on how large an M we can choose, which affect collision rates.

One might be tempted to choose $M = 2^{64}$ and use the overflow of 64-bit integers as a cheap way of using hashes modulo 2^{64}. This is a bad idea, since it is possible to construct strings which are highly prone to collisions.

Definition 19.2 — Thue-Morse Sequence
Let the binary sequence τ_i be defined as

$$\tau_i = \begin{cases} 0 & \text{if } i = 0 \\ \tau_{i-1}\overline{\tau_{i-1}} & \text{if } i > 0 \end{cases}$$

The *Thue-Morse* sequence is the infinite sequence τ_i as $i \to \infty$.

This sequence is well-defined since τ_i is a prefix of τ_{i-1}, meaning each recursive step only append a string to the sequence. It starts 0, 01, 0110, 01101001, 0110100110010110.

Exercise 19.7. Prove that τ_{2i} is a palindrome.

Theorem 19.2
For a polynomial hash H with an odd p, $2^{\frac{n(n+1)}{2}} \mid H(\overline{\tau_n}) - H(\tau_n)$.

Proof. We will prove this by induction on n. For $n = 0$, we have $1 \mid H(\overline{\tau_n}) - H(\tau_n)$ which is vacuously true.

In our inductive step, we have that

$$H(\tau_n) = H(\tau_{n-1}\|\overline{\tau_{n-1}}) = p^{2^{n-1}} \cdot H(\overline{\tau_{n-1}}) + H(\tau_{n-1})$$

and

$$H(\overline{\tau_n}) = H(\overline{\tau_{n-1}}\|\tau_{n-1}) = p^{2^{n-1}} \cdot H(\tau_{n-1}) + H(\overline{\tau_{n-1}})$$

Then,

$$\begin{aligned} H(\overline{\tau_n}) - H(\tau_n) &= p^{2^{n-1}}(H(\tau_{n-1}) - H(\overline{\tau_{n-1}})) + (H(\overline{\tau_{n-1}}) - H(\tau_{n-1})) \\ &= (p^{2^{n-1}} - 1)(H(\tau_{n-1}) - H(\overline{\tau_{n-1}})) \end{aligned}$$

Note that $p^{2^{n-1}} - 1 = (p^{2^{n-2}} - 1)(p^{2^{n-2}} + 1)$ If p is odd, the second factor is divisible by 2. By expanding $p^{2^{n-2}}$, we can prove that $p^{2^{n-1}}$ is divisible by 2^n.

Using our induction assumption, we have that

$$2^n \cdot 2^{\frac{(n-1)n}{2}} \mid (p^{2^{n-1}} - 1)(H(\tau_{n-1}) - H(\overline{\tau_{n-1}}))$$

But $2^n \cdot 2^{\frac{(n-1)n}{2}} = 2^{\frac{n(n+1)}{2}}$, proving our statement. □

This means that we can construct a string of length linear in the bit size of M that causes hash collisions if we choose M as a power of 2, explaining why it is a bad choice.

2D Polynomial Hashing

Polynomial hashing can also be applied to pattern matching in grids, by first performing polynomial hashing on all rows of the grid (thus reducing the grid to a sequence) and then on the columns.

Surveillance

By Aron Granberg and Johan Sannemo. Swedish Olympiad in Informatics 2016, IOI Qualifiers

Given a matrix of integers $A = (a_{r,c})$ find all occurrences of another matrix $P = (p_{r,c})$ in A which may differ by a constant C. An occurrence (i, j) means that $a_{i+r,j+c} = p_{r,c} + C$ where C is a constant.

If we assume that $C = 0$, the problem is reduced to simple 2D pattern matching, which is easily solved by hashing. The requirement that such a pattern should be invariant to addition by a constant is a bit more complicated.

How would we solve this problem in one dimension, i.e. when $r = 1$? In this case, we have that a match on column j would imply

$$a_{1,j} - p_{1,1} = c$$

$$\dots$$

$$a_{1,j+n-1} - p_{1,n} = c$$

Since c is arbitrary, this means the only condition is that

$$a_{1,j} - p_{1,1} = \dots = a_{1,j+n-1} - p_{1,n} = c$$

Rearranging this gives us that

$$a_{1,j} - a_{1,j+1} = p_{1,1} - p_{1,2}$$

$$a_{1,j+1} - a_{1,j+2} = p_{1,2} - p_{1,3}$$

$$\dots$$

By computing these two sequences of the adjacent differences of elements $a_{1,i}$ and $r_{1,j}$, we have reduced the problem to substring matching and can apply hashing. In 2D, we can do something similar. For a match (i, j), it is sufficient that this property holds for every line and every column in the match. We can then find matches using two 2D hashes.

Problem 19.8.

Chasing Subs	chasingsubs

Additional Exercises

Problem 19.9.

Baza	baza
Prefix Free Code	prefixfreecode
Just a Quiz	justaquiz

Notes

rabin karp paper

- KMP
- hashing

Appendices

Chapter A

Competition Strategy

Competitive programming is what we call the mind sport of solving algorithmical problems and coding their solutions, often under the pressure of time. Most programming competitions are performed online, at your own computer through some kind of online judge system. For students of either high school or university, there are two main competitions. High school students compete in the *International Olympiad in Informatics* (IOI), and university students go for the *International Collegiate Programming Contest* (ICPC).

Different competition styles have different difficulty, problem types and strategies. In this chapter, we will discuss some basic strategy of programming competitions, and give tips on how to improve your competitive skills.

A.1 IOI

The IOI is an international event where a large number of countries send teams of up to 4 high school students to compete individually against each other during two days of competition. Every participating country has its own national selection olympiad first.

During a standard IOI contest, contestants are given 5 hours to solve 3 problems, each worth at most 100 points. These problems are not given in any particular order, and the scores of the other contestants are hidden until the end of the contest. Generally none of the problems are "easy" in the sense that it is immediately obvious how to solve the problem in the same way the first 1-2 problems of most other competitions are. This poses a large problem, in particular for the amateur. Without any trivial problems nor guidance from other contestants on what problems to focus on, how does an IOI competitor prioritize? The problem is further exacerbated by problems not having a simple binary scoring, with a submission being either accepted or rejected. Instead, IOI problems contain many so-called *subtasks*. These subtasks give partial credit for the problem, and contain additional restrictions and limits on either input or output. Some problems do not even use discrete subtasks. In these tasks, scoring is done on some scale which determines how "good" the output produced by your program is.

Strategy

Very few contestants manage to solve every problem fully during an IOI contest. You are most likely not one of them, which leaves you with two options – you either skip a problem entirely, or you solve some of its subtasks. At the start of the competition, you should

read through every problem and *all of the subtasks*. In the IOI you do not get extra points for submitting faster. Thus, it does not matter if you read the problems at the beginning instead of rushing to solve the first problem you read. Once you have read all the subtasks, you will often see the solutions to some of the subtasks immediately. Take note of the subtasks which you know how to solve!

Deciding on which order you should solve subtasks in is probably one of the most difficult parts of the IOI for contestants at or below the silver medal level. In IOI 2016, the difference between receiving a gold medal and a silver medal was a mere 3 points. On one of the problems, with subtasks worth 11, 23, 30 and 36 points, the first silver medalist solved the third subtask, worth 30 points (a submission that possibly was a failed attempt at 100 points). Most competitors instead solved the first two subtasks, together worth 34 points. If the contestant had solved the first two subtasks instead, he would have gotten a gold medal.

The problem basically boils down to the question *when should I solve subtasks instead of focusing on a 100 point solution?* There is no easy answer to this question, due to the lack of information about the other contestants' performances. First of all, you need to get a good sense of how difficult a solution will be to implement correctly before you attempt it. If you only have 30 minutes left of a competition, it might not be a great idea to go for a 100 point solution on a very tricky problem. Instead, you might want to focus on some of the easier subtasks you have left on this or other problems. If you fail your 100 point solution which took over an hour to code, it is nice to know you did not have some easy subtasks worth 30-60 points which could have given you a medal.

Problems without discrete scoring (often called *heuristic* problems) are almost always the hardest ones to get a full score on. These problems tend to be very fun, and some contestants often spend way too much time on these problems. They are treacherous in that it is often easy to increase your score by *something*. However, those 30 minutes you spent to gain one additional point may have been better spent coding a 15 point subtask on another problem. As a general rule, go for the heuristic problem last during a competition. This does not mean to skip the problem unless you completely solve the other two, just to focus on them until you decide that the heuristic problem is worth more points given the remaining time.

In IOI, you are allowed to submit solution attempts a large number of times, without any penalty. Use this opportunity! When submitting a solution, you will generally be told the results of your submission on each of the secret test cases. This provides you with much details. For example, you can get a sense of how correct or wrong your algorithm is. If you only fail 1-2 cases, you probably just have a minor bug, but your algorithm in general is probably correct. You can also see if your algorithm is fast enough, since you will be told the execution time of your program on the test cases. Whenever you make a change to your code which you think affect correctness or speed – submit it again! This gives you a sense of your progress, and also works as a good regression test. If your change

introduced more problems, you will know.

Whenever your solution should pass a subtask, submit it. These subtask results will help you catch bugs earlier when you have less code to debug.

Getting Better

The IOI usually tend to have pretty hard problems. Some areas get rather little attention. For example, there are basically no pure implementation tasks and very little geometry.

First and foremost, make sure you are familiar with all the content in the IOI syllabus[1]. This is an official document which details what areas are allowed in IOI tasks. This book deals with most, if not all of the topics in the IOI syllabus.

In the Swedish IOI team, most of the top performers tend to also be good mathematical problem solvers (also getting IMO medals). Combinatorial problems from mathematical competitions tend to be somewhat similar to the algorithmic frame of mind, and can be good practice for the difficult IOI problems.

When selecting problems to practice on, there are a large number of national olympiads with great problems. The Croatian Open Competition in Informatics[2] is a good source. Their competitions are generally a bit easier than solving IOI with full marks, but are good practice. Additionally, they have a final round (the Croatian Olympiad in Informatics) which are of high quality and difficulty. COCI publishes solutions for all of their contests. These solutions help a lot in training.

One step up in difficulty from COCI is the Polish Olympiad in Informatics[3]. This is one of the most difficult European national olympiad published in English, but unfortunately they do not publish solutions in English for their competitions.

There are also many regional olympiads, such as the Baltic, Balkan, Central European and the Asia-Pacific Olympiads in Informatics. Their difficulty is often higher than that of national olympiads, and of the same format as an IOI contest (3 problems, 5 hours). These, and old IOI problems, are probably the best sources of practice after having mastered the basics of competitive programming.

A.2 ICPC

In ICPC, you compete in teams of three to solve about 10-12 problems during 5 hours. A twist in in the ICPC-style competitions is that the team shares a single computer. This makes it a bit harder to prioritize tasks in ICPC competitions than in IOI competitions. You will often have multiple problems ready to be coded, and wait for the computer. In ICPC, you see the progress of every other team as well, which gives you some suggestions on what to solve. As a beginner or medium-level team, this means you will generally have

[1] https://heap.link/ioi-syllabus

[2] https://heap.link/judge:coci

[3] https://heap.link/judge:poi

a good idea on what to solve next, since many better teams will have prioritized tasks correctly for you.

ICPC scoring is based on two factors. First, teams are ranked by the number of solved problems. As a tie breaker, the penalty time of the teams are used. The penalty time of a single problem is the number of minutes into the contest when your first successful attempt was submitted, plus a 20 minute penalty for any rejected attempts. Your total penalty time is the sum of penalties for every problem.

Strategy

In general, teams will be subject to the penalty tie-breaking. In the 2016 ICPC World Finals, both the winners and the team in second place solved 11 problems. Their penalty time differed by a mere 7 minutes! While such a small penalty difference in the very top is rather unusual, it shows the importance of taking your penalty into account.

Minimizing penalties generally comes down to a few basic strategic points:

- Solving the problems in the right order.
- Solving each problem quickly.
- Minimizing the number of rejected attempts.

In the very beginning of an ICPC contest, the first few problems will be solved quickly. In 2016, the first accepted submissions to five of the problems came in after 11, 15, 18, 32, 44 minutes. On the other hand, after 44 minutes no team had solved all of those problems. Why does not every team solve the problems in the same order? Teams are of different skill in different areas, make different judgment calls regarding difficulty or (especially early in the contest) simply read the problem in a different order. The better you get, the harder it is to distinguish between the "easy" problems of a contest – they are all "trivial" and will take less than 10-15 minutes to solve and code.

Unless you are a very good team or have very significant variations in skill among different areas (e.g., you are graph theory experts but do not know how to compute the area of a triangle), you should probably follow the order the other teams choose in solving the problems. In this case, you will generally always be a few problems behind the top teams.

The better you get, the harder it is to exploit the scoreboard. You will more often be tied in the top with teams who have solved the exact same problems. The problems that teams above you have solved but you have not may only be solved by 1-2 teams, which is not a particularly significant indicator in terms of difficulty. Teams who are very strong at math might prioritize a hard maths problem before an easier (on average for most teams) dynamic programming problem. This can risk confusing you into solving the wrong problems for the particular situation of your team.

