Principles of Algorithmic Problem Solving

Johan Sannemo

2017

This version of the book is a preliminary draft. Expect to find typos and other mistakes. If you do, please report them to johan.sannemo+book@gmail.com. Before reporting a bug, please check whether it still exists in the latest version of this draft, available at http://csc.kth.se/~jsannemo/slask/main.pdf.

Contents

Preface

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 events 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. Instead, much of the content is 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 to give an insight into 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. In this book, a single language is used – C++. The text 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 will teach 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. Additionally, it 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 an extent. As such, we will include more real code and efficient C++ implementations of algorithms than you will see in 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. Finally, thanks to several others, who have read through drafts and caught numerous mistakes of my own.

Reading this Book

This book consists of two parts. The first part contains some preliminary background, such as algorithm analysis and programming in C++. With an undergraduate education in computer science, most of these chapters are 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 makes up most of the material in the book. 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 an algorithms course. Therefore, we recommend that you read through the parts even if you feel familiar with them – in particular those on the basic problem solving paradigms, i.e. brute force, greedy algorithms, dynamic programming and divide & conquer. The chapters in this part are structured so that a chapter builds upon only the preliminaries and previous chapters to the largest extent possible.

At the end of the book, you can find a appendix with some mathematical background.

This book can also be used to improve your competitive programming skills. Some parts are unique to competitive programming (in particular Chapter 15 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 the competitions.

The book often refers to exercises from the Kattis online judge. These are named *Kattis Exercise*, and give a problem name and ID.

Exercise 0.1 — Kattis Exercise

Problem Name – problemid

The URL of such a problem is `http://open.kattis.com/problem/problemid`.

The C++ code in this book makes use of some preprocessor directives from a template. Even if you are familiar with C++, or does not wish to learn it, we still recommend that you read through this template (Section 2.13) to better understand the C++ code in the book.

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 which 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. writing 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.

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 string", or some other kind of mathematical object. Using this input, we have a goal which we wish 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

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

to the highest.

Input
The input consists of a sequence of N integers $a_0, a_1, ..., a_{N-1}$.

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

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

We will see some variations of this format later, such as problems without inputs, but in general this is what our problems will look like.

Competitive Tip

Sometimes, problem statements contain huge amounts of text. Skimming through the input and output sections before any other text in the problem can often give you a quick idea about its topic and difficulty, which helps determining what problems to solve first.

Exercise 1.1

What is the input and output for the following computational problems?

1) Compute the greatest common divisor of two numbers.

2) Find a root of a polynomial.

3) Multiply two numbers.

Exercise 1.2

Consider the following problem. I am thinking of an integer between 1 and 100. Your task is to find this number, by asking me questions of the form "is your number higher, lower or equal to x" for different numbers 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 the solutions to computational problems. They define a method which uses the input to the problem to produce the correct output. A computational problem can have many solutions. Algorithms to solve the sorting problem are a research area by themselves! Let us look at one possible sorting algorithm as an example, called selection sort.

Algorithm 1.1: Selection Sort

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

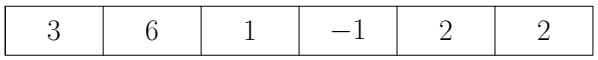
Assume that we have chosen the K smallest elements of the original sequence, and have sorted them. Then, the smallest element remaining in that sequence must be the $(K+1)$'st smallest element of the original sequence. Thus, by finding the smallest element among those that remain, we know what the $(K+1)$'st element of the sorted sequence is. Combining this with the first K sorted elements, we can find the first $K+1$ elements of the output.

By repeating this process N times, the result will be the N numbers of the original sequence, but sorted.

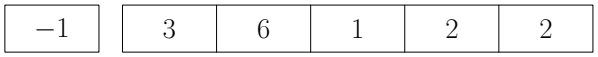
If you want to see this algorithm in practice, it is performed on our previous example instance, the sequence $3, 6, 1, -1, 2, 2$, in Figures 1.1a-1.1f.

So far, we have been vague about what exactly an algorithm is. Looking at our example (Algorithm 1.1) 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 the ambiguity and the lack of detail.

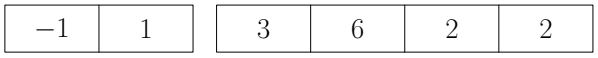
Until an algorithm is described in sufficient detail, it is possible to accidentally abstract away operations we may not know how to perform behind a few English words. As a somewhat contrived example, our textual description of selection sort includes actions such as "choosing the smallest number of 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 thus require us to formulate our algorithm in steps small enough that even a computer knows how to perform it. In this sense, a computer is rather stupid.



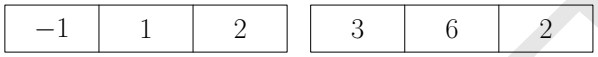
(a) Originally, we start out with the unsorted sequence `3 6 1 -1 2 2`.



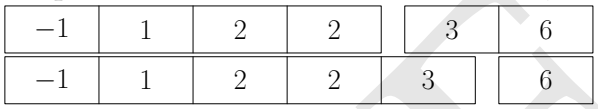
(b) The smallest element of the sequence is -1, so this is the first element of the sorted sequence.



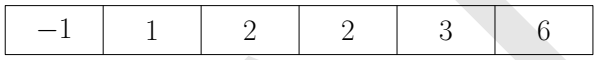
(c) We find the next element of the output by removing the -1, and finding the smallest remaining element – in this case 1.



(d) Here, there is no unique smallest element. In this case, we choose any of the two 2s.



(e) The next two elements chosen will be a 2 and a 3.



(f) Finally, we choose the last remaining element of the input sequence – the 6. This concludes the sorting of our sequence.

Figure 1.1: An example execution of Selection Sort.

The English language is also ambiguous. We are sloppy with references to "this variable" and "that set". 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.

We will generally describe our algorithms in a representation called pseudo code (Section 1.4), accompanied by an online exercise to implement the code. Sometimes, we will instead give explicit code that solves a problem. This will be 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 1.3

Do you know any algorithms, for example from school? (Hint: you use many algorithms to solve certain arithmetic and algebraic problems, such as those in Exercise 1.1.)

Exercise 1.4

Construct an algorithm which solves the guessing problem in exercise 1.2. How many questions does it use? The optimal number of questions is about $\log_2 100 \approx 7$ questions. Can you achieve this?

1.2.1 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***. The first notion requires an algorithm to, upon termination, have produced an output that fulfill all the criteria laid out in the output description. Total correctness additionally require an algorithm to terminate within finite time. When we talk about correctness of our algorithms later on, we will generally focus on the partial correctness. Termination will instead be proved implicitly, as we will consider more granular measures of efficiency (called *time complexity*, Chapter 4) than just finite termination. These measures will imply the termination of the algorithm, completing the proof of total correctness.

Proving that the selection sort algorithm terminates 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 considering 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 formally prove. The main idea behind a formal proof is contained within our description of the algorithm itself.

Later on, we will compromise on both conditions. 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 will become 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. Unfortunately, this makes it even harder to decide if an incorrect submission is due to an incorrect algorithm or an

incorrect implementation.

Exercise 1.5

Prove the correctness of your algorithm to the guessing problem from Exercise 1.4.