The amount of cooperation during a contest is difficult to decide upon. The optimal amount varies a lot between different teams. In general, the amount of cooperation should

increase within a single contest from the start to the end. In the beginning, you should work in parallel as much as possible, to quickly read all the problems, pick out the easy-medium problems and start solving them. Once you have competed in a few contests, you will generally know the approximate difficulty of the simplest tasks, so you can skim the problem set for problems of this difficulty. Sometimes, you find an even easier problem in the beginning than the one the team decided to start coding.

If you run out of problems to code, you waste computer time. Generally, this should not happen. If it does, you need to become faster at solving problems.

Towards the end of the contest, it is a common mistake to parallelize on several of the hard problems at the same time. This carries a risk of not solving any of the problems in the end, due to none of the problems getting sufficient attention. Just as with subtasks in IOI, this is the hardest part of prioritizing tasks. During the last hour of an ICPC contest, the previously public scoreboard becomes frozen. You can still see the number of attempts other teams make, but not whether they were successful. Hence, you can not really know how many problems you have to solve to get the position that you want. Learning your own limits and practicing a lot as a team – especially on difficult contests – will help you get a feeling for how likely you are to get in all of your problems if you parallelize.

Read all the problems! You do not want to be in a situation where you run out of time during a competition, just to discover there was some easy problem you knew how to solve but never read the statement of. ICPC contests are made more complex by the fact that you are three different persons, with different skills and knowledge. Just because you can not solve a problem does not mean your team mates will not find the problem trivial, have seen something similar before or are just better at solving this kind of problem.

The scoreboard also displays failed attempts. If you see a problem where many teams require extra attempts, be more careful in your coding. Maybe you can perform some extra tests before submitting, or make a final read-through of the problem and solution to make sure you did not miss any details.

If you get Wrong Answer, you may want to spend a few minutes to code up your own test case generators. Prefer generators which create cases where you already know the answers. Learning e.g. Python for this helps, since it usually takes under a minute to code a reasonably complex input generator.

If you get Time Limit Exceeded, or even suspect time might be an issue – code a test case generator. Losing a minute on testing your program on the worst case, versus a risk of losing 20 minutes to penalty is be a trade-off worth considering on some problems.

You are allowed to ask questions to the judges about ambiguities in the problems. Do this the moment you think something is ambiguous (judges generally take a few valuable minutes in answering). Most of the time they give you a "No comment" response, in which case the perceived ambiguity probably was not one.

If neither you nor your team mates can find a bug in a rejected solution, consider coding it again from scratch. Often, this can be done rather quickly when you have already

coded a solution.

Getting Better

- Practice a lot with your team. Having a good team dynamic and learning what problems the other team members excel at can be the difference that helps you to solve an extra problem during a contest.
- Learn to debug on paper. Wasting computer time for debugging means not writing code! Whenever you submit a problem, print the code. This can save you a few minutes in getting your print-outs when judging is slow (in case your submission will need debugging). If your attempt was rejected, you can study your code on paper to find bugs. If you fail on the sample test cases and it takes more than a few minutes to fix, add a few lines of debug output and print it as well (or display it on half the computer screen).
- Learn to write code on paper while waiting for the computer. In particular, tricky subroutines and formulas are great to hammer out on paper before occupying valuable computer time.
- Focus your practice on your weak areas. If you write buggy code, learn your programming language better and code many complex solutions. If your team is bad at geometry, practice geometry problems. If you get stressed during contests, make sure you practice under time pressure. For example, Codeforces [4] has an excellent gym feature, where you can compete retroactively in a contest using the same amount of time as in the original contest. The scoreboard will then show the corresponding scoreboard from the original contest during any given time.

[4] https://heap.link/judge:codeforces

Chapter B

Mathematical Notation

This book is intended to be mostly readable by a mathematically strong high school student. There some certain knowledge about e.g. pre-calculus (trigonometry, polynomials) is assumed, but not much else. In most places where more math is required it's described from the ground up. This appendix briefly reviews mathematical notation that's used but might not yet have been encountered.

B.1 Sets

A ***set*** is an unordered collection of **distinct** objects, such as numbers, letters, other sets, and so on. The objects contained within a set are called its ***elements***, or ***members***. Sets are written as a comma-separated list of its elements, enclosed by curly brackets:

$$A = \{2, 3, 5, 7\}.$$

In this example, A contains four elements: the integers 2, 3, 5 and 7.

Because a set is unordered and only contains distinct objects, the set $\{1, 2, 2, 3\}$ is the exact same set as $\{3, 2, 1, 1\}$ and $\{1, 2, 3\}$.

If x is an element in a set S, we write that $x \in S$. For example, we have that $2 \in A$ (referring to our example A above). Conversely, we use the notation $x \notin S$ when the opposite holds. We have e.g., that $11 \notin A$.

Another way of describing the elements of a set uses the ***set builder*** notation, in which a set is constructed by explaining what properties its elements should have. The general syntax is

$$\{\text{element} \mid \text{properties that the element must have}\}.$$

To construct the set of all even integers, we use the syntax

$$\{2i \mid i \text{ is an integer}\}$$

which is read as "the set containing all numbers of the form $2i$ where i is an integer. To construct the set of all primes, we write

$$\{p \mid p \text{ is prime}\}.$$

Some sets are used often enough to be assigned their own symbols:

- $\mathbb{Z}$ – the set of integers $\{\ldots, -2, -1, 0, 1, 2, \ldots\}$,
- $\mathbb{Z}_+$ – the set of *positive* integers $\{1, 2, 3, \ldots\}$,
- $\mathbb{N}$ – the set of *non-negative* integers $\{0, 1, 2, \ldots\}$,
- $\mathbb{Q}$ – the set of all rational numbers $\{\frac{p}{q} \mid p, q \text{ integers where } q \neq 0\}$,
- $\mathbb{R}$ – the set of all real numbers,
- $[n]$ – the set of the first n positive integers $\{1, 2, \ldots, n\}$, and
- $\varnothing$ – the empty set.

A set A is a ***subset*** of a set S if, for every $x \in A$, we also have $x \in S$ (i.e., every member of A is a member of S). We denote this with $A \subseteq S$. For example

$$\{2, 3\} \subseteq \{2, 3, 5, 7\}$$

and

$$\left\{\frac{2}{4}, 2, \frac{-1}{7}\right\} \subseteq \mathbb{Q}.$$

For any set S, we have that $\varnothing \subseteq S$ and $S \subseteq S$. Whenever a set A is not a subset of another set B, we write that $A \nsubseteq B$. For example,

$$\{2, \pi\} \nsubseteq \mathbb{Q}$$

since π is not a rational number.

We say that sets A and B are ***equal*** whenever $x \in A$ if and only if $x \in B$. This is equivalent to $A \subseteq B$ and $B \subseteq A$. Sometimes, we will use the latter condition when proving set equality, i.e., first proving that every element of A must also be an element of B and then the other way round.

Sets have many useful operations defined on them. The ***intersection*** $A \cap B$ of two sets A and B is the set containing all the elements which are members of **both** sets. If the intersection of two sets is the empty set, we call the sets ***disjoint***. A similar concept is the ***union*** $A \cup B$ of A and B, defined as the set containing the elements which are members of **either** set. The ***set difference*** $A \setminus B$ is defined as those elements present in A but not in B. Finally, there's a special set difference called the ***complement***. It's sometimes clear from context that we're only working with subsets of some specific "universe set" U. The complement set A^c is then defined to be the difference between U and A.

For example, if

$$X = \{1, 2, 3, 4\}, Y = \{4, 5, 6, 7\}, Z = \{1, 2, 6, 7\}$$

then

$$X \cap Y = \{4\}$$

$$X \cap Y \cap Z = \varnothing$$

$$X \cup Z = \{1, 2, 3, 4, 6, 7\}$$

$$Y \setminus Z = \{4, 5\}.$$

For two numbers a and b, we define the ***interval*** $[a, b]$ to be the set $\{c \mid a \leq c \leq b\}$. If $b < a$, this is the empty set. An interval that includes both of its endpoint is called ***closed***. If an interval does not include an endpoint, it is written as $[a, b)$, $(a, b]$ or (a, b) if it does not include the right endpoint, left endpoint or neither endpoint, respectively (note that $[a, a)$ is also the empty set). The first two are called ***half-open*** intervals and the last a ***open*** interval.

B.2 Functions

A ***function*** f is a rule that maps values x to another value called the ***image*** of x, denoted $f(x)$. The set of values that f maps is called its ***domain***, while the set of values it can map values to is called its ***codomain***. If a function f has domain X and codomain Y, we write $f : X \to Y$. While the codomain is the set of allowed images $f(x)$, it's not necessarily so that every element in the codomain is the image of some value. Instead, the set of actual images $\{f(x) \mid x \in X\}$ is in turn called the image of f.

If a function has no two $x_1 \neq x_2$ where $f(x_1) = f(x_2)$, we call f ***injective***. An injective function also has an ***inverse function*** f^{-1} that maps the image $f(x)$ back to the argument x, so that $f^{-1}(f(x)) = x$. The codomain and image can of course coincide, so that f^{-1} is defined for the entire codomain. Such a function is called ***surjective***. Finally, a function that's both injective and surjective is called ***bijective***.

As an example, the function $f : \mathbb{R} \setminus \{1\} \to \mathbb{R}$ defined by $f(x) = \frac{1}{1-x}$ has the domain $\mathbb{R} \setminus \{1\}$ ($x = 1$ would lead to zero division), the codomain $\mathbb{R}$, but the image $\mathbb{R} \setminus \{0\}$ (there's no x such that $f(x) = 0$). It's injective since $f(x_1) = \frac{1}{1-x_1} = \frac{1}{1-x_2} = f(x_2)$, after taking inverses on both sides, implies $1 - x_1 = 1 - x_2$ so that $x_1 = x_2$. The function has the inverse $f^{-1}(x) = \frac{x-1}{x}$. If f instead had the codomain $\mathbb{R} \setminus \{0\}$ (it's image) it would be surjective (and thus also bijective).

B.3 Sequences and Intervals

A ***sequence*** is an **ordered** collection of values (predominantly numbers) such as 1, 2, 1, 3, 1, 4, ... Sequences will mostly be a list of sub-scripted variables, such as $a_1, a_2, \ldots, a_n$. A shorthand for this is $(a_i)_{i=1}^{n}$, denoting the sequence of variables a_i where i ranges from 1 to n. An infinite sequence is given ∞ as its upper bound: $(a_i)_{i=1}^{\infty}$.

A ***subsequence*** of a sequence a_i is a sequence b_j where all the values of b_j appear in order in a_i, i.e. there is some $c_1 < c_2 < \ldots$ such that $b_1 = a_{c_1}$, $b_2 = a_{c_2}$ and so on.

In contrast, a ***subarray*** or ***sublist*** is a subsequence where all the values are consecutive (i.e. the subsequence $a_i, a_{i+1}, \ldots, a_j$ for some i, j.

B.4 Sums and Products

The most common mathematical expressions we deal with are sums of sequences of numbers, such as $1 + 2 + \cdots + n$. Such sums often have a variable number of terms and complex summands, such as $1 \cdot 3 \cdot 5 + 3 \cdot 5 \cdot 7 + \cdots + (2n+1)(2n+3)(2n+5)$. In these cases, sums given in the form of a few leading and trailing terms, with the remaining part hidden by ... is too imprecise. Instead, we use a special syntax for writing sums in a formal way – the *sum operator*:

$$\sum_{i=j}^{k} a_i.$$

The symbol denotes the sum of the $j - k + 1$ terms $a_j + a_{j+1} + a_{j+2} + \cdots + a_k$, which we read as "the sum of a_i from j to k".

For example, we can express the sum $2 + 4 + 6 + \cdots + 12$ of the 6 first even numbers as

$$\sum_{i=1}^{6} 2i.$$

Many useful sums have closed forms – expressions in which we do not need sums of a variable number of terms. For example:

$$\sum_{i=1}^{n} i = \frac{n(n+1)}{2}$$

$$\sum_{i=0}^{n} 2^i = 2^{n+1} - 1.$$

The sum of the inverses of the first n natural numbers has a very neat approximation, which we occasionally use in the book:

$$\sum_{i=1}^{n} \frac{1}{n} \approx \ln n.$$

This is a reasonable approximation, since $\int_1^l \frac{1}{x} dx = \ln(l)$.

There is an analogous notation for products, using the *product operator* $\prod$:

$$\prod_{i=j}^{k} a_i.$$

denotes the product of the $j - k + 1$ terms $a_j \cdot a_{j+1} \cdot a_{j+2} \cdot \cdots \cdot a_k$, which we read as "the product of a_i from j to k". In this way, the product $1 \cdot 3 \cdot 5 \cdot \cdots \cdot (2n-1)$ of the first n odd integers can be written as

$$\prod_{i=1}^{n} 2i - 1.$$

Notes

If you need a refresher on basic mathematics such as single-variable calculus, *Calculus* [47] by Michael Spivak is a solid textbook. It is not the easiest book, but one the best undergraduate text on single-variable calculus if you take the time to work it through.