Exercise 1.6

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 where a computer can execute them. While we in textual descriptions of methods are often satisfied with describing *what* we wish to do, programming languages 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 are often a bit arcane at first, but they grow 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, with current above or below a certain threshold denoting the digits 0 and 1. Above these circuit-level electronics lie a processors 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 what we call *compilers*. When programming in C++, we can not immediately tell a computer to run its code.

As you now know, C++ is 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 machine code, which a processor knows how to handle. It is a program of its own, that takes the code files we will write and produce *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++.

1.4 Pseudo Code

Somewhere in between describing algorithms in English text and in a programming language lie ***pseudo code***. As hinted by its name, it is not quite actual code, in two aspects. First of all, the instructions we write is not the programming language of any particular computer. The point of pseudo code is to be independent of what computer it is implemented on. Instead, it tries to convey the main points of an algorithm in a detailed manner such that it can easily be translated into any particular programming language. Secondly, we sometimes fall back to the liberties of the English language. At some point, we may decide that "choose the smallest number of a sequence" is clear enough for our audience.

With an explanation of this distinction in hand, let us look at a concrete example of what is meant by pseudo code. Again, the honor of being an example falls upon selection sort, now described in pseudo code in Algorithm 1.2.

Algorithm 1.2: Selection Sort

```
procedure SELECTIONSORT(A)
    Let A′ be an empty sequence
    while A is not empty do
        minIndex ← 0
        for every element A_i in A do
            if A_i < A_minIndex then
                minIndex ← i
        Append A_minIndex to A′
        Remove A_minIndex from A
```

Pseudo code reads somewhat like our English language variant of the algorithm, except having the actions broken down into smaller pieces. Most of the constructs of our pseudo code are more or less obvious. The notation *variable* $\leftarrow$ `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`.

Pseudo code will appear when we try to explain some part of a solution in great detail, but where programming language specific aspects would draw unnecessary attention to themselves.

Competitive Tip

In team competitions where a team only have a single computer, a team 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 1.7

Write pseudo code for your algorithm to the guessing problem from Exercise 1.4.

1.5 The Kattis Online Judge

Most of the exercises in this book exist as problems on the ***Kattis*** web system. You can find the judge at `https://open.kattis.com`. Kattis is a so called ***online judge***, which has been used in the university competitive programming world finals (the International Collegiate Programming Contest World Finals) for several years. It contains a large collection of computational problems, and allows you to submit a program you have written that purports to solve the problem. Kattis will then run your program on a large number of predetermined instances of the problem.

When solving problems on a judge, a problem will generally include some additional information. Since actual computers only have a finite amount of time and memory, problems limit the amount of these resources available to our programs when solving an instance. This also means the size of inputs to a problem need to be constrained as well, or else the resource limits for a given problem would not be obtainable – an arbitrarily large input generally takes arbitrarily large time to process, even for a computer. Thus, a more complete version of the sorting problem as given in a competition could look like this:

Sorting

Time: 1s, memory: 1MB

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

Input

The input consists of a sequence of N integers ($0 \leq N \leq 1000$) $a_0, a_1, ..., a_{N-1}$ ($|a_i| \leq 10^9$).

Output

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

If your program exceeds the allowed resource limits (i.e. takes too much time or memory), crashes, or gives an invalid output, Kattis will tell you so with a *rejected* judgment. Assuming your program passes all the instances, it will be be given the *Accepted* judgment.

Note that getting a program accepted by Kattis is not the same as having a correct program – it is a necessary but not sufficient criteria for correctness. This is also a fact which is possible to exploit during competitions, which we will see later in this book.

We recommend that you get an account on Kattis, so that you can follow along with the book's exercises.

Exercise 1.8

Register an account on Kattis

Many other online judges exists, such as:

- Codeforces (`http://codeforces.com`)
- TopCoder (`https://topcoder.com`)
- HackerRank (`https://hackerrank.com`)

1.6 Chapter Notes

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

Many other books delve into the theoretical study of algorithms deeper than we will, in particular regarding subjects not relevant to algorithmic problem solving. *Introduction to Algorithms* [5] is a rigorous introductory text book on algorithms, with both depth and width.

For a gentle introduction to the technology that underlies computers, *CODE* [19] is a well-written journey from the basics of bits and bytes all the way to assembly code and operating systems.

Chapter 2

Programming in C++

We will learn some more practical matters – the basics of the C++ programming language. This language 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 most other competitive programming languages, and there are many routines in the accompanying standard code libraries that are useful when implementing algorithms.

Of course, no language is without downsides. C++ is a bit difficult to learn as a beginner's 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 become 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 the benefits of C++ weigh up for the disadvantages in the long term even though it demands more from you as a reader. 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. Additionally, we recommend that you do **every** exercise in this chapter.

Finally, know that our treatment of C++ is minimal. We will not explain all the details behind the language, nor good coding style or general software engineering principles. In fact, we will frequently make use of bad coding practices. If you want

to delve deeper, you can find more resources in the chapter notes.

2.1 Development Environments

Before we get to the juicy parts of C++, you first need to install a compiler for C++ and (optionally) a code editor. If you are running Windows, we recommend *Code::Blocks*, a code editor that also installs a compiler for you. You can download this from the Code::Blocks web site, `http://www.codeblocks.org/downloads/26`. Choose the file that ends with `mingw-setup.exe`.

If you are using Mac OS, you can instead get a compiler by installing *Xcode* from the Mac App Store. You can then either use Xcode to write your programs, or install Code::Blocks.

Running a Linux distribution (such as Ubuntu), you should be able to find the package `g++` in your favorite package manager. This installs the GCC compiler for you, which is the most popular compiler for Linux systems. Possibly, `codeblocks` exists as a package there as well.

Note that instructions like these tend to rot, with applications disappearing from the web, operating systems changing names and so on. In this case, you are on your own, and will have to find instructions to by yourself.

In this chapter, we will assume you are using Code::Blocks. If you chose some other editor or compiler, you will need to find instructions on how to compile and run programs yourself.

2.2 Hello World!

Now that you have a compiler and editor ready, we will learn the basic structure of a C++ program. The classical example of a program when learning a new language is to print the text `Hello World!`. We will also solve our first Kattis problem in this section.

Start by opening Code::Blocks, and create a new file (by going to File ⇒ New ⇒ Empty File). Save the file as `hello.cpp` in a new folder (such as `Code`).

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

In Code::Blocks, You can save your program by typing Ctrl+S, and running it by pressing the F9 key. A window will appear, containing the text `Hello World!`. If no window appear, you probably mistyped the program.

Listing 2.1 Hello World!

```
#include <iostream>

using namespace std;

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

Coincidentally, Kattis happens to have a 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 Kattis.

Exercise 2.1 — Kattis Exercise

Hello World! – hello

When you submit your solution, Kattis will grade it and give you her judgment. If you typed everything correctly, Kattis will tell you it got Accepted. Otherwise, you will probably get Wrong Answer, meaning you typed the wrong text.

Once you have managed to solve the problem, it is time to talk a bit about the code you typed.

The first line of the code,

```
#include <iostream>
```

is used to include the `iostream` 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 in particular contains routines for reading and writing data to your screen. Your program used code from this file when it printed `Hello World!` upon execution.

Competitive Tip

On some platforms, there is a special include file called `bits/stdc++.h`. This file basically 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 can use this and not include anything else.

The third line,

```
using namespace std;
```

tells the compiler that we wish to use code from the **st**an**d**ard library. If we did not use it, we would have to specify this every time we used code from the standard library later in our program.

The fifth line defines our ***main function***. When we instruct the computer to run our program, this is where it will start looking for code to execute. The first line of the main function is where the program will start to run, and then execute further lines sequentially. We will later see how we can define 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 would not know which lines belonged to the function.

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

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

Comments are explanatory lines which are not executed by the computer. The purpose of a comment is to explain what the code around it does, and why. They begin 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 e.g:

```
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, and are mandatory.

Exercise 2.2

Must the main function be named `main`? What happens if you changed `main` to something else and try to run your program?

Exercise 2.3

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

2.3 Variables and Types

The Hello World! program is boring. It only prints text, which is seldom the only necessary component of an algorithm (aside from the Hello World! problem on Kattis). We now move on to a new but hopefully familiar concept.

When we solve mathematical problems, it often proves useful to introduce all kinds of names for known and unknown values. Math problems often deal with classes of N students, ice cream trucks with velocity v_{car} km/h and candy prices of p_{candy} \$/kg.

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. We say that "let $x = 5$", and that is that. In C++, we need to be a bit more verbose. We must write that "I want to introduce a variable x now. It is going to be an integer – more specifically, 5". Once we have decided what kind of value x will be (in this case integer), it will always be an integer. We cannot just 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).

Another major difference is that variables in C++ are not tied to a single value for the entirety of its lifespan. Instead, we are able to modify the value which our variables take using something called ***assignment***. Some languages does not permit this, preferring their variables to be ***immutable***.

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

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

```
int five = 5;
```

declares an integer variable `five`, and assign 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 5. Finally, we may assign a value to the variable. Note that further use of the variable never include the `int` part. We declare the type of a variable once, and only once.

Listing 2.2 Variables

```
1  #include <iostream>
2
3  using namespace std;
4
5  int main() {
6    int five = 5;
7    cout << five << endl;
8    int seven = 7;
9    cout << seven << endl;
10
11   five = seven + 2; // = 7 + 2 = 9
12   cout << five << endl;
13
14   seven = 0;
15   cout << five << endl; // five is still 9
16   cout << 5 << endl; // we print the integer 5 directly
17 }
```

Later on, 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, this evaluates to $7 + 2 = 9$. Thus `five` will actually be 9, which explains 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 sets the value of the variable on the left hand side to the value of the expression on the right hand side at a particular moment in time, nothing more.

Finally, the snippet demonstrates how to print the values of a variable on the screen – we `cout` it the same way as with text! This also makes the reason for why text needs to be enquoted more clear. Without quotes, we would not be able to distinguish from the text string `"hi"` and the variable `hi`.

Exercise 2.4

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?

Exercise 2.5

What value 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, - denotes subtraction and * 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 2.6

What happens when an integer is divided by another integer? Try running the following code:

```
cout << (5 / 3) << endl;
cout << (15 / 5) << endl;
cout << (2 / 2) << endl;
cout << (7 / 2) << endl;
```

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

The text data type is called `string`. As we have already seen, values of this type must be enclosed with double quotes. If we actually want to include a quote in a string, we type `\"`.

Listing 2.3 Types

```
#include <iostream>

using namespace std;

int main() {
  string text = "Johan said: \"heya!\" ";
  cout << text << endl;

  char letter = '@';
  cout << letter << endl;

  int number = 7;
  cout << number << endl;

  long long largeNumber = 888888888888LL;
  cout << largeNumber << endl;

  double decimalNumber = 513.23;
  cout << decimalNumber << endl;

  bool thisisfalse = false;
  bool thisistrue = true;
  cout << thisistrue << " and " << thisisfalse << endl;
}
```

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 quote is written `'\''`, similarly to how we included double quotes in strings.

Then comes the `int`, which we discussed earlier. The `long long` type, just as the `int` type, contains integers. 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 2.7

Write a program that assigns the minimum and maximum values of an `int` to a `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 when we try 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 longs`!

Next comes the `double` type. This type represents decimal numbers. Note that the decimal sign in C++ is a **dot**, not a comma.

Finally, we will look at a rather special type, the `bool` (short for ***boolean***). This type can only contain one of two different values – it is either `true` or `false`. While this may look useless at a first glance, boolean values will become important later.

Exercise 2.8

If we type `\"` to include a double quote in a string, this means we cannot include a backslash by simply typing `\`. Find out how to include a literal backslash in a string (either by searching the web, or thinking about how we included the different quote characters).

Exercise 2.9

Just like the integer types, a `double` cannot represent arbitrarily large values. Find out what the minimum and maximum values a `double` can store is.

C++ has a construct called the ***typedef***, or type definition. It allows us to give certain types new names. For example, we might alias `long long` with `ll`. Such a typedef statement looks like this:

```
typedef long long ll;
```

After this statement, we can use `ll` just as if it were a long long:

```
ll largeNumber = 888888888888LL
```

2.4 Input and Output

In previous sections, we have 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 which we type on our keyboards into a running program! When we run a program, we may type things in the window that appears. By pressing the Enter key, we allow 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 Listing 2.4 to see how it works.

Listing 2.4 Input

```cpp
#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 2.10

What happens if you type an invalid input, such as your first name instead of your age?

When reading input into a string variable, the program will only read text until the first whitespace. To read an entire line, you can use the `getline` function (Listing 2.5).

Listing 2.5 getline

```cpp
#include <iostream>

using namespace std;

int main() {
  string line;
  cout << "Type some text, and press enter: " << endl;
  getline(cin, line);

  cout << "You typed: " << line << endl;
}
```

Exercise 2.11 — Kattis Exercises

Two-Sum – twosum

Triangle Area – triarea

2.5 Operators

We have already seen examples of what we call ***operators***. Earlier we have used the assignment operator =, and the arithmetic operators `+ - * /`, which stand for addition, subtraction, multiplication and division. They work almost like they do in mathematics, and allows us to create code such as the one in Listing 2.6.

Listing 2.6 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 2.12

Type in Listing 2.6 and test it on a few different values. In particular, test:

- $b = 0$
- Negative values for a and/or b
- Values where the expected result is outside the valid range of an `int`

As you have probably noticed, the division operator of C++ performs so-called *integer division*, meaning it rounds the answer to an integer (towards 0). Hence `7 / 3 = 2`, with remainder 1.

The snippet introduces the ***modulo*** operator, %. It computes the remainder of the first operand, when divided by the second. As an example, `7 % 3 = 1`.

In case we wish the answer to be a decimal number instead of performing integer division, one of the operands must be a `double` (Listing 2.7).

We end this section with some shorthand operators. Check out Listing 2.8 for some examples.

Listing 2.7 Operators

```
int a = 6;
int b = 4;

cout << (a / b) << endl;

double aa = 6.0;
double bb = 4.0;

cout << (aa / bb) << endl;
```

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. Thus, the above statement increase the variable a with 5.

It turns out addition and subtraction with 1 was a fairly common operation. In fact, it was so common additional operators were introduced for this purpose, saving an entire character instead of the highly verbose `+=1` operator. These operators consist of two plus signs or two minus signs. Thus, `a++` increments the variable with 1.

Listing 2.8 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;
```

We will sometimes use the fact that these expressions evaluate to some value, and which value this is depends on whether we put the operator before or after the variable name. If we put ++ before the variable, the value of the expression will be 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 yourself and

analyze the results.

2.6 If Statements

In addition to assignment and arithmetic, a large number of ***comparison*** operators exists. These compare two values, resulting in a `bool` value with the result of the comparison (see Listing 2.9).