For a gentle introduction to discrete mathematics, *Discrete and Combinatorial Mathematics: An Applied Introduction* [20] by Ralph Grimaldi is a nice book with a lot of breadth.

Logic in Computer Science [24] is an introduction to formal logic, with many interesting computational applications. The first chapter on propositional logic is sufficient for most algorithmic problem solving, but the remaining chapters shows many non-obvious applications that makes logic relevant to computer science.

One of the best works on discrete mathematics ever produced for the aspiring algorithmic problem solver is *Concrete Mathematics* [29], co-authored by famous computer scientist Donald Knuth. It is rather heavy-weight, and probably serves better as a more in-depth study of the foundations of discrete mathematics rather than an introductory text.

Hints

Chapter 1

1.1 Try dividing cards into smaller piles that can be sorted separately.
1.6 The optimal number of questions is 6.
1.8 What happens if we run the algorithm several times?

Chapter 2

2.16 Try solving it for the special case $y = 2$ first.
2.34 What exactly are we changing when we assign to the reference in `func`?)

Chapter 3

Chapter 4

Chapter 5

5.1 In the best case, line 4 of the insertion sort pseudo code never executes.
5.3 When is $\log^2 n < n$?
5.4 $c = 2$ for the upper bound.
5.5

1. Yes.
2. No.

5.6 Binomial expansion.
5.7 How many times does each digit in the number change?

Chapter 6

6.3 What element in an array can be removed in $\Theta(1)$?
6.6 What happens if you insert 2 elements into one stack, and then pop them to insert them into the other one?
6.7 How can you get the last element of a queue?
6.9 How many elements does each level in the tree have?
6.12

1. Sum the maximum number of steps each element can move.
2. What is the limit of a geometric series?

6.14 What happens when x is even?
6.16 Remove an element immediately after expansion.

Chapter 7

7.4 Use that $1.62 + 1 < 1.62^2$.
7.5 The positive root of the equation $x^3 = x^2 + x^1 + 1$ lies between 1.83 and 1.84.
7.6

1. The n choices are which of the two letters to put on each position in the string.
2. The n choices are whether to include each element or not.

7.7 Since the three recursions are structurally identical, they will have the same time complexity $T(n)$.
7.10 Add the lines one at a time.

Chapter 8

8.2 How many edges are added by the n'th vertex?
8.4 How many different degrees can the graph have?
8.15 What does the algorithm say that the distance for such a cycle is?
8.17 Add a new vertex to the graph.
8.19 Where in the queue are the vertices of the current distance and the distance plus one located?
8.22 Look at the first time the DFS algorithm to color vertices fails.
8.26 Induction.
8.27 How does the search visit the entire graph?
8.29 Look at the tree rooted in p.
8.31 What vertices can be the first in the ordering?
8.33 Assume there is a cycle of length 4 – shorten it.

Chapter 9

9.3 Consider two different hours. What values can the angles take?
9.5 How many calls are made for each value of *at*?

Chapter 10

10.2 Try $T = 12$.
10.6

1. Evaluate the sum for the sorted order and remember the triangle inequality.

10.8 How many intervals in the optimal solution can it overlap?
10.9 The proof is similar to the argument we used when deciding what interval to choose; use a swapping argument!
10.11 What does the previous solution do when allocating intervals into an infinite number of intervals?
10.16 Look at the first beam that exits at an incorrect row. Why was this beam not redirected into the right row?
10.17

1. There is a 1×3 counter-example.
2. It's in the solution to Exercise .

Chapter 11

11.3 How does *answers* for a vertex i and $next[i]$ relate?
11.5 The recursion only needs one more case.
11.16 What values of *best* are used?
11.20 How does the new recurrence look?
11.22 Is there duplicate work done when processing the subsets $\{1, 2, 3, 4\}$ and $\{2, 3, 4, 5\}$?

Chapter 12

12.1 $4^n = (2^n)^2$, and $1 = 1^2$.
12.3 What's the complexity of each step?
12.5 Can you find the smallest element of the combined sorted array?
12.7 Quickly find the next and previous occurrence of the k'th element.
12.16 Look at the set of links that a newly added link places on a cycle.
12.17 What's true for one of the subtrees of every non-leaf centroid vertex?
12.18 Assume there are two centroids.
12.19 The query to ask is the obvious one.
12.20 The idea behind the centroid finding algorithm gives you necessary conditions for the optimal meeting point.
12.21 When is $\mathrm{lca}(\mathrm{b}, \mathrm{c})$ and b equally far away from a?
12.22 Derive a way to answer "is u an ancestor of v?".
12.23 Compute the probability of the complement event.
12.24 Similar to Exercise 12.20.
12.25 Fix a and let b be uniformly random; consider now the possible subtrees.
12.26 Count occurrences.
12.27 Let A only contain values 0 and 1.
12.28 What do you want to check instead of $a = el$?
12.30 Consider the two parts of the course after splitting it in v.

Chapter 13

13.6 What happens when a single edge is added to a tree?

Chapter 14

14.9 Replace each trip between two non-terminal stations on a line by a sequence of trips that are as good.
14.12 For sufficiency, prove that a negative length cycle can reach v.

Chapter 15

Chapter 16

16.4 The solution is almost identical to the Knight Packing.
16.7 Formulate inequalities for the worst-case number of terminal states that must be checked for a winning and losing position with n moves remaining. What happens if they are equalities?
16.8 Proof by induction, similar to why a breadth-first search visits vertices in distance order.

Chapter 17

17.3 Use the symmetry of divisors.
17.5 Use the definition.
17.6 Use the definition.
17.14 What can be said about the remaining primes of N after a prime i is divided away?
17.30 Look at 10^n.
17.37 Look at $(n, i) = d$.

Chapter 18

18.1 What's the choice for each element?
18.7 How many equivalent rotations are there?
18.11 What's the order modulo l?
18.12 Look at what elements a specific one in the cycle is mapped to.
18.14 Both sides count the number of ways to do the same two choices.
18.15

1. Overflow of ab when $b \neq 0$ can always be detected by checking if $\frac{ab}{b} = a$ for unsigned integers.
2. Again, both sides represent the same two choices.

18.17 Consider the last of the n elements.

18.18

1. What are the possible sizes of a subset?
2. Find a bijection between odd- and even-sized subsets.
3. You're counting the subsets of all subsets.
4. Which is the last element chosen?

18.20 Let $b(i) = \binom{k+i}{k}$.

18.22 Where does the line cross the top-left to bottom-right diagonal?

18.27 What are the sets A, B, C?

18.28 How many times is any given element counted?

18.31 The big stack is the only limiting factor.

18.32 Look at all elements between the two that swap places.

18.33 There are several approaches. The simplest one uses segment trees.

18.34 You already know this is true for 3×3 subgrid.

18.35 How many swaps do you need to transform a permutation to the identity permutation?

18.36 Find a bijection.

Chapter 19

Solutions

Chapter 1

1.1 One possible solution is to first divide the cards into separate piles by values 1 – 100 000, 100 001 – 200 000, ... If we sort each such pile, the entire stack of cards is sorted by putting the piles together. Each such pile can be sorted the same way by instead dividing the cards up based on their 10′000 digits, and so on.

1.2

1. The input consists of two integers a and b, not both 0. The output should be the greatest common divisor of a and b.
2. The input consists of a sequence of real numbers, the coefficients x_i of a polynomial. The output should be a real number that is a root of the polynomial.
3. The input consists of two integers a and b. The output should be the product ab.

1.3 The input and output could be described using a protocol that dictates what the algorithm and the counterpart answering questions (called an *oracle*) are allowed to do and at what times. For example: the algorithm should repeatedly output an integer, to ask how it relates to the hidden integer. The oracle will then reply with the string "higher", "lower" or "equal".

The problem is called interactive since the algorithm interactively gathers information from the oracle.

1.4 Many common arithmetic and algebraic problems, such as those in Exercise 1.2 are solved using formal methods that constitute algorithms.

1.6 One can achieve 6 questions by always asking about the midpoint of the range of possible numbers. For example, after asking about the number 50, you know if the correct number is between 1 – 49 or 51 – 100.

1.8 Given an algorithm that is correct with a probability $0.5 + \alpha$ for some $\alpha > 0$, we can find the correct answer by running it many times and choosing the answer that was most common.

If the algorithm is incorrect on 25% of cases, there is a good chance a problem has a case that the algorithm fails on.

1.12 If we let the input n-letter word S have the letters $s_0, s_1, \ldots, s_{n-1}$, it reads the same backwards and forwards if $s_0 s_1 \ldots s_{n-1} = s_{n-1} s_{n-2} \ldots s_0$. We thus need to check all the letter pairs (s_0, s_{n-1}), (s_1, s_{n-2}) and so forth for equality.

```
1: procedure PALINDROME(string S)
```

```
2:    for i from 0 to n − 1 do
3:        if S_i ≠ S_{n−1−i} then
4:            return false
5:    return true
```

Chapter 2

2.7 Integer division in C++ is always rounded *towards zero*.

2.13 When the program reads input into a string variable it only reads the text until the first whitespace, so only the first word is read.

2.16 We only analyze the case where x and y are positive. Assume that $0 \leq a < \frac{x}{y} \leq a+1$, so that the result when rounded to an integer away from zero is $a+1$. Multiplying by y gives us $ay < x \leq ay + y$, so that $ay \leq x - 1 < ay + y$ (since all values are now integers). Finally, adding y to both inequalities gives us $ay + y \leq x - 1 + y < ay + 2y$. After dividing by y, we get $a + 1 \leq \frac{x-1+y}{y} < a + 2$. This means that the result of $\frac{x-1+y}{y}$ rounded towards zero is $a+1$, which is what we wanted.

Analysis for negative x and y is similar.

2.23

```
for (int i = repetitions - 1; i >= 0; i--)
```

2.34 A reference must refer to something that can be changed. Variables can be changed, but not the constant 4.

2.35

```
Point translate(double tx, double ty) {                return Point(x + tx, y + ty);
```

Chapter 3

3.2

- it works; members are not initialized,
- it works; members are initialized using the constructor,
- compilation fails if the vector is initialized with a size, otherwise it works,
- it works; members are initialized using the constructor.

3.4 `for (auto it = v.rbegin(); it != v.rend(); ++it)`

3.5 `Simon is fish food`

Chapter 4

Chapter 5

5.1 Consider the case when the array is already sorted. In this case, the inner loop on line 4 never executes, since $A[j] \geq A[j-1]$ for all j. Thus, only the lines that take linear time in total are executed, making $O(n)$ an upper bound on the base case. On the other hand, the loop on line 2 always executes a linear number of times no matter the case, so $\Omega(n)$ is also a lower bound. Thus, the algorithm has a $\Theta(n)$ best-case running time.

5.2 To compute the sum in $\Theta(n)$ time, we can add all the variables to a counter using a for loop, one at a time.

To solve the problem in constant time, the formula $1 + 2 + \cdots + n = \frac{n(n+1)}{2}$ can be used.

5.3 Let $n_0 = 7$. For any $n \geq 1$, we have $\log^2 n < n$ as $n < 2^n$ (which can be proved using either induction or simple calculus). In this case, $10n^2 + 7n - 5 + \log^2 n \leq 10n^2 + n^2 + n^2 = 12n^2$. Thus, with $c = 12$ we get the required statement.

5.4 Clearly $\max\{f(n), g(n)\} \leq f(n) + g(n)$ since the maximum of the two functions is always equal to one of the functions. This means that $f(n)+g(n) = \Omega(\max\{f(n), g(n)\})$ with $c = 1$. Similarly, $f(n) + g(n) \leq 2\max\{f(n), g(n)\}$ by the fact that each function individually is smaller than their maximum. Thus $f(n) + g(n) = O(\max\{f(n), g(n)\})$ with $c = 2$. Together this proves the statement.

5.5

1. This is clear with $c = 2$.
2. For any c, picking n such that $2^n > c$ gives us $2^{2n} = 2^n \cdot 2^n > c2^n$, so no c can satisfy the definition.

5.6 First, note that polynomials of higher powers are always greater than polynomials of lower powers eventually:

$$an^k < n^{k+1}$$

is true whenever $n > a$.

Next, we can write $(n + a)^b$ as the sum of n^b plus a lot of terms of lower powers of n using the formula for the binomial expansion. However, this means that $\max\{n^b, (n+a)^b - n^b\} = n^b$ for sufficiently large n. Thus, $(n + a)^b = n^b + ((n + a)^b - n^b) = \Theta(\max\{n^b, (n + a)^b - n^b\}) = \Theta n^b$ by a previous result.