Listing 2.9 Comparison Operators

```
a == b // check if a equals b
a != b // check if a and b are different
a > b // check if a is greater than b
a < b // check if a is less than b
a <= b // check if a is less than or equal to b
a >= b // check if a is greater than or equal to b
```

Exercise 2.13

Write a program that reads two integers as input, and prints the result of the different comparison operators from Listing 2.9.

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 major use of bool variables is in conjunction with ***if*** statements (also called ***conditional*** statements). They come from the necessity of only executing certain lines of code **if** (and only if) some certain condition is true. For example, assume we need to check if a number is odd or even. We can do this by computing the remainder of a number 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 Listing 2.10.

Listing 2.10 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`.

In our odd or even example, we can see a certain redundancy. If a number is not even, we already know it is odd. Checking this explicitly using the modulo operator seems to be 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 provides code that should be run if the condition given to the condition of an if statement is `false`. We can thus simplify our odd and even program to the one in Listing 2.11.

Listing 2.11 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 Listing 2.12, implementing a helper for the children's game *FizzBuzz*.

Listing 2.12 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 2.14

Run the program with the values 30, 10, 6, 4. Explain the output you get.

Exercise 2.15 — Kattis Exercises

Three Sorting – threesort

Casino – casino

Grading – grading

2.7 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, such as in the example from Listing 2.13.

Listing 2.13 For

```
#include <iostream>

using namespace std;

int main() {
  int repetitions = 0;
  cin >> repetitions;

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

A for loop is built up from four parts. The first three parts are the semi-colon separated expressions immediately after the `for` keyword. In the first of these parts, you can write some expression, 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 expression.

The first part will be executed only once – it is the first thing that happens in a loop. In this case, we decide to 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 iteration.

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, and is the code that will be executed in each repetition of the loop.

Exercise 2.16

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

Exercise 2.17

Design a loop that instead counts backwards, from *repetitions* − 1 to 0.

Exercise 2.18 — Kattis Exercises

N-sum – nsum

Cinema – cinema

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 en example, consider Listing 2.14.

Listing 2.14 Break and Continue

```
int check = 36;

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 2.19

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;
```

```
12      cout << i << endl;
13  }
```

Exercise 2.20 — Kattis Exercise

Cinema 2 – cinema2

2.8 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 (Listing 2.15).

Listing 2.15 While

```
1  #include <iostream>
2
3  using namespace std;
4
5  int main() {
6    int num = 9;
7    while (num != 1) {
8      if (num % 2 == 0) {
9        num /= 2;
10     } else {
11       num = 3 * num + 1;
12     }
13     cout << num << endl;
14   }
15 }
```

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

2.9 Functions

In mathematics, a ***function*** is something that takes one or more arguments, and compute some value based on them. For example, the squaring function $\text{square}(x) = x^2$, the addition function $\text{add}(x, y) = x + y$ or the minimum function $\min(a, b)$.

Functions exists in programming as well, but work a bit differently. Indeed, we have already seen a function – the `main()` function.

We have implemented the example functions in Listing 2.16.

Listing 2.16 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 proclaiming what data type the 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 (which is a comma-separated list of variable declarations). Finally, we give it a body of code, wrapped in curly brackets.

Unlike our main function, we see the `return` keyword in our functions. 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 2.21**
>
> What will the following function calls evaluate to?
>
> `square(5);`

```
add(square(3), 10);
min(square(10), add(square(9), 23));
```

Exercise 2.22

In our code, we declared all of the new arithmetic functions above our `main` function. Why did we do this? What happens if you move one below the `main` function instead? (Hint: what happens if you try to use a variable before declaring it?)

An important fact of function calling 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 Listing 2.17 for an example).

Listing 2.17 Copying

```
1  void change(int val) {
2    val = 0;
3  }
4
5  int main() {
6    int variable = 100;
7    cout << "Variable is " << variable << endl;
8    change(variable);
9    cout << "Variable is " << variable << endl;
10 }
```

We can also choose to not return *anything*, using 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. It turns out that variables may be declared outside of a function. It is then available to every function in your program. Changes to a global variable by one function will also be seen by other functions (try out Listing 2.18 to see them in action).

Secondly, we may actually change the variables given to us 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 will change the variable used for this argument in the calling function instead. For an example of references, check out Listing 2.19.

Listing 2.18 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 current balance is " << currentMoney << " :(" << endl;
}
```

Listing 2.19 References

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

int main() {
  int variable = 100;
  cout << "Variable is " << variable << endl;
  change(0);
  cout << "Variable is " << variable << endl;
}
```

Exercise 2.23

Why is the function call `change(4)` not valid C++? (hint: what exactly are we changing when we assign to the reference in `func`?)

Exercise 2.24 — Kattis Exercise

Arithmetic Functions – arithmethic

2.10 Structures

We will now turn our attention to some more advanced concepts, starting with ***structures***. Structures are a special kind of data type that can contain member variables – variables inside them – and member functions – 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 particular structure contains two member variables, `x` and `y`, representing the coordinates of a point in 2D Euclidean space.

Once we have defined a structure, we can create ***instances*** of it. Every instance has its own copy of the member variables of the structure.

To create an instance, we use the same syntax as with other variables. We can access the member variables of a structure using the `instance.variable` syntax:

```
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, using ***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 will receive these arguments, to help set up the instance.

Let us add a constructor to our point structure, to more easily create instances:

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

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

This particular constructor lets us pass two arguments when constructing the instance, to set the coordinates correctly. This lets us avoid the two extra statements to set the member variables.

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

We can also define functions inside the structure. These functions are as any other functions, with the addition that they can access the member variables of the instance which the member function is called on. For example, we might want a convenient way to mirror a certain point in the x-axis. This could be accomplished by adding a member function:

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

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

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

To call the member function `mirror()` on the point `p`, we write `p.mirror()`.

Exercise 2.25

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

In our `mirror` function, we are not modifying any of the internal state of the function. We can make this fact clearer by declaring the function to be `const` (similarly to a `const` variable):

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

This change ensures that our function will not be able to change any of the member variables.

Exercise 2.26

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

Exercise 2.27

Write a struct which contains three variables of type `Point` called `Triangle`. It should have an `area()` member function which returns the area of the triangle.

Exercise 2.28

Fill in the remaining code to implement this structure:

```
struct Quotient {
        int nominator;
        int denominator;

        // Construct a new Quotient with the given nominator and denominator
        Quotient(int n, int d) {
                ...
        }

        // Return a new Quotient, this instance plus the "other" instance
        Quotient add(const Quotient &other) const {
                ...
        }

        // Return a new Quotient, this instance times the "other" instance
        Quotient multiply(const Quotient &other) const {
                ...
        }

        // Output the value on the screen in the format n/d
        void print() const {
                ...
        }

};
```

2.11 Arrays

An ***array*** is another special type of variable, which can contain 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. When declaring 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 50 named `seq` would be declared using

```
int seq[50];
```

This creates 50 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[49]`.

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.

Later on, we will transition from using arrays to using a structure from the standard library which serve the same purpose – the *vector*.

> **Exercise 2.29 — Kattis Exercise**
>
> *Reversing Strings* – reverse

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. For example, `#include` is a preprocessor directive that includes a certain file in your code.

Most commonly, we will 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 because we are going to use them in code in the book, so it is a good idea to at least be familiar with their meaning.

2.13 Template

In competitive programming, one often use a *template*, with some shorthand typedef's and preprocessor directives. Here, we given an example of such a template, which will be used in some of the C++ code in this book.

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

#define rep(i, a, b) for(int i = a; i < (b); ++i)
#define trav(a, x) for(auto& a : x)
#define all(x) x.begin(), x.end()
#define sz(x) (int)(x).size()
typedef long long ll;
typedef pair<int, int> pii;
typedef vector<int> vi;

int main() {
}
```

The `trav(a, x)` macro is used to iterate through all members of a data structure from the standard library. We will study such data structures in later chapters.

2.14 Additional Exercises

Exercise 2.30 — Kattis Exercises

Solving for Carrots – carrots

Paul Eigon – pauleigon

Dice Game – dicegame

Reverse Binary – reversebinary

2.15 Chapter Notes

C++ was invented by Danish computer scientist Bjarne Stroustrup. Bjarne has also published a book on the language, *The C++ Programming Language*[23] which 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. They 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:

- `http://en.cppreference.com/w/`
- `http://www.cplusplus.com/reference/`

[1] `https://isocpp.org/`

Chapter 3

Implementation Problems

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

The Recipe

Swedish Olympiad in Informatics 2011, School Qualifiers

You have decided to cook some food. The dish you wish to make requires N different ingredients. For every ingredient, you know the amount you have at home, how much you need for the dish, and how much it costs to buy (per unit).

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.

Input

The first line of input is an integer $N \leq 10$, the number of ingredients in the dish.

The next N lines contain the information about the ingredients, one per line. An ingredient is given by three space-separated integers $0 \leq h, n, c \leq 200$ – the amount you have, the amount you need, and the cost per unit for this ingredient.

Output

Output a single integer – the cost for purchasing the remaining ingredients needed to make the dish.

This problem is not particularly hard. For every ingredient, we first calculate the amount which we need to purchase. The only gotcha in the problem is the mistake of calculating 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.

Algorithm 3.1: The Recipe

```
procedure RECIPE(N, has h, needs n, costs c)
    ans ← 0
    for i ← 0 to N − 1 do
        ans ← ans + max(0, n_i − h_i) · c_i
    return ans
```

Generally, the implementation problems are the easiest type of problems in a contest. They do not require much algorithmic knowledge, so more teams are able to solve them. However, not every implementation problem is easy to code. Just because implementation problems are usually easy to spot, understand and formulate a solution to, you should not underestimate the difficulty coding them. Implementation problems are usually failed 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 simply because more lines of code tend to include more bugs.

Exercise 3.1 — Kattis Exercise

The Recipe – recipe

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

Game Rank

Nordic Collegiate Programming Contest 2016

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. At the contest where it was used, it was the second most solved problem, but 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 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 will 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();
  }
  --curStars;
}

void addLoss() {
  conseqWins = 0;
  if (curRank <= 20) loseStar();
}

int main() {
  string seq;
```

```
49    cin >> seq;
50    for (char res : seq) {
51      assert(1 <= curRank && curRank <= 25);
52      if (res == 'W') addWin();
53      else addLoss();
54      if (curRank == 0) break;
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`. This is helpful when solving problems, since it allows us to verify that assertions we make regarding the internal state of the program indeed holds. In fact, when the above solution was written, the assertions in it actually managed to catch some bugs before submitting the problem!

Exercise 3.2 — Kattis Exercise

Game Rank – gamerank

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

Mate in One

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[1].
- 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.

```
#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},
```

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

```
34    {-2, -1}
35  };
36
37  void display(int x, int y) {
38    printf("%c%d", y + 'a', 7 - x + 1);
39  }
40
41  vii next(int x, int y) {
42    vii res;
43
44    if (board[x][y] == 'P' || board[x][y] == 'p') {
45      // pawn
46
47      int dx = is_white(x, y) ? -1 : 1;
48
49      if (is_valid(x + dx, y) && iz_empty(x + dx, y)) {
50        res.push_back(ii(x + dx, y));
51      }
52
53      if (is_valid(x + dx, y - 1) && is_white(x, y) != is_white(x + dx, y - 1)) {
54        res.push_back(ii(x + dx, y - 1));
55      }
56
57      if (is_valid(x + dx, y + 1) && is_white(x, y) != is_white(x + dx, y + 1)) {
58        res.push_back(ii(x + dx, y + 1));
59      }
60
61    } else if (board[x][y] == 'N' || board[x][y] == 'n') {
62      // knight
63
64      for (int i = 0; i < 8; i++) {
65        int nx = x + rook[i][0],
66        ny = y + rook[i][1];
67
68        if (is_valid(nx, ny) && (iz_empty(nx, ny) || is_white(x, y) != is_white(nx, ny))) {
69          res.push_back(ii(nx, ny));
70        }
71      }
72
73    } else if (board[x][y] == 'B' || board[x][y] == 'b') {
74      // bishop
75
76      for (int dx = -1; dx <= 1; dx++) {
77        for (int dy = -1; dy <= 1; dy++) {
78          if (dx == 0 && dy == 0)
79            continue;
80
81          if ((dx == 0) != (dy == 0))
82            continue;
83
84          for (int k = 1; ; k++) {
85            int nx = x + dx * k,
```

```
86              ny = y + dy * k;
87
88            if (!is_valid(nx, ny)) {
89              break;
90            }
91
92            if (iz_empty(nx, ny) || is_white(x, y) != is_white(nx, ny)) {
93              res.push_back(ii(nx, ny));
94            }
95
96            if (!iz_empty(nx, ny)) {
97              break;
98            }
99          }
100       }
101     }
102
103   } else if (board[x][y] == 'R' || board[x][y] == 'r') {
104     // rook
105
106     for (int dx = -1; dx <= 1; dx++) {
107       for (int dy = -1; dy <= 1; dy++) {
108         if ((dx == 0) == (dy == 0))
109           continue;
110
111         for (int k = 1; ; k++) {
112           int nx = x + dx * k,
113             ny = y + dy * k;
114
115           if (!is_valid(nx, ny)) {
116             break;
117           }
118
119           if (iz_empty(nx, ny) || is_white(x, y) != is_white(nx, ny)) {
120             res.push_back(ii(nx, ny));
121           }
122
123           if (!iz_empty(nx, ny)) {
124             break;
125           }
126         }
127       }
128     }
129
130   } else if (board[x][y] == 'Q' || board[x][y] == 'q') {
131     // queen
132
133     for (int dx = -1; dx <= 1; dx++) {
134       for (int dy = -1; dy <= 1; dy++) {
135         if (dx == 0 && dy == 0)
136           continue;
137
```

```
138           for (int k = 1; ; k++) {
139             int nx = x + dx * k,
140               ny = y + dy * k;
141
142             if (!is_valid(nx, ny)) {
143               break;
144             }
145
146             if (iz_empty(nx, ny) || is_white(x, y) != is_white(nx, ny)) {
147               res.push_back(ii(nx, ny));
148             }
149
150             if (!iz_empty(nx, ny)) {
151               break;
152             }
153           }
154         }
155       }
156
157
158     } else if (board[x][y] == 'K' || board[x][y] == 'k') {
159       // king
160
161       for (int dx = -1; dx <= 1; dx++) {
162         for (int dy = -1; dy <= 1; dy++) {
163           if (dx == 0 && dy == 0)
164             continue;
165
166           int nx = x + dx,
167             ny = y + dy;
168
169           if (is_valid(nx, ny) && (iz_empty(nx, ny) || is_white(x, y) != is_white(nx, ny))) {
170             res.push_back(ii(nx, ny));
171           }
172         }
173       }
174     } else {
175       assert(false);
176     }
177
178     return res;
179   }
180
181   bool is_mate() {
182
183     bool can_escape = false;
184
185     char new_board[8][8];
186
187     for (int x = 0; !can_escape && x < 8; x++) {
188       for (int y = 0; !can_escape && y < 8; y++) {
189         if (!iz_empty(x, y) && !is_white(x, y)) {
```