5.7 The amortized complexity is equal to the number of times a digit changes in the number. The last digit changes every time, i.e. 2^n times. The second last digit only changes every second time when the last digit carries over. This holds in general, so that the total number of times each digit changes is $2^0 + 2^1 + \cdots + 2^n = 2^{n+1} - 1 = \Theta(2^n)$.

Chapter 6

6.3 Swap the element to be removed and the last element in the array. Now remove the last element of the array in $\Theta(1)$.

6.6 Perform all queue insertions into one of the stacks, and all queue removals from the other one. When the removal stack is empty, pop all elements in the insertion stack and insert them into the removal stack instead. This reverses the order of the elements in the first stack, giving us the queue ordering.

6.7 Insert all elements into the first queue. When poping an element, move all but the last elements in the first queue into the second, and then return the last element. Then, swap the roles of the queues, inserting elements into the second one.

6.8 Once the queue has more unused elements than used elements, create a new vector of size equal to the number of elements, move the elements into that vector and remove the old one. This never uses a vector of size larger than $2n$.

6.9 The i'th level in the tree has 2^{i-1} elements, since they double on each level. Consider the k'th element on the i'th level except the lowest one. It has index $2^0 + 2^1 + \cdots + 2^{i-2} + k = 2^{i-1} + k - 1$. Left of its children, there are $2(k-1)$ children, since the $k-1$ elements to the left each have 2 children. This means that the children have indices $2^i + 2(k-1) = 2^i + 2k - 2$ and $2^i + 2k - 1$. These indices satisfy the numbering properties.

6.11 The only parents that change are on the path that the new element bubbles up along. On this path, a parent is always replaced by its own parent (except for the last one, which is replaced by the new element). A parent is always greater than its grand-children, so after these swaps the heap property still holds.

6.12 Assume that the tree has $n = 2^k$ elements. The bottom n elements will move 0 steps. The $\frac{n}{2}$ elements of the next layer moves at most 1 step. The $\frac{n}{4}$ elements of the next layer moves at most 2 step, and so on. In total, this means that there are most

$$\frac{n}{2} + 2\frac{n}{4} + 3\frac{n}{8} + \cdots + k\frac{n}{2^k}$$

movements. Note that

$$\frac{n}{2} + \frac{n}{4} + \cdots \leq n$$
$$\frac{n}{4} + \frac{n}{8} + \cdots \leq \frac{n}{2}$$
$$\frac{n}{8} + \frac{n}{16} + \cdots \leq \frac{n}{4}$$

and so on. If we sum up all of these inequalities, we get that the original sum

$$\frac{n}{2} + 2\frac{n}{4} + 3\frac{n}{8} + \cdots + k\frac{n}{2^k} \leq$$
$$n + \frac{n}{2} + \frac{n}{4} + \cdots \leq 2n$$

proving that the elements are at most moved a linear number of time.

6.13 There are many ways that work. One of them is

```
int y = x + (x & -x);
int next = ((x^y) >> (__builtin_ctz(x) + 2)) | y;
```

6.14 If x has many trailing zeroes in its binary representation, the lower bits of Ax do too, reducing the usefulness to hashing.

6.15 No – the analysis still results in an expected constant number of collisions.

6.16 Removing an element immediately after the capacity increased causes it to shrink due to how the limits are chosen. Adding and removing an element at this boundary repeatedly causes each operation to take linear time, rather than amortized constant.

Chapter 7

7.1 They are 0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144, 233, 377.

7.4 We have that $T(n) = T(n-1) + T(n-2) + \Theta(1)$. Let m be the largest value that the $\Theta(1)$ term takes. Since scaling a function by a constant doesn't change its asymptotic behaviour, we can let all terms be divided by $\frac{m}{0.0044}$ so that the inequality $T(n) \leq T(n-1) + T(n-2) + 0.0044$ holds for all n.

The proof is now on induction on n. Assume that $T(n-1) \leq c \cdot 1.62^{n-1}$ and $T(n-2) \leq c \cdot 1.62^{n-2}$ where $c \geq 1$ and $n - 2 \geq 0$. Then

$$T(n) \leq c \cdot 1.62^{n-2} + c \cdot 1.62^{n-1} + 0.0044 = c \cdot 1.62^{n-2}(1 + 1.62) + 0.0044.$$

Next, note that $1 + 1.62 < 1.62^2 - 0.0044$, so that

$$T(n) \leq c \cdot 1.62^n - c \cdot 1.61^{n-2} \cdot 0.0044 + 0.0044 \leq c \cdot 1.62^n$$

where we used that $c \cdot 1.61^{n-2} \leq 1$ as $c \leq 1$ and $n - 2 \geq 0$.

Finally, choose c to be such that $T(0) \leq c$ and $T(1) \leq c \cdot 1.62$. By induction on n, the above result shows that $T(n) \leq c \cdot 1.62^n$ for all $n \geq 0$, so $T(n) = O(1.62^n)$.

7.5 It's given that $T(n) \geq T(n-1) + T(n-2) + T(n-3)$. If, by induction, $T(k) \geq 1.83^k$ for all $k < n$ we get

$$\begin{aligned} T(n) &\geq 1.83^{n-1} + 1.83^{n-2} + 1.83^{n-3} \\ &= 1.83^{n-3}(1 + 1.83 + 1.83^2) \\ &\geq 1.83^{n-3} 1.83^3 \\ &= 1.83^n \end{aligned}$$

so the claim holds for $T(n)$ too.

To formally prove an upper bound, you need to apply the same strategy as in the solution to Exercise 7.4.

7.6

1. Let $A(n)$ be the number of such strings. If the last character of the string was a B, the remaining string can be formed in $A(n-1)$ ways. If the last character of the string was an A, the second last character must have been a B (to avoid two consecutive A's). There are $A(n-2)$ ways in which the remaining string can be formed after fixing these two letters, so that $A(n) = A(n-1) + A(n-2)$. The base cases are $A(0) = 1$ and $A(1) = 2$.
2. Let $B(n)$ be the number of such subsets. If the element n is to be included in the subset, we can choose the remaining $n-1$ elements in $B(n-1)$ ways. If the element n is to be excluded from the subset, the element $n-1$ must be according to the problem. The remaining $n-2$ elements can then be chosen in $B(n-2)$ ways, for the recursion $B(n) = B(n-1) + B(n-2)$. The base cases are $A(0) = 1$ and $A(1) = 2$.

7.7 The time complexity fulfills $T(n) = 2T(n-1) + O(1)$. By induction, we get $T(n) = \Theta(2^n)$. We perform three calls with this complexity, but that is a constant factor so the complexity does not change.

7.10 If there are $n-1$ lines in the plane already, adding the n'th line adds an extra n regions – one for each intersection with another line, and one more for dividing the plane itself into a region. This gives the recursion $f(n) = f(n-1) + n$ with $f(0) = 1$ as base case. Using induction, $f(n) = \frac{n(n+1)}{2} + 1$.

Chapter 8

8.2 Adding the n'th vertex increases the number of edges by $n-1$, so the total number of edges is $1 + 2 + \cdots + (n-1) = \frac{n(n-1)}{2}$ (an identity that can be proven using induction).

8.3 The sum of all degrees is even. If there were an odd number of vertices of odd degree, the sum of their degrees would be odd.

8.4 If the graph has n vertices, there are only n possible degrees – those between 0 and $n-1$. The graph can not have one vertex with each of them, since the vertex with degree $n-1$ must be connected to the vertex with degree 0 – a contradiction. The n vertices can only have $n-1$ different degrees, so at least two of them have the same one.

8.8

1. The adjacency matrix is typically the best here, since the graph is not sparse at all. The exception is if edges adjacent to a vertex have an important ordering among themselves.
2. The adjacency matrix would take too much memory here. Adjacency lists are typically faster, unless the problem needs to support one of the operations for which adjacency maps are required.
3. Here, all three representations work, and which one is best depends entirely on what graph operations the problem requires.

8.12 In the second loop, we process the vertices in the exact same order, namely that in which they were added to either *atDistPlusOne*. The only other change is that we use *distance*[*from*] rather than *dist* when computing the distance of a neighbor, but these are the same – the distance of the vertex *from*.

8.15 No – the distance reported for these cycles are always larger than for the smallest cycle, so we don't get an incorrect answer by including them.

8.17 Add an new vertex as single source and connect it to the original multiple source.s The shortest distance to the new vertex is the shortest distance to any of the original sources plus 1.

8.19 Replace the queue with a double-ended queue ("deque") instead and push vertices visited through 0 weight edges to the beginning of the queue.

8.22 In a two-colored graph, the vertices on a cycle alternate between red and blue, so it has an equal number of red and blue vertices. The total number of vertices on the cycle is thus an even number.

For the other direction, take a vertex s and color each vertex v blue or red depending on whether $d(s, v)$ is even or odd. If two adjacent vertices v and u have the same parity, the distance must be the same (the distance of adjacent vertices differ by at most 1). Assume that the shortest paths from s to v and u meets at some vertex w. Then there are disjoint paths from w to v and from w to u of the same length. Combining these paths with the edge $\{u, v\}$ thus forms a cycle of odd length. This means that the suggested coloring works as long as there is no odd cycle.

8.24 Assume that there are two different paths from u to v. If they meet at some other vertex w, there are also two paths between u and w. Replace v with w.

Since this operation shortens the length of these paths, we can perform this until the two paths between u and v are disjoint. The combination of two disjoint paths is a cycle.

8.25 Construct a walk in the following way. First, pick an arbitrary vertex. Now, repeatedly walk along an arbitrary edge, except for walking backwards to the vertex you just came from. Since the graph has no cycles, you never get back to a previously visited vertex. On the other hand, the walk must end since there is a finite number of vertices. This means that you at some point must reach a vertex that only had a single edge – the one we walked to get to the vertex.

This proves that there is at least one leaf. To find the other, perform the same procedure but start from the leaf.

8.26 This is true for all graphs on 1 vertices. Assume that this holds for all trees with k vertices. A tree with $k + 1$ vertices must have a some leaf. Removing this leaf from the tree removes a single vertex and edge, but results in a tree on k vertices with, by inductive assumption, $k - 1$ edges. This means that the original tree had k vertices.

8.27 We enter a vertex exactly $n - 1$ times and visit all n vertices. Thus, the edges used when entering a vertex during the search all form a spanning tree.

8.29 Assume that the DFS from c reaches another vertex v that was marked as invalid.

Why did the DFS that marked v as invalid not also mark c as invalid? There must have been something that prevented it from passing the edge from v to the vertex closer to p – let's call it u. This can only happen if u and v formed a parent-child constraint, so that v was actually the start of the DFS. If this is the case, then all other vertices in reachable from v must also have been marked as invalid by that DFS.

8.31 The ordering can be constructed one vertex at a time. If a vertex has no outgoing edges (i.e. dependencies), it can be put first in the ordering. All incoming edges to it can then be removed and the remainder of the ordering constructed iteratively in the same way. If the graph has no cycles, there must be a vertex without an outgoing edge. Choose a random vertex and repeatedly follow an outgoing edge from it. Two things can happen. Either we repeat this infinitely, and so we must revisit a vertex – there is a cycle, which we assumed to be false. The only other possibility is that the process ends because we did indeed find a vertex without an outgoing edge.

8.33 Clearly, no cycle of length 1 or 2 exists. Let the shortest cycle have length at least 4, and the first vertices on the cycle be $p_1, p_2, p_3, \ldots, p_l$. By assumption, $p_1 \to p_3$ is an edge. Otherwise, $p_1 \to p_2 \to p_3 \to p_1$ is a cycle of length 3. But then $p_1, p_3, \ldots, p_l$ is a shorter cycle. Thus the shortest cycle could not have length at least 4, so it must have been 3.

Chapter 9

9.3 For two times h_1, m_1 and h_2, m_2, the difference between their angles is $300(h_1 - h_2) + 5(m_1 - m_2) - 60(m_1 - m_2) = 300(h_1 - h_2) - 55(m_1 - m_2)$. If $h_1 = h_2$, then $m_1 = m_2$ if this difference is 0. Otherwise, the second term must also be a multiple of 300 to make the difference 0. If $55(m_1 - m_2)$ is a multiple of 300, $m_1 - m_2$ must be a multiple of 60, which is only possible if they are equal. But if they are equal, so must the hours.

Thus, no two times can have the same angle.

9.5 A single call is made for $at = 0$. This call makes two calls with $at = 1$, and these two in turn make two calls with $at = 2$, and so on. This means that there will be 2^{at} calls for every value of at from 0 to N, i.e. $2^0 + \cdots + 2^N = 2^{N+1} - 1$ calls in total.

9.7 Assume that we in a connected graph have placed at least one drone, but all remaining intersections have four neighbors. Then, none of these intersections can be neighbors with any vertex we have removed so far. This means that the set of intersections removed and the set of intersections remaining are actually disconnected, to the contrary of our assumption of connectedness.