```
190
191         vii moves = next(x, y);
192         for (int i = 0; i < size(moves); i++) {
193           for (int j = 0; j < 8; j++)
194             for (int k = 0; k < 8; k++)
195               new_board[j][k] = board[j][k];
196
197           new_board[moves[i].first][moves[i].second] = board[x][y];
198           new_board[x][y] = '.';
199
200           swap(new_board, board);
201
202
203           bool is_killed = false;
204           for (int j = 0; !is_killed && j < 8; j++) {
205             for (int k = 0; !is_killed && k < 8; k++) {
206               if (!iz_empty(j, k) && is_white(j, k)) {
207                 vii nxts = next(j, k);
208
209                 for (int l = 0; l < size(nxts); l++) {
210                   if (board[nxts[l].first][nxts[l].second] == 'k') {
211                     is_killed = true;
212                     break;
213                   }
214                 }
215               }
216             }
217           }
218
219           swap(new_board, board);
220
221           if (!is_killed) {
222             can_escape = true;
223             break;
224           }
225         }
226
227       }
228     }
229   }
230
231   return !can_escape;
232 }
233
234 int main()
235 {
236   for (int i = 0; i < 8; i++) {
237     for (int j = 0; j < 8; j++) {
238       scanf("%c", &board[i][j]);
239     }
240
241     scanf("\n");
```

```
242   }
243
244   char new_board[8][8];
245   for (int x = 0; x < 8; x++) {
246     for (int y = 0; y < 8; y++) {
247       if (!iz_empty(x, y) && is_white(x, y)) {
248
249         vii moves = next(x, y);
250
251         for (int i = 0; i < size(moves); i++) {
252
253           for (int j = 0; j < 8; j++)
254             for (int k = 0; k < 8; k++)
255               new_board[j][k] = board[j][k];
256
257           new_board[moves[i].first][moves[i].second] = board[x][y];
258           new_board[x][y] = '.';
259
260           swap(new_board, board);
261
262
263           if (board[moves[i].first][moves[i].second] == 'P' && moves[i].first == 0) {
264
265             board[moves[i].first][moves[i].second] = 'Q';
266             if (is_mate()) {
267               printf("%c%d%c%d\n", y + 'a', 7 - x + 1,
268                       moves[i].second + 'a', 7 - moves[i].first + 1);
269               return 0;
270             }
271
272             board[moves[i].first][moves[i].second] = 'N';
273             if (is_mate()) {
274               printf("%c%d%c%d\n", y + 'a', 7 - x + 1,
275                       moves[i].second + 'a', 7 - moves[i].first + 1);
276               return 0;
277             }
278
279           } else {
280             if (is_mate()) {
281               printf("%c%d%c%d\n", y + 'a', 7 - x + 1,
282                       moves[i].second + 'a', 7 - moves[i].first + 1);
283               return 0;
284             }
285           }
286
287           swap(new_board, board);
288         }
289       }
290     }
291   }
292
293   assert(false);
```

```
294
295   return 0;
296 }
```

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 will reduce this to about 100 lines.

First of, let us clean up the move generation a bit. Currently, it is implemented as the function `next`, together with some auxillary data (lines 25-179). It is not particularly abstract, with a lot of code duplication. First of, almost all the moves of the pieces can be described 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 this abstraction is done with little code.

```
const vii DIAGONAL = {{-1, 1}, {-1, 1}, {1, -1}, {1, 1}};
const vii CROSS = {{0, -1}, {0, 1}, {-1, 0}, {1, 0}};
const vii ALL_MOVES = {{-1, 1}, {-1, 1}, {1, -1}, {1, 1},
  {0, -1}, {0, 1}, {-1, 0}, {1, 0}};
const vii KNIGHT = {{-1, -2}, {-1, 2}, {1, -2}, {1, 2},
    {-2, -1}, {-2, 1}, {2, -1}, {2, 1}};
vii directionMoves(const vii& D, int L, int x, int y) {
  vii moves;
  trav(dir, D) {
    rep(i,1,L+1) {
      int nx = x + dir.first * i, ny = y + dir.second * i;
      if (!isValid(nx, ny)) break;
      if (isEmpty(nx, ny)) moves.emplace_back(nx, ny);
      else {
        if (isWhite(x, y) != isWhite(nx, ny)) moves.emplace_back(nx, ny);
        break;
      }
    }
  }
  return moves;
}
```

A short and sweet abstraction, that will prove very useful. This will handle 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 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 loops 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 will use a `vector<string>`, typedef'd 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)) goto skip;
      if (isMate()) {
          outputSquare(mv.first.first, mv.first.second);
          outputSquare(mv.second.first, mv.second.second);
          cout << endl;
          break;
      }
skip: board = oldBoard;
  }
  return 0;
}
```

Now, we have actually rewritten the entire solution. From the 300-line behemoth with gigantic functions, we have refactored the solution into few, short functions with are easy to follow. The rewritten solution is less than half the size, clocking in at less than 140 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.

3.1 Additional Exercises

Exercise 3.3

Flexible Spaces – flexiblespaces

Sort of Sorting – sortofsorting

Permutation Encryption – permutationencryption

Jury Jeopardy – juryjeopardy

Fun House – funhouse

Settlers of Catan – settlers2

Cross – cross

Basic Interpreter – basicinterpreter

Cat Coat Colors – catcoat

3.2 Chapter Notes

Many good sources exist to become more proficient at writing readable, simple code. *Clean Code*[13] 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[14] 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++*[16], *Effective Modern C++*[17], *Effective STL*[15], by Scott Meyers,

Java *Effective Java*[3] by Joshua Bloch,

Python *Effective Python*[21] by Brett Slatkin, *Python Cookbook*[2] by David Beazley and Brian K. Jones.

Chapter 4

Time Complexity

How do you know if your algorithm is fast enough before you have coded it? In this chapter, we will look at this question from the perspective of ***time complexity***, a common tool of algorithm analysis to determine roughly how fast an algorithm is.

We will start our study of complexity by looking at a new sorting algorithm – ***insertion sort***. Just like selection sort, which we studied in Chapter 1, insertion sort works by iteratively sorting the array.

4.1 The Complexity of Insertion Sort

The insertion sort algorithm works by ensuring 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.

Algorithm 4.1: Insertion Sort

Assume we wish to sort the list $a_0, a_1, ..., a_{N-1}$ of N integers. If we know that the first K elements $a_0, ..., a_{K-1}$ numbers 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 of 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 then 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 will 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.

In this section, we will determine how long time insertion sort takes to run. When analyzing an algorithm, we generally do not attempt 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. When sorting fixed-size integers, this would be the number of elements we are sorting, N. We denote the time the algorithm takes in relation to N as $T(N)$. Note that this is the *worst-case* time, over every instance of N elements.