Chapter 10

10.2 For $T = 12$ the optimal solution is $6 + 6$ (two coins), but the greedy solution chooses $7 + 1 + 1 + 1 + 1 + 1$ (6 coins).

10.3

1. A shelf of width 7 with two books of width 3 and four books of width 2.

2. Let $w_1 = w_2 = w_7 = 1$ and $w_3 = w_4 = w_5 = w_6 = 0$.
3. A counterexample is $R = 2$, $C = 6$, $a = 5$, $b = 4$, $c = 3$.

10.6

1. Let a_i and a_j be the smallest and largest elements. By the triangle inequality ($|a| + |b| \geq |a + b|$), the terms $|a_i - a_{i+1}| + \cdots + |a_{j-1} - a_j|$ together equal at least $|a_i - a_j|$, so the full sum can not be smaller. However, when all the numbers are sorted, the entire sum evaluates to the same value, so it must be optimal.
2. For example, $3, 0, 1, 2$.
3. For example, $1, 3, 4, 2, 0$.

10.8 Since we choose the shortest interval, we know that it can't overlap three intervals in the optimal solution. If it did, it would have to contain one of them in its entirety, but then we haven't picked the shortest interval. Thus, for each shortest interval we pick, at most two optimal intervals are removed, so we pick at least half as many as in the best solution.

10.9 Assume that we have chosen a number of intervals according to this strategy, and that it so far corresponds to an optimal solution, but that adding the next interval was not optimal. It can not be sub-optimal due to the interval itself not belonging in an optimal solution. If it does not, then look at whatever interval was placed in its place in the designated subset. Without loss of generality, we can exchange that other interval for the one we think is optimal, and not get a worse solution.

Let the subset we want to place the interval in (call it A) have r_a as its right endpoint, and the optimal subset (call it B) have r_b as its right endpoint. There must be some other interval we placed next in A instead (otherwise we just insert our interval there). It must also fit in B (since by assumption $r_a \geq r_b$). Thus, we can swap these two intervals and get a solution that is not worse.

10.10

1. If we so far have scheduled intervals up to r, it is irrelevant which of the intervals of minimum $r' > r$ that we choose, since the left endpoint of the chosen interval is never used.
2. Tings are slightly more subtle in this case. The key is that no matter the ordering of the ties for a given right endpoint, the exact same subsets will be chosen to place an interval in (but possibly in different orders, and with different intervals to be inserted). Assume that the subsets chosen for some arbitrary ordering have right endpoints $r_1 \geq r_2 \geq \cdots \geq r_n$, and that an other ordering chose different subsets; instead of r_k, the k'th largest subset has right endpoint r'.

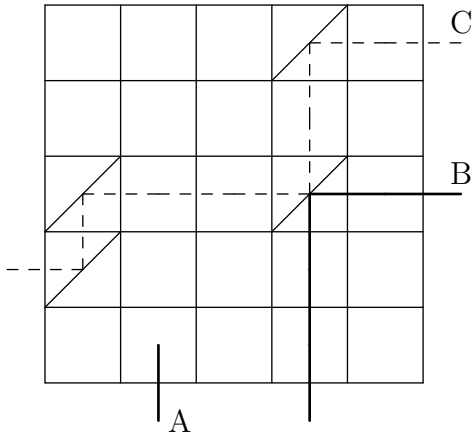
   We can not have that $r > r_k$. If $l_1, \ldots, l_k$ are the left endpoints of the intervals added to those k subsets with greatest r_i, they must all have been placed in the subsets

$r_1, \ldots, r_{k-1}$. If one of them weren't, an attempt would have been made to place it into the unused r_k – a contradiction to that $r > r_k$. Can $r < r_k$ then? No – the proof is entirely symmetrical, so we can swap the role of the two orderings so that $r > r_k$ again.

10.11 Assume that the optimal solution is K. The algorithm will then use exactly K subsets, since the behavior of the previous solution where we preallocate K subsets is identical to its behavior where we add a new subset when we can't insert an interval into an existing subset. This is true since the algorithm always tries to choose the "most full" subset first. Since the behavior is identical, the algorithm will terminate using exactly K subsets.

10.13 The algorithm almost works as-is simply by treating L as meaning "the rightmost point to have been covered so far". The only problem is that if $L = R$, we would produce an empty covered, even though the point L itself has not been covered. This can be fixed by updating the check on line 4 to read "while $L < R$ or the cover is empty".

10.16 Call the first beam to exit at the wrong row A, and the beam *entering* through A's intended exit hole B. The beam B hits exactly one mirror and exits downward. Otherwise, it would be hitting a mirror vertically that some other processed beam is already reflecting against. Call the beam whose processing placed that particular mirror C.



Beams B and C meet in a point and exit on different sides of A, so they cut A of from any possible exit forcing A to cross at least one of them. If it crosses B, the algorithm would add a mirror at the point of crossing to force the beam upwards and make it exit correctly.

The other option is that it crosses C somewhere while going upwards in some given column. Since A is traveling upwards, we must have added a mirror somewhere because there was another mirror further up in the column. To actually cross C, this mirror would have to be above the line C. Since C was itself not directed upwards at this point (adding a mirror and preventing the beams from crossing), it must be so that C was actually on its

way to leave the grid at the correct row. This is impossible, since we know that the point where A first cross C must be before C is reflected upwards at the point where it touches B (and thus before it reaches its correct row of exit).

One more possibility exists, demonstrated in the figure. A can go upwards not due to a mirror, but because it entered the grid from below. Since all beams exiting upwards does so correctly, there is some mirror in this column preventing A from leaving. By the same reasoning as before, this mirror must actually be placed at a lower point than where A would cross C.

10.17

1. For example the 1×3 case where the first three beams should exit horizontally in order, and the last beam exit to the right.
2. We assume that whenever a beam is going vertically, the beam supposed to exit through that column has already done so (and thus placed a mirror in the column). This assumption is not true when going right-to-left. Several crucial parts of the proof depend on this fact.

Chapter 11

11.2

1. Yes – there are many recursion paths that lead to the same argument i. In fact, when calling Fibonacci(n) the number of times Fibonacci(i) is called is the $(n - i)$'th Fibonacci number.
2. No. The use of the *seen* array guarantees that the function is only called once for each vertex in the graph.
3. Yes. This is most obvious when e is a power of two. There are then 2 calls with $\lceil \frac{e}{2} \rceil$, 4 to $\lceil \frac{e}{4} \rceil$ and so on. In total, there are $\Theta(e)$ calls. When e is not a power of two, there are still only at most $2\log_2(e)$ possible values of e.

11.3 The longest path $answers[i]$ is always 1 higher than $answers[next[i]]$. You can find $next[i]$ by checking if this relation holds for each vertex u that i has an out-edge to.

11.5 Whenever Simone performs the short sell (the case $C(i) = 100P_i - K$), she may already have some profit earned from her earlier sells that should be added to the amount. This means that $100p - K$ in the code should be $100p - K + profit$.

11.7 Otherwise the function would be invoked with the same parameters, causing an infinite recursion.

11.13 No. The base case of not needing to build an exit on a vertex that also have no children that need one is handled by line 4 (if $d \leq D$), where the cost evaluates to 0.

11.14 This represents not having an exit further up that can be used. When called with $d = D + 1$, the recursive call on line 4 would not trigger, so no calls with higher d will be made.

11.16 For each iteration of i, only $best[i]$ and $best[i-1]$ are used. Keep only those two arrays in memory at all times.

11.20 The recurrence instead becomes LNDS(i) = $1 + \max_{s_i \geq B[x]} x$. In the code, we would translate this to the equivalent LNDS(i) = $\min_{s_i < B[x]} x$. The C++ change is that the `lower_bound` should be on the pair `make_pair(S[i], i)` instead. Since all existing entries with the same `S[i]` have a lower `i`, this guarantees that we find the first greater element instead.

11.22 Keep a 2^n array marking all the subsets that are possible extensions. For each of the k subsets, mark them in the array. Now, go through the array in order of descending number of elements in the subsets. If the subset s is marked true, then for each $x \in s$ mark the set $s - x$ true as well.

Chapter 12

12.1 We have that $4^n - 1 = 2^{2n} - 1 = (2^n + 1)(2^n - 1)$ (difference of squares). Among the three consecutive integers $2^n - 1$, 2^n, and $2^n + 1$, at least one must be divisible by 3, but this can clearly not be 2^n. Thus one of the others must be, but they are both factors of $4^n - 1$, which then too is divisible by 3.

12.3 After i steps, there are 2^{k-i} integers left. Each step takes linear time in the number of integers. The complexity is then $\Theta(2^k) + \Theta(2^{k-1}) + \cdots = \Theta(2^k)$.

12.5 Since both arrays are sorted, the smallest element in each array is first. Clearly the smallest element of the combined array must be the smallest of those two elements. Hence we can repeatedly find the next element of the combined array by choosing and removing the smaller of the two smallest elements of the partial arrays. When one array runs out of elements the remaining elements of the other array are appended to the combined array.

12.7 For the k'th element, denote the index of the previous and the next occurrence of it in the array by b_k and f_k (or $-\infty$, ∞ if there are none). Answering the given query is then as easy as checking whether b_k or f_k lies in the interval. These numbers can be precomputed in linear time. Iterate through the array one element at a time and keep a hashmap with the index of the last occurrence of each element. Then b_k is simply the value in the hashmap for the k'th element. Computing f_k is done in an analogous manner.

12.16 By adding an edge $\{u, v\}$, all edges on the path from u to v in the original tree will be on a cycle. Let u' be any of the leaves in the subtree of u (going away from v), and v' the corresponding leaf for v. The path between u' and v' goes through at least all the edges between u and v, so adding the edge $\{u', v'\}$ instead does not change the two-connectedness of the graph.

12.17 Pick an arbitary vertex in the tree as root of the tree. If it has no subtree with a strict majority of the leaves, you've found c. Otherwise, go into the subtree that has a majority. You never move back up along an edge you once traversed down into: for any edge, each leaf lies either above or below the edge, so only one side has a strict majority. There's a finite number of vertices, so the algorithm must terminate at some point because it found

c. The only remaining detail is to memoize the leaf counts for the subtrees to get a linear rather than quadratic time complexity.

12.18 Assume that u and v are two centroids. Then, the subtree of u that v lies in has at most $\frac{N-1}{2}$ vertices. A consequence is that the other subtrees has at least $\frac{N-1}{2}$ vertices altogether. However, all those other subtrees and u itself lies in the same subtree of v, which thus has size at least $\frac{N+1}{2}$ which is strictly more than half.

12.19 Ask what the optimal meeting point between a, b and c is. If b and c are in different subtrees, then clearly a lies on the path between b and c, and thus it is the optimal meeting point. If they are in the same subtree, a can never be the optimal meeting point, since the immediate child of a in that subtree is a better meeting point.

12.20 Let v be the optimal meeting point. No two of a, b, c lie on the other side of the same adjacent edge of v, as moving along that edge decreases the sum of the distances. Thus b and c must lie in the subtree of v (possibly $v \in \{a, b, c\}$), or else v's parent would be a better meeting point. If b and c lie in different subtrees of v, then v is their LCA. If they lie in the same subtree, v is not the optimal meeting point since moving down that subtree decreases the sum of the distances, unless $v \in \{b, c\}$, but then v is also the LCA of b and c.

12.21 Since $\text{lca}(b, c)$ is an ancestor of b and c it can never be further away from a than the closest of b and c. The remaining case is that b, c and $\text{lca}(b, c)$ are all equally far from a. Note that all ancestors of b have different distances from a, so if $\text{lca}(b, c)$ is equally far from a as b is, we have $\text{lca}(b, c) = b$, and similarly $\text{lca}(b, c) = c$, so that $b = c$.

12.22 If (and only if) a, b and c line on a path are any of them the optimal meeting point. Thus, whenever b or c is the optimal meeting point, we know which one of them is an ancestor of the other. With this information there are many ways of reconstructing the tree in quadratic time.

12.23 If there are l leaves remaining, the probability that yet another leaf is picked is $\frac{l}{l+1}$ and the probability it is not $\frac{1}{l}$. The probability that fewer than $\frac{k}{2}$ leaves are picked therefore equals

$$\frac{1}{k+1} + \frac{k}{k+1}\frac{1}{k} + \frac{k}{k+1}\frac{k-1}{k}\frac{1}{k-1} + \cdots + \left(\prod_{i=1}^{\frac{k}{2}-1} \frac{k-i+1}{k-i+2}\right) \frac{1}{\frac{k+2}{2}} = \frac{k}{2}\frac{1}{k+1} < \frac{1}{2}.$$

12.24 Let v be the optimal meeting point. At least two of x, y, z must lie in the subtree rooted at v, or else v's parent would be a better meeting point (or $v \in \{x, y, z\}$, but then v is LCA of itself and whichever of x, y, z is its descendant). If those two lie in different subtrees of v, then v is their LCA. If they lie in the same subtree of v, then the root of that subtree would be a better meeting point.