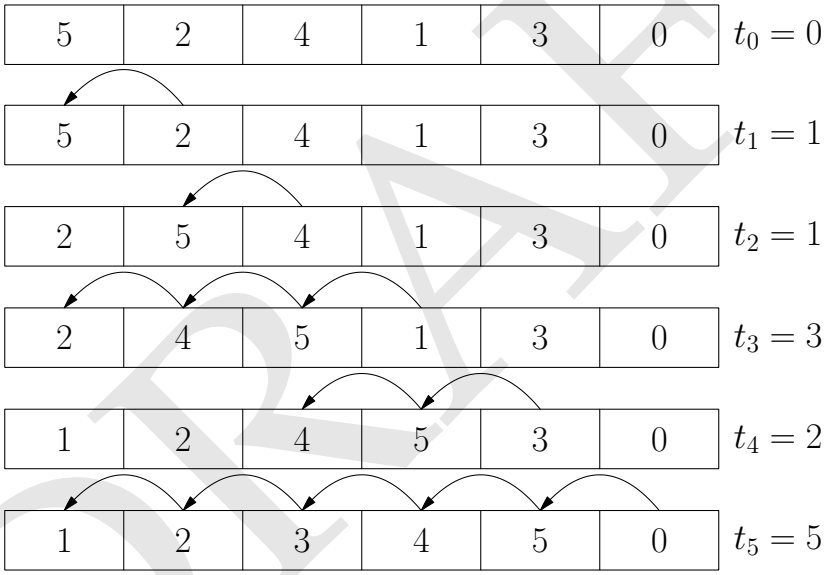


Figure 4.1: Insertion Sort sorting the sequence $2, 1, 4, 5, 3, 0$.

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:

Algorithm 4.2: Insertion sort

```
procedure INSERTIONSORT(A) ▷ Sorts the sequence A containing N elements
    for i ← 0 to N − 1 do
        j ← i
        while j > 0 and A[j] < A[j − 1] do
            Swap A[j] och A[j − 1]
            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, Chapter 8).

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 will execute N times. The number of times the inner loop runs depends on how the input looks. We will introduce the notation t_i to mean the number of iterations the inner loop runs during the i'th iteration of the outer loop. These are included in figure 4.1 for every iteration.

Let us take our pseudo code, and write how many times each line execute:

Algorithm 4.3: Insertion sort

```
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 Σ_{i=0}^{N−1} t_i times, cost 1
5: |  |  |  Swap A[j] och A[j − 1]                   ▷ Runs Σ_{i=0}^{N−1} t_i times, cost 1
6: |  |  |  j ← j − 1                                ▷ Runs Σ_{i=0}^{N−1} t_i times, cost 1
```

We can now express $T(N)$ 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, we have $t_i = i$. This happens when the list we are sorting is in descending order. Then, each element must be moved to the front, which requires i swaps.

This substitution gives us a way to 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. Since we assign the approximate growth of the time a function takes such importance, a notation was developed for it.

4.2 Asymptotic Notation

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

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

to state this fact.

Similarly, the linear function $2N + 15$ is bounded by $c \cdot N$ for large N, with $c = 3$. We also have that $2N + 15 = O(N^2)$, since the asymptotic notation only concerns upper bounds. However, N^2 is not bounded by $c \cdot N$ for any constant c when N is large, so we have that $N^2 \neq O(N)$.

Definition 4.1 — O-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 $f(n) \leq cg(n)$ whenever $n \geq n_0$, we say that $f(n) = O(g(n))$.

Intuitively, the notation means that $f(n)$ grows *slower than or as fast as* $g(n)$, within a constant. Any quadratic function $an^2 + bn + c = O(n^2)$. Similarly, any linear function $an + b = O(n^2)$ as well. This definition implies that for two functions f and g which are always within a constant 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 bounded $O(N^2)$ in the worst case.

Example 4.1 Prove that $\frac{3}{2}N^2 - \frac{N}{2} = O(N^2)$.

Proof. When $N \geq 0$, we have $\frac{3}{2}N^2 - \frac{N}{2} \leq \frac{3}{2}N^2$. Using the constants $c = \frac{3}{2}$ and $n_0 = 1$ we fulfill the condition from the definition. □

For a constant k, we say that $k = O(1)$. This is a slight abuse of notation, but a well-established abuse. We are basically using the symbol 1 to mean the constant function of value 1 – maybe it should be written $1(N)$ instead.

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, with a time limit of 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 4.1: Approximations of needed time complexities

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

Complexity analysis can also be used to determine *lower bounds* of 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 4.2 — Ω-notation
Let f and g be non-negative functions from $\mathbb{R}_{\geq \nu}$ to $\mathbb{R}_{\geq \nu}$. If there exists positive constants n_0 and c such that $cg(n) \leq f(n)$ whenever $n \geq n_0$, we say that $f(n) = O(g(n))$. If $cg(n) \leq f(n)$ for every $n \geq n_0$, where c and n_0 are positive constants of our choice, 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? In fact, it has the same lower bound, i.e. $T(N) = \Omega(N^2)$.

Example 4.2 Prove that $\frac{3}{2}N^2 - \frac{N}{2} = \Omega(N^2)$.

Proof. When $N \geq 1$, we have $\frac{3}{2}N^2 - \frac{N}{2} \geq N^2$ since $N^2 \geq N$. 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 of the worst-case running time of insertion sort coincided (asymptotically). We have another notation for when this is the case:

Definition 4.3 — 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 with 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 8, you will also see some ways of computing the time complexity of a particular type of recursive solution, called Divide and Conquer algorithms.

4.3 NP-complete problems

Of particular importance in computer science are the problems that can be solved algorithms running in polynomial time (often considered to be the "tractable" problems). There are a number of problems for which 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 appear in algorithmic problem solving, so 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.

4.4 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 memory complexity. In most modern programming competitions, the allowed memory usage is high enough for the memory complexity not be a problem. 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. Such a procedure call 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.

4.5 Exercises

Exercise 4.1

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

Exercise 4.2

Give an $O(n)$ algorithm and an $O(1)$ algorithm to compute the sum of the n first integers.

Exercise 4.3

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

Exercise 4.4

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

Exercise 4.5

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

Exercise 4.6

Prove that $(n + a)^b = O(n^b)$ for positive constants $a, b > 0$.

4.6 Chapter Notes

Advanced algorithm analysis sometimes use rather complicated discrete mathematics, such as number theoretical (as in Chapter 14) or combinatorial (as in Chapter 13) facts. *Concrete Mathematics* [7] by Donald Knuth, et al, does a thorough job on both accounts.

An Introduction to the Analysis of Algorithms [8] 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 forms a research area called *computational complexity theory*. *Computational Complexity* [18] by Papadimitriou is a classical introduction to computational complexity, although *Computational Complexity: A Modern Approach* [1] 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 (since we are interested in those problems which can be solved), but is still of general interest. A book on e.g. automata theory (such as *Introduction to Automata Theory, Languages, and Computation* [24] by Ullman et al) can be a good compromise, mixing both some foundations of the theory of computation with topics more applicable to algorithms (such as automatons and languages).

Part II

Basics

Chapter 5

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. The same brute force problem might have a simple algorithm which 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. In this chapter, we will study a few such techniques.

5.1 Optimization Problems

In an ***optimization problem***, we have some ***solution set*** S and a ***value function*** f. The goal is to find an $x \in S$ which maximize $f(x)$, i.e., optimizing the function.

Optimization problems constitute a large class of the problems that we solve in algorithmic problem solving, such as the Max Clique problem and the Buying Books problems we will study in this chapter. One of the most famous optimization problems is the NP-complete *Travelling Salesman Problem*. The problem seeks the shortest cycle that visits all vertices of a weighted graph. The practical applications of this problem are many. A logistics company that must perform a number of deliveries probably want to minimize the distance traveled when visiting all the points of delivery. When planning your backpacking vacation, you may prefer to minimize the cost of travelling between all your destinations. In this problem, the solution set S would consist of all cycles in the graph that visit all the vertices, with $f(x)$ being the sum of all edges in the cycle x.

The brute force technique essentially consists of evaluating $f(x)$ for a large number (sometimes even all) of $x \in S$. For large S, this is slow.

The focus of this chapter and the chapters on Greedy Algorithms (Chapter 6) and Dynamic Programming (Chapter 7) will be to develop techniques that exploit particular structures of optimization problems to avoid evaluating the entire set S.

5.2 Generate and Test

Our first brute force method is the ***generate and test*** method. This particular brute force strategy consists of *generating* solutions – naively constructing candidate solutions to a problem – and then *testing* them – removing invalid solutions. It is applicable whenever the number of candidate solutions is quite small.

Max Clique

In a graph, a subset of the vertices form a ***clique*** if each pair of vertices is connected by an edge. Given a graph, determine the size of the largest clique.

Input

The first line of input contains an integer N and M – the number of vertices and the number of edges of the graph. The vertices are numbered $0, 1, \ldots, N-1$. The next M lines contain the edges of the graph. An edge is given as two space-separated integers A and B – the endpoints of the graph.

Output

Output a single number – the size of the largest clique in the graph.

This problem is one of the so-called NP-complete problems we mentioned in Chapter 4. Thus, a polynomial-time solution is out of reach. We will solve the problem for $N \leq 15$.

Let us analyze whether a generate and test approach is suitable. First of all, we must define what our candidate solutions are. In this problem, only one object comes naturally; subsets of vertices. For every such candidate, we must first test whether it is a clique, and then compute its size, picking the largest clique to arrive at our answer.

In the Max Clique problem, there are only 2^N subsets of vertices (and thus candidate solutions), which is a quite small number. Given such a set, we can verify whether it is a clique in $O(N^2)$ time, by checking if every pair of vertices in the candiate set has an edge between them. To perform this check in $\Theta(1)$ time, we keep a 2D vector `adj` such that `adj[i][j]` is true if and only if vertices `i` and `j` are adjacent to each other. This gives us a total complexity of $O(2^N \cdot N^2)$. According to our table of complexities

(Table 4.1), this should be fast enough for N = 15. A C++ implementation for this can be seen in Algorithm 5.1. Note the nifty use of integers interpreted as bitsets to easily iterate over every possible subset of an N-element set, a common technique of generate and test solutions based on subsets.

Algorithm 5.1: Max Clique