12.25 Let a be fixed and b be a random vertex. The possible LCA's are then exactly the ancestors of a. Let the subtrees rooted at those ancestors have the sizes $s_1 \le s_2 \cdots \le s_{k-1} \le s_k = N$. We get the subtree s_1 exactly when b belongs to the subtree, which happens

with probability $\frac{s_1}{N}$. Similarly, the probability that s_2 is chosen becomes $\frac{s_2-s_1}{N}$, and so on. Summing probabilities, we see that the subtree has size at most $s_l \leq qN < s_{l+1}$ with probability $\frac{s_1+(s_2-s_1)+(s_3-s_2)+\cdots+(s_l-s_l-1)}{N} = \frac{s_l}{N} \leq \frac{qN}{N} = q$. Since this inequality holds for every fixed a, it also holds for a random choice of a.

12.26 The majority element occurs some $k > \frac{N}{2}$ times. After removing two distinct elements, it occurs at least $k - 1 > \frac{N-2}{2}$, which is strictly more than half of the remaining $N - 2$ elements.

12.27 Let A contain only two different values, each occurring $\frac{N}{2}$ times. Exchanging the positions of all elements of the different values also causes Majority to change its return value.

12.28 Iterate through all vertices in the tree. Instead of checking $a = el$ we want to check if a and el are vertices of the same subtree, which has been described earlier.

12.30 Each path that passes through the v can be decomposed into two subpaths that has one of their ends in two different subtrees and their other ends being v. For each subtree, perform a DFS from v into that subtree to find the lengths and number of edges of each path that ends in that subtree. Now iterate through each of the subtrees, and keep a hashmap mapping path lengths to the smallest number of edges for each path in one of the previous subtrees. When finding a path of length K' in a subtree, we check the hashmap for a path of length $K - K'$ in one of the other subtrees to get the best path with length K.

Chapter 13

13.6 Consider the old plan with the edge e from the new plan added. Since this graph now contains N edges, it is no longer a tree and thus contains a cycle. At least one of the edges in this cycle must belong only to the old plan, since the new plan doesn't contain a cycle. Removing an edge that lies on a cycle never disconnects a graph, so the other $N - 1$ edges must still form a single connected tree.

13.11 We need to compute two different kinds of prefixes. First, the normal ones: $a_0 \star a_1 \star \cdots \star a_j$. Since the operation is not commutative, we also need to compute the prefixes $a_{i-1}^{-1} \star \cdots \star a_1 \star a_0$ in order to remove the values up to a_{i-1} (here, x^{-1} denotes the inverse element of x).

Chapter 14

14.1 Assume to the contrary that there is a shorter path P from s to v_i. The path $P \to v_{i+1} \to \cdots \to v_n$ is thus shorter than the original path, so the original path can not have been a shortest one.

14.2 No, we never assumed that edges were undirected. Dijkstra's algorithm works perfectly fine even with directed edges.

14.6 Consider the counter-example path $P = s \to v_1 \to \cdots \to v_{k-1} \to v_k$ with the fewest number of edges. Then $s \to v_1 \to \cdots \to v_{k-1}$ is a shortest path to v_{k-1} and have length

$d(s, v_{k-1})$. Let the edge $v_{k-1} \to v_k$ have weight w. Since we assumed P was not a shortest path, we must have that $d(s, v_k) < d(s, v_{k-1}) + w$. This contradicts that $v_{k-1} \to v_k$ is part of some shortest path since there is a shorter path from s to v_k not including this edge.

14.7 If $d_s(t) = d_s(u) + w + d_t(v)$, the shortest path from s to u, followed by $u \to v$, ending with the shortest path from v to t has the same distance as a shortest path from s to t.

Conversely, if there is a shortest $(s - t)$-path including the edge $u \to v$, the paths from s to u and from v to t must be a shortest paths, or the $(s - t)$-path could be shortened by replacing one of those subpaths with a shorter path. Thus the length of the path must be $d_s(u) + w + d_t(v)$ and equal to $d_s(t)$ since this was a shortest $(s - t)$-path.

14.9 Consider a trip between two stations a and b, neither of which are the first or the last, where a is closer to the first station. Replace the trip by three trips: from a to the last station, back to the first station, and then to b. These three trips have the same waste as the trip directly from a to b.

14.12 The condition is indeed necessary. If $D(|V| - 1, x) = D(|V|, x)$ for each x that can reach v, none of those values ever change. To prove that formally, let $k > |V|$ be the smallest integer such that $D(k, b) < D(k - 1, b) = D(|V| - 1, b)$ for some b that can reach v. This happens only when there's an edge $a \to b$ where $D(k - 1, b) > D(k - 1, a) + w(a, b)$, implying $D(|V|, b) = D(|V| - 1, b) > D(|V| - 1, a) + w(a, b)$, a contradiction.

To see that it is also sufficient, consider a walk with $|V|$ edges to v of length $D(|V|, v)$. Since it uses $|V|$ edges, it must contain a cycle. If the cycle has a non-negative length, it could be removed to obtain a walk of length $\leq D(|V|, v)$ using fewer edges, contradicting $D(|V| - 1, v) > D(|V|, v)$. Thus the walk contains a negative length cycle. Traversing it an arbitrary number of times results in a walk to v of arbitrarily short distance.

14.13

1. Consider a cycle $v_1 \to \cdots \to v_n \to v_1$ in the path reconstruction graph. Assume WLOG that $v_n \to v_1$ is the *last* edge that decreased a distance in the cycle. When $v_1 \to v_2$ was used to decrease the distance to v_2, v_1 had a higher distance than it does now (since it was decreased most recently), so it must be possible to use that edge again to decrease the distance to v_3. But by the same reasoning we must now be able to decrease the distances to $v_3, v_4, \ldots$, all the way until we decrease the distance to v_n and finally v_1 once more. Since traversing the cycle lowered the distance to v_1, it has negative length.

2. We prove the contrapositive instead: if the path reconstruction graph lacks cycles, the graph lacks negative length cycles. If the graph is a DAG it must be a rooted tree in s, since every other vertex has exactly a single in-edge by construction of the graph. This means that $D(s) = 0$ has remained unchanged through the entire algorithm.

   Now, consider a path $s = v_1 \to v_2 \to \cdots \to v_n$ in the path reconstruction tree. We

prove by induction on the length n on such paths that

$$D(v_n) = w(v_1 \to v_2) + \cdots + w(v_{n-1} \to v_n).$$

This is trivially true for $n = 1$ since $D(s) = 0$.

Assume that equality holds for a path up to v_n that has an edge $v_n \to v_{n+1}$ in the path reconstruction tree. Since the path has $n - 1 \leq |V| - 1$ edges, the value $D(v_n)$ was reached by at most $|V| - 1$ Bellman-Ford iterations, so after $|V|$iterations it must hold that $D(v_{n+1}) \leq D(v_n) + w(v_n \to v_{n+1})$. Because $v_n \to v_{n+1}$ was the last edge to relax v_{n+1}, we also have equality, implying that the statement holds true also for the path to v_{n+1}. By induction, the equality holds for path in the path reconstruction tree.

We have proved that after $|V|$ iterations of Bellman-Ford, each distance $D(v)$ is exactly the length of the $(s-v)$-path in the path reconstruction tree. These distances must also have been found already after $|V| - 1$ iterations, so all distances have converged. As we've shown earlier, this implies that there are no arbitrarily short distances, i.e. the graph lacks negative cycles.

14.15

1. Let k be the highest-numbered vertex (except possibly v) on the cycle. Splitting the cycle on k and v gives us a shortest $(v - k)$-path and a $(k - v)$-path with lengths $D(v, k, k-1) + D(k, v, k-1) < 0$. Thus $D(v, v) \leq D(v, v, k) \leq D(v, k, k-1) + D(k, v, k-1) < 0$.
2. During execution $D(i, j, k)$ is always the length of a walk from i to j.

14.16 For each pair (i, j), keep track of the second-to-last vertex on the path shortest $(i - j)$-path. Initially, that is i if $i \to j$ is an edge, as well as i for the pair (i, i). Whenever a shorter path is found through an intermediate vertex k, the new second-to-last vertex is that of the shortest $(k - j)$-path. These values can then be used to backtrack the full path one vertex at a time.

14.18 Assume that we arrive to a non-v vertex the i'th time. Then we have used i incoming edges and $i - 1$ outgoing edges, so by assumption there is at least one more outgoing edge that can be followed. Since the walk must eventually end (there are finitely many edges), it can thus only do so at v.

14.19 Since exactly one incoming and outgoing edge are used each time a vertex p_i is visited, a vertex on p_i with an incoming edge must also have an outgoing edge. Split the graph into two sets of vertices P and Q containing those vertices on the walk and not on the walk, respectively. The graph is weakly connected, so there must exist at least one edge between P and Q. Either it's outgoing from some p_i, or it's incoming, but that in turn also implies that it has an outgoing edge.

14.20 Pretend that you've added a new vertex r and two edges $r \to s$ and $t \to r$, so that all vertices have equal in-degree and out-degree. Running the Eulerian walk algorithm from r is equivalent to starting it from s and ending at t in the original graph since the first and last edge by necessity are $r \to s$ and $t \to r$.

14.21 The only time we use the directedness of the edges in the algorithm is to conclude that whenever we arrive at a vertex, it has either no unused incoming edges, or at least one unused outgoing edge. This statement is vacuously true since "incoming edges" don't exist at all in undirected graphs.

14.23 Since the in-degree and out-degree of all vertices are equal, traversing the outgoing spanning tree edge from some vertex implies *all* other adjacent edges were traversed. Next, assume that all edges adjacent to some vertex u have been traversed during the walk. In particular, all edges in the spanning tree pointing to u were then traversed, so in turn all edges adjacent to u's children (in the spanning tree) must have been visited. Applying this reasoning recursively, all edges adjacent to a vertex in u's subtree must have been traversed.

Finally, as we have shown earlier, the walk will terminate in the same vertex that it starts from, i.e. v, and only do so once all its adjacent edges have been visited. But then all edges adjacent to a vertex in v's subtree must have been visited. As v is the root of the spanning tree, its subtree is the entire graph, so all edges in the graph were indeed traversed, making the walk Eulerian.

14.26 Perform a DFS starting at the root of a normal spanning tree, always prioritizing to traverse edges that are part of the tree. Now, consider the first edge $\{u, v\}$ that is not part of the normal spanning tree, where v is an ancestor of u, that the DFS adds to the DFS tree. Clearly the edge can't have been added when going from u to v, since the only way the DFS can have gotten to u is by first going to v. However, since the DFS first picks edges that are part of the normal spanning tree, it will follow the path from v to u in the tree before traversing the edge $\{v, u\}$. This contradicts the assumption that any edge $\{u, v\}$ not part of the spanning tree was added to the DFS tree, so the DFS did indeed produce the normal spanning tree.

14.28 Assume for the sake of contradiction that two biconnected components share the vertices u, v. If any vertex is removed, both components are still individually connected since they were biconnected, and since at least one of u and v remains and are in both components, they are also still connected to each other. The union of the components are thus also biconnected, which is impossible by definition since they were not maximal.

14.29 After removing a single vertex from a cycle, the remaining vertices all lie on a path, so they are still connected. Thus a cycle is always biconnected, so it must be part of a common biconnected component.

14.30

1. Removing an edge $\{u, v\}$ disconnects a graph if and only if there is no other path from u to v. This is equivalent to $\{u, v\}$ being part of a cycle: if a path exists, it can

be extended to form a cycle, and if $\{u, v\}$ is on a cycle, the rest of the cycle is a path between u and v. A bridge can thus be defined as an edge that isn't part of any cycle. By the previous subexercise, all edges on a cycle is part of the same biconnected component, so a non-bridge edge $\{u, v\}$ – guaranteed to lie on cycle – can't be a maximal biconnected set.

Conversely, let $\{u, v\}$ be an edge in a biconnected component that contains at least one more vertex x. By definition removing u doesn't disconnect the component, so there's a path between v and x that doesn't use the edge $\{u, v\}$. Similarly, there is a path between v and x that doesn't pass through $\{u, v\}$. Consequently u and v are connected by some path except the edge $\{u, v\}$, so $\{u, v\}$ does actually lie on a cycle, and thus cannot be a bridge.

2. Let v be shared by different two biconnected components A and B. Pick a neighbor a and b of v in A and B respectively. If v isn't a cutvertex, there's a path between a and b after removing v, so we can form a cycle by adding the edges $\{a, v\}$ and $\{v, b\}$ to the path. All vertices on a cycle lie are part of a biconnected component, so a, v and b all belong to some component C. Distinct biconnected components can't have more than one vertex in common, so we must have $A = C = B$, contradicting that v wasn't a cutvertex.

   For the other direction, let v be a cutvertex and assume that it's only part of a single biconnected component V. Since v is a cutvertex, removing it causes some vertices a and b to become disconnected. In the original graph, consider the paths from a to v and b to v, letting $\{a', v\}$ and $\{b', v\}$ be the last edges on the paths. Remember that both ends of every edge are always in the same biconnected component, so since v only belongs to V, we know that a' and b' must belong to V as well. V is a biconnected component though, so after removing v there is still some path between a' and b' by definition, which by extension gives us a path $a \to a' \to b' \to b$, a contradiction. Thus, the assumption that v wasn't shared by two biconnected components was false.

Chapter 15

Chapter 16

16.4 The first player wins; after placing a coin in the center of the table on their first move, they can rotate their opponent's move 180° every time.

16.7 Let $W(n)$ be the worst-case number of terminal states we must check when recursing to recurse into a winning position with n moves left, and $L(n)$ the same but for a losing position. Losing positions must recurse into two winning positions, so $L(n) = 2W(n-1)$. For winning positions, the worst case is one winning and one losing move. If both moves were winning, we'd have to only look at a single move which would give the worst-case

$W(n) = L(n-1)$. Having one losing move as well is strictly better, since we get the equality $W(n) = L(n-1) + 0.5W(n-1)$. Substituting $L(n)$ with the first equality yields $W(n) = 0.5W(n-1) + 2W(n-2)$. Solving the quadratic equation $x^2 = 0.5x + 2$ gives $x = \frac{1+\sqrt{33}}{4}$, from which the result follows.

16.8 Assume by induction that all positions with DTW $\leq n$ were processed to the queue in order. We then show that processing all positions with DTW $= n$ adds exactly the positions with DTW $= n+1$ to the queue. If a losing position has DTW $n+1$, all moves have DTW at most n, and at least one move exactly DTW n. Thus, it will reach the $movesLeft[u] = 0$ condition for the first time when processing the DTW n positions and be added to the queue now. If a winning position has DTW $n+1$, there is a losing position with DTW n that we can reach from the position. Since we process all such losing positions correctly by assumption, all these winning positions will be added to the queue now too. Proving that all positions that are added have the correct DTW is similar.

Chapter 17

17.1 7 only has the trivial divisors 1 and 7. 18 has the divisors $1, 2, 3, 6, 9, 18$. 39 has the divisors $1, 3, 13, 39$.

17.3 Each divisor d can be paired with the corresponding divisor $\frac{n}{d}$ except when $d = \frac{n}{d}$. But then $d^2 = n$, so n is a perfect square. For the other direction, it's enough to note that $(\sqrt{n}, \sqrt{n})$ is indeed a divisor pair exactly when n is a perfect square.

17.5 By definition, there exists q, q' such that $a = bq$ and $b = cq'$. Then $a = (cq')q = c(q'q)$, so $c \mid a$.

17.6 By definition there are q, q' such that $b = aq$ and $c = aq'$. Then $b + c = aq + aq' = a(q + q')$, so that $a \mid b + c$.

17.8 The complexity would be the same:

$$\sum_{i \leq \sqrt{n}} \frac{n}{i} = n \ln \sqrt{n} = \Theta(n \ln n).$$

17.12 All non-primes below N^2 must have a prime divisor below N, so primality can be checked by testing divisibility by all of them.

17.14 If there are two of more primes left, they must both be greater than i, so the algorithm will still find them. The worst-case complexity is unchanged, since for prime N no prime is ever factored out.

17.16 There are $\lfloor \frac{m}{n} \rfloor$ numbers up to m that are divisible by n (every n'th). Generally, there are $\lfloor \frac{m}{p^k} \rfloor$ numbers divisible by the prime p k times. The formula follows from counting this for each k.

17.20 $(a, 0) = a$ and $(a, a) = a$ both follow from $(a, b) \leq \max(a, b)$. $(a, b) \leq max(a, b)$ follows from that if $d \mid a$, then $d \leq a$.

For the fourth equation, let $d = (a, b)$. Then $cd \mid ac$ and $cd \mid bc$, so $cd \mid (ac, bc)$ so $(ac, bc) = cdn$ for some n. But if $cdn \mid ac$ and $cdn \mid bc$ we have $dn \mid a$ and $dn \mid b$ so $dn \mid (a, b)$ meaning $n = 1$, and $(ac, bc) = cd = c \cdot (a, b)$.

The fifth follows directly from the transitivity of divisibility.

The set of divisors of ac is exactly the set of $a'c'$ where $a' \mid a$ and $c' \mid c$. The greatest divisor

17.25 First, divide the equation by 4 to get $6x + 11y = 1$. Let $[x, y] = 6x + 11y$. Then

$$(6, 11) = ([1, 0], [0, 1]) = (6, 11 \bmod 6) =$$

$$(6, 11 - 1 \cdot 6) = ([1, 0], [0, 1] - [1, 0]) =$$

$$(6, 5) = ([1, 0], [-1, 1]) = (6 \bmod 5, 5) =$$

$$(6 - 5, 5) = ([1, 0] - [-1, 1], [-1, 1]) =$$

$$(1, 5) = ([2, -1], [-1, 1]) = (1, 5 \bmod 1) =$$

$$(1, 5 - 5) = ([2, -1], [-1, 1] - 5[2, -1]) =$$

$$(1, 0) = ([2, -1], [-11, 6])$$

so $x = 2$ and $y = -1$ gives a solution.

17.26 Let x_1, y_1 be a solution to $ax + by = 1$. Then cx_1, cy_1 is a solution to $ax + by = c$. We can use the same argument as for $c = (a, b)$ to show that any solution must be of the form $(cx_1 + kb, cy_1 - ka)$.

17.27 Clearly we must have that $(a, b)|c$. Otherwise the LHS would have a divisor that the RHS does not. We can then divide the equation by (a, b) so that $(a, b) = 1$. Then the equation is also solvable, since we can find x, y such that $ax + by = 1$.

17.29 It's equivalent to $n \mid a$.

17.30 Write s as a decimal number on the form $d_n d_{n-1} \dots d_1 d_0$, so that $s = \sum_{i=0} 10^i \cdot d_i$. Since $10 \equiv 1 \pmod 9$, $10^n \equiv 1 \pmod 9$ as well. Then $s = \sum_{i=0} 10^i \cdot d_i \equiv 1 \cdot d_i \pmod 9 = s(d)$.

17.37 For each integer $d \mid n$, let's count the solutions to $d = \gcd(n, i)$. Let $i = dk$. By the GCD laws, we know that $\frac{n}{d}$ is coprime to $\frac{i}{d} = k$. Furthermore, since $1 \le dk \le n$ we have $1 \le k \le \frac{n}{d}$. But this is exactly what $\phi(\frac{n}{d})$ counts. Thus, each integer $1 \le i \le n$ is counted once over all the d, so that $n = \sum_{d|n} \phi(\frac{n}{d})$. By symmetry,

$$\sum_{d|n} \phi(\frac{n}{d}) = \sum_{d|n} \phi(d)$$

which proves the statement.

Chapter 18

18.1 Each element can be in either A, B, or neither of the sets. The choice for each element is independent, so by the multiplication principle there are 3^N such pairs.

18.7 Each placement is a permutation of the people, so there's 8! ways.

A cyclically unique arrangement can be rotated in 8 ways, so there must be $\frac{8!}{8} = 7!$ such arrangements.

18.11 Assume that $l \nmid \operatorname{ord}\pi$. Let $\operatorname{ord}\pi = ql + r$ with $0 < r < l$. If a is on the cycle, we know that $\pi^{ql+r}(a) = \pi^r(a)$ since $\pi^l(a) = a$. By the definition of the cycle decomposition, l is the smallest positive integer for which $\pi^l(a) = a$. But $\pi^r(a) = a$ and $0 < r < l$, a contradiction, so the assumption that $l \nmid \operatorname{ord}\pi$ (equivalent to $r \neq 0$ was false).

18.12 Let a be an element on the cycle and let $m = \frac{l}{\gcd(l,K)}$. Note that $\pi^n(a) = \pi^m(a)$ if and only if $n \equiv m \pmod{l}$, so if $(\pi^K)^i(a) = \pi^{iK}(a) = a$ then $iK \equiv 0 \pmod{l}$. This is equivalent to $l \mid iK$, which also means

$$\frac{l}{\gcd(l,K)} \mid i \cdot \frac{K}{\gcd(l,K)}.$$

By definition $\frac{l}{\gcd(l,K)}$ and $\frac{K}{\gcd(l,K)}$ are coprime, so by the cancellation law

$$\frac{l}{\gcd(l,K)} \mid i.$$

The smallest positive such i is $\frac{l}{\gcd(l,K)}$, so this is also the order of a (and its cycle length) in π^K.

Since this applies to every element in cycle of length l, we have $\frac{l}{\frac{l}{\gcd(l,K)}} = \gcd(l,K)$ such cycles.

18.14 We want to prove

$$\binom{n}{k}k = n\binom{n-1}{k-1}.$$

We claim that both sides count the number of ways to choose a k-person committee, of which one person is chair. The LHS counts the number of such committees, multiplied by the number of choices for who should be the chair. The RHS counts the number of possible chairs, and then selects the remaining $k-1$ people in the committee.

18.15

1. No, they are exactly the same with regards to overflows. Note that both of them, before dividing by the final k, need to represent $k\binom{n}{k}$. This is also the greatest intermediate result that must be stored.
2. The binomial coefficients are $\binom{n}{0}, \binom{n}{1}, \ldots$. The identity we get is

$$\binom{n}{i}\frac{n-i}{i+1} = \binom{n}{i+1}$$

so we want to prove that

$$\binom{n}{i}(n-i) = \binom{n}{i+1}(i+1).$$

This represent two ways to choose a committee with $(i+1)$ people and a chair. The RHS we saw in the previous exercise, but the LHS is different. First we choose the regular i members of the committee, and then among the $n-i$ people left, we choose the chair.

18.17 Both sides count k-subsets of an n-set. The last of the n elements either is in that subset or not. In the first case, $k-1$ elements must be chosen among the remaining $n-1$ elements. In the second case, k elements must be chosen among them instead. These two cases sum to the RHS.

18.18

1. The LHS counts all the subsets of an n-set, one size at a time.
2. The sum adds the number of even-sized subsets of an n-set S, and removes the number of odd-sized. Pick any element a from the n-set. Now, pair up each subset $T \subseteq S$ where $a \notin T$ with $T \cup \{a\}$. Each pair contains exactly one even-sized and one odd-sized subset, so the number of each type is the same.
3. The LHS counts the number of subsets B of all subsets A of an n-set S. Since $B \subseteq A \subseteq B$, each element has three choices. Either it's in S but not A, or it's in A but not B, or it's in B.
4. The RHS counts $(k+1)$-subsets of an $(n+1)$-set. The LHS does the same. Let $i+1$ be the index of the last element chosen in the subset. Then, there are $\binom{i}{k}$ ways to choose the remaining k. The sum is over all valid choices of i.

18.19 It's not hard to prove combinatorially that the central binomial coefficients are also the greatest. Since the sum of all $n+1$ binomial coefficients $\binom{n}{k}$ sum to 2^n, the greatest must be at least $\frac{2^n}{n+1} = \Theta(\frac{2^n}{n})$.

18.20 The formula for the binomial coefficients gives us that

$$\binom{n}{k} = \frac{1}{k!}(n-k+1)(n-k+2)\cdots n.$$

By substituting $n = k + i$, this equals

$$\binom{n}{k} = \frac{1}{k!}(i+1)(i+2)\cdots(i+k)$$

which is a polynomial in i of degree k, so $\binom{n}{k} = \Theta(n^k)$ since $i = n - k = \Theta(n)$ for fixed k.

18.22 The RHS counts the number of Dyck paths in an $n \times n$ Dyck path. Note that a Dyck path in that grid always crosses the top-left to bottom-right diagonal after n steps. The

path can be split up into two parts: one from the start to the diagonal crossing, and one from the crossing to the end. Assume that the crossing is in the i'th column. The first part is then a Dyck path in a $i \times (n-i)$ grid, the and second part a path in a $(n-i) \times i$ grid. There are $\binom{n}{i}\binom{n}{n-i}$ such paths. Now we take the sum over all i to get the LHS.

18.25 Each of the N rectangles must have as its top-right corner one of the steps of the staircase. Consider the topmost rectangle in the first column. If this has height k, none of the topmost rectangles in the next $k-1$ columns can extend further down than the first rectangle, or else we'd form a rectangle with a top-right corner that's not one of the steps of the staircase. This means that the brackets are nested correctly, since each bracket pair includes within it fewer bracket pairs than the bracket pair that it's nested in itself.

Now consider a valid bracket sequence. Assume that the first bracket pair consists of $k-1$ nested brackets, so that the height of the first rectangle is k. We have two other valid bracket sequences: the $k-1$ brackets inside the first, as well as the $N-k$ that comes after it. By recursively constructing a staircases for these two brackets and placing them at the right place, there's only one issue – there's nothing that fills the $N-k$ squares *below* the topmost rectangle in the first column. To fix this, extend all the rectangles in the first column of the sub-staircase immediately beneath it starting in $(k+1)$'th column k steps to the left.

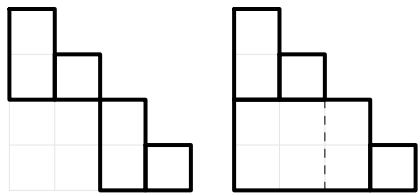


Figure B.1: The combination of the bracket sequences (()) and (()) into (())(()).

This can be done because that staircase was constructed independently of everything else as the staircase of the last $N-k$ bracket pairs, so it has no rectangle extending into the rectangle already in the first column.

18.27 Let A, B and C be the sets of the integers divisible by 2, 3 and 5. We seek

$$|A \cup B \cup C| = |A| + |B| + |C| - |A \cap B| - |A \cap C| - |B \cap C| + |A \cap B \cap C|.$$

We have $|A| = \left\lfloor \frac{1000}{2} \right\rfloor = 500$, $|B| = \left\lfloor \frac{1000}{3} \right\rfloor = 333$, $|C| = \left\lfloor \frac{1000}{5} \right\rfloor = 200$. Since 2, 3 and 5 are relatively prime, the intersections of these sets are also easy to compute: $|A \cap B| = \left\lfloor \frac{1000}{6} \right\rfloor = 166$, $|A \cap C| = \left\lfloor \frac{1000}{10} \right\rfloor = 100$, $|B \cap C| = \left\lfloor \frac{1000}{15} \right\rfloor = 66$, and finally $|A \cap B \cap C| = \left\lfloor \frac{1000}{30} \right\rfloor = 33$. The answer is then $500 + 333 + 200 - 166 - 100 - 66 + 33 = 734$.

18.28 Assume that an element is present in k of the elements. It's then counted $\binom{k}{1}$ times in the first sum, $\binom{k}{2}$ times in the second, $\binom{k}{3}$ in the third and so on. In total, it's counted

$\sum_{i=1}^{k}(-1)^{i+1}\binom{k}{i}$ times. From a previous exercise we know that $\sum_{i=0}^{k}(-1)^{i}\binom{k}{i} = 0$. But

$$\sum_{i=1}^{k}(-1)^{i+1}\binom{k}{i} = -\sum_{i=0}^{k}(-1)^{i}\binom{k}{i} + \binom{k}{0} = 0 + 1$$

so each element is counted exactly once.

18.31 Assume that no stack has more coins than the other stacks combined. Then there are two distinct stacks with coins, so a move is possible (unless there are no more coins). Remove one coin from the two largest stacks. If the largest stack is still the largest, can't have more coins than the other stacks since one was removed from each. If another stack became the largest, it has at most one more coin than previously largest stack. If it has more coins than all the other coins, there must not be any non-empty third stack, but this is not possible since the total number of coins would then be odd. Thus, the invariant holds after each move, and a move is always possible unless there are no remaining coins.

18.32 Let $i < j$ be the positions of the swapped elements. Any inversion not including i or j is unchanged, and any inversion including them but where the other element is to the left of i or to the right of j is also unchanged.

Consider an element at a position $i < k < j$. Either (i, k) is an inversion or not. In the first case an inversion is removed, while in the second case an inversion is added. The same applies for (k, j). No matter what the case is, the number of inversions changes with only −2, 0 or 2, leaving the parity unchanged.

Finally, we have that (i, j) itself either is or isn't an inversion. Swapping them thus either removes or adds one inversion, changing the parity of the permutation.

18.33 We saw one approach in Chapter 12 using divide and conquer. The one that's used in practice is the following. For each element, you want to compute the number of elements to the left of it that are also greater than the element. Go through the permutation one element at a time. Keep a segment tree T where $T[x] = 1$ if you have seen the element x so far. When you see x, add $\sum_{i=x+1}^{N} T[i]$ to the number of inversions and add one to $T[x]$.

18.34 You already know that each 3 × 3 subgrid can be transformed to any subgrid, except that two elements might be swapped. In particular, this means that the first either 2 rows or 2 columns can be transformed into anything. This allows you to first iteratively solve all rows except the bottom one. Then, use the same method on the bottom 3 columns left-to-right, which leaves you only with a scrambled 3 × 3 subgrid in the bottom-right corner. Assuming the parity of the overall grid permutations was correct, that subgrid can also be sorted.

18.35 A cycle of length l can be transformed to the identity permutation with $l - 1$ swaps. The number of swaps is then even if and only if it has an even number of even-length cycles, but the parity of number of swaps is also that of the permutation.

18.36 Swapping the first two elements of the permutation gives you a bijection between odd and even permutations.

Chapter 19

Bibliography

[1] Noga Alon, Raphy Yuster, and Uri Zwick. Color-coding: A new method for finding simple paths, cycles and other small subgraphs within large graphs. In *Proceedings of the Twenty-Sixth Annual ACM Symposium on Theory of Computing*, STOC '94, page 326–335, New York, NY, USA, 1994. Association for Computing Machinery.

[2] Sanjeev Arora and Boaz Barak. *Computational Complexity: A Modern Approach*. Cambridge University Press, 2009.

[3] Bengt Aspvall, Michael F. Plass, and Robert Endre Tarjan. A linear-time algorithm for testing the truth of certain quantified boolean formulas. *Information Processing Letters*, 8(3):121–123, 1979.

[4] David Beazley and Brian K. Jones. *Python Cookbook*. O'Reilly, 2013.

[5] Richard Bellman. On a routing problem. *Quarterly of Applied Mathematics*, 16(1):87–90, 1958.

[6] Elywn R. Berlekamp, John H. Conway, and Richard K. Guy. *Winning Ways for Your Mathematical Plays: Volume 1*. Number v. 1. CRC Press, 2018.

[7] Joshua Bloch. *Effective Java*. Pearson Education, 2008.

[8] Robert S. Boyer and Strother J. Moore. *MJRTY—A Fast Majority Vote Algorithm*, pages 105–117. Springer Netherlands, Dordrecht, 1991.

[9] Xuan Cai. Canonical coin systems for change-making problems. In *2009 Ninth International Conference on Hybrid Intelligent Systems*, volume 1, pages 499–504, Aug 2009.

[10] John H. Conway. *On Numbers and Games*. Ak Peters Series. Taylor & Francis, 2000.

[11] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. *Introduction to Algorithms*. The MIT Press, 3rd edition, 2009.

[12] Marek Cygan, Fedor V. Fomin, Łukasz Kowalik, Daniel Lokshtanov, Dániel Marx, Marcin Pilipczuk, Michał Pilipczuk, and Saket Saurabh. *Parameterized Algorithms*. Springer, 2015.

[13] Reinhard Diestel. *Graph Theory*. Springer, 2016.

[14] Edsger W. Dijkstra. A note on two problems in connexion with graphs. *Numer. Math.*, 1(1):269–271, dec 1959.

[15] Jack Edmonds and Richard M. Karp. Theoretical improvements in algorithmic efficiency for network flow problems. *Journal of the ACM*, 19(2):248 – 264, 1972.

[16] Philippe Flajolet and Robert Sedgewick. *Analytic Combinatorics*. Cambridge University Press, 2009.

[17] Philippe Flajolet and Robert Sedgewick. *An Introduction to the Analysis of Algorithms*. Addison-Wesley, 2013.

[18] Robert W. Floyd. Algorithm 97: Shortest path. *Communications of the ACM*, 5(6):345 – N–PAG, 1962.

[19] Fedor V. Fomin and Dieter Kratsch. *Exact Exponential Algorithms*. Springer, 2010.

[20] Ralph P. Grimaldi. *Discrete and Combinatorial Mathematics: An Applied Introduction*. Pearson Education, 2003.

[21] Patrick M. Grundy. Mathematics and games. *Eureka*, 2:6–8, 1939.

[22] Godfrey H. Hardy and Edward M. Wright. *An Introduction to the Theory of Numbers*. Oxford University Press, 2008.

[23] C. Antony R. Hoare. Algorithm 65: Find. *Communications of the ACM*, 4(7):321–322, jul 1961.

[24] Michael Huth. *Logic in Computer Science*. Cambridge University Press, 2004.

[25] Donald B. Johnson. A note on dijkstra's shortest path algorithm. *Journal of the ACM*, 20(3):385 – 388, 1973.

[26] Ellis L. Johnson. On shortest paths and sorting. In *Proceedings of the ACM Annual Conference - Volume 1*, ACM '72, page 510–517, New York, NY, USA, 1972. Association for Computing Machinery.

[27] Anatoly Karatsuba and Yuri Ofman. Multiplication of Multidigit Numbers on Automata. *Soviet Physics Doklady*, 7:595, January 1963.

[28] Donald E. Knuth. *The Art of Computer Programming, Volume 4A: Combinatorial Algorithms, Part 1*. Number del 1. Pearson Education, 2014.

[29] Donald E. Knuth, Oren Patashnik, and Ronald Graham. *Concrete Mathematics: A Foundation for Computer Science*. Addison-Wesley, 1994.

[30] Lázló Lovász. *Combinatorial Problems and Exercises*. AMS/Chelsea publication. North-Holland Publishing Company, 1993.

[31] George S. Lueker. Two NP-complete problems in nonnegative integer programming. 1975.

[32] Robert C. Martin. *Clean Code: A Handbook of Agile Software Craftsmanship*. Pearson Education, 2009.

[33] Steve McConnell. *Code Complete: A Practical Handbook of Software Construction*. Microsoft Press, 2004.

[34] Scott Meyers. *Effective STL*. O'Reilly, 2001.

[35] Scott Meyers. *Effective C++*. O'Reilly, 2005.

[36] Scott Meyers. *Effective Modern C++*. O'Reilly, 2014.

[37] Gary L. Miller. Riemann's hypothesis and tests for primality. In *Proceedings of the Seventh Annual ACM Symposium on Theory of Computing*, STOC '75, page 234–239, New York, NY, USA, 1975. Association for Computing Machinery.

[38] J. Misra and David Gries. Finding repeated elements. *Science of Computer Programming*, 2(2):143–152, 1982.

[39] Christos Papadimitriou. *Computational Complexity*. Addison-Wesley, 1994.

[40] Charles Petzold. *CODE*. Microsoft Press, 2000.

[41] Michael O. Rabin. Probabilistic algorithm for testing primality. *Journal of Number Theory*, 12(1):128–138, 1980.

[42] Bernard Roy. Transitivité et connexité. *Comptes Rendus Hebdomadaires Des Seances De L Academie Des Sciences*, 249(2):216–218, 1959.

[43] Stuart Russell and Peter Norvig. *Artificial Intelligence: A Modern Approach*. Always learning. Pearson, 2016.

[44] Alfonso Shimbel. Structure in communication nets. In *Proceedings of the symposium on information networks, New York, April, 1954*, 1955.

[45] Victor Shoup. *A Computational Introduction to Number Theory and Algebra*. Cambridge University Press, 2008.

[46] Brett Slatkin. *Effective Python*. Addison-Wesley, 2015.

[47] Michael Spivak. *Calculus*. Springer, 1994.

[48] Roland Sprague. Über mathematische kampfspiele. *Tohoku Mathematical Journal, First Series*, 41:438–444, 1935.

[49] Richard P. Stanley and Gian-Carlo Rota. *Enumerative Combinatorics: Volume 1*. Cambridge Studies in Advanced Mathematics. Cambridge University Press, 1997.

[50] Volker Strassen. Gaussian elimination is not optimal. *Numerische Mathematik*, 13:354–356, 1969.

[51] Bjarne Stroustrup. *The C++ Programming Language*. Addison-Wesley, 2013.

[52] Robert Endre Tarjan. Depth-first search and linear graph algorithms. *SIAM Journal on Computing*, 1(2):146–160, 1972.

[53] Jeffrey Ullman and John Hopcroft. *Introduction to Automata Theory, Languages, and Computation*. Pearson Education, 2014.

[54] Jacobus Hendricus van Lint and Richard M. Wilson. *A Course in Combinatorics*. Cambridge University Press, 2001.

[55] John von Neumann and Oskar Morgenstern. *Theory of Games and Economic Behavior: 60th Anniversary Commemorative Edition*. Princeton Classic Editions. Princeton University Press, 2007.

[56] Stephen Warshall. A theorem on boolean matrices. *J. ACM*, 9(1):11–12, jan 1962.

[57] Mark A. Weiss. *Data Structures and Algorithm Analysis in C++*. Pearson, 2013.

[58] Herbert S. Wilf. *Generatingfunctionology*. Elsevier Science, 2014.

[59] Mingyu Xiao and Hiorshi Nagamochi. A refined algorithm for maximum independent set in degree-4 graphs. *Journal of Combinatorial Optimization*, 34(3):830–873, Oct 2017.

[60] Mingyu Xiao and Hiroshi Nagamochi. Exact algorithms for maximum independent set. *Information and Computation*, 255:126–146, 2017.

Index