```
int main() {
  int N, M;
  cin >> N >> M;
  vector<vector<bool>> adj(N, vector<bool>(N));
  rep(i,0,M) {
    int A, B;
    cin >> A >> B;
    adj[A][B] = adj[B][A] = true;
  }
  rep(i,0,N) adj[i][i] = true;
  rep(i,0,1<<N) {
    rep(j,0,N) {
      if (!(i & (1 << j))) continue;
      rep(k,0,N) {
        if (!(i & (1 << k))) continue;
        if (!adj[j][k]) goto skip;
      }
    }
    ans = max(ans, __builtin_popcount(i));
skip:;
  }
  cout << ans << endl;
}
```

This kind of brute force problem is often rather easy to spot. There will be a very small input limit on the parameter you are to brute force over. The solution will often be subsets of some larger base set (such as the vertices of a graph).

Exercise 5.1 — Kattis Exercises

4 thought – 4thought

Lifting Walls – walls

Let us look at another example of this technique, where the answer is not just a subset.

The Clock

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 between the minute and the hour hands, since this uniquely determines the time. However, many people are unused to this way of specifying the time, so it would be nice to have a program which translates this to a more common format.

We assume that our clock have no seconds hand, and only display 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 will be specified in tenths of a degree.

Input

The first and only line of input contains a single integer $0 \leq A < 3600$, the angle specified in tenths of a degree.

Output

Output the time in the format `hh:mm` between 00:00 and 11:59.

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 will then be at angle $\frac{360}{60}m = 6m$ degrees. Similarly, the hour hand moves $\frac{360}{12}h = 30h$ degrees due to the hours, and $\frac{360}{12}\frac{1}{60}m = 0.5m$ degrees due to the minute. While computing the current time directly from the angle is difficult, computing the angle directly from the current time is easy.

Our solution will be to test the $60 \cdot 12 = 720$ different times, and pick the one which matched the given angle (Algorithm 5.2).

Algorithm 5.2: The Clock

```
procedure CLOCK(A)
    for h ← 0 to 11 do
        for m ← 0 to 59 do
            hourAng ← 300h + 5m
            minuteAng ← 60m
            angBetween ← (hourAng − minuteAng + 3600) mod 3600
            if angBetween = A then
                return h:m
```

Competitive Tip

Sometimes, competitions pose problems which are solvable quite fast, but a brute force algorithm will suffice as well. As a general rule, code the simplest correct solution that is fast enough, even if you see a faster one.

Exercise 5.2 — Kattis Exercises

All about that base – allaboutthatbase

Natjecanje – natjecanje

Perket – perket

5.3 Backtracking

Backtracking is a variation of the generate and test method. Most of the time it is faster, but it can sometimes be more difficult to code (in particular when the solutions are subsets).

Let us return to the Max Clique problem. In the problem, we generated all the candidate solutions (i.e., subsets of vertex) by using bitsets. In problems where the candidate solutions are other objects than subsets, or the number of subsets is too large to iterate through, we need to construct the solutions in another way. Generally, we do this recursively. For example, when generating subsets, we would go through every element one at a time, and decide whether to include it. Backtracking extends generate and test to not only testing all the candidate solutions, but also these partial candidates. Thus, a backtracking approach can be faster than an iterative approach, by testing fewer candidate solutions.

For example, assume that two vertices a and b are not connected by an edge in the graph. If we decide to include a in our solution, we already know that a candidate solution can not include b. Thus, we have managed to reduce four choices – including or excluding a and b – to three choices, by eliminating the inclusion of both vertices. Algorithm 5.3 contains a C++ implementation of this.

Algorithm 5.3: Max Clique, Recursive Variant

```
1  // excluded is a bit set containing the vertices with a chosen neighbour
2  int best(int at, int excluded, int clique, const vector<int>& adjacent) {
3    if (at == sz(adjacent)) return clique;
4      // Case 1: Not including the current vertex
5    int answer = best(at + 1, excluded, clique, adjacent);
6      // Case 2: We can include the current vertex
```

```
7     if (!((1 << at)&excluded)) {
8       answer = max(answer,
9         best(at + 1, excluded | ~adjacent[at], clique + 1, adjacent));
10    }
11    return answer;
12  }
13
14  int main() {
15    int N, M;
16    cin >> N >> M;
17    vector<int> adjacent(N);
18    rep(i,0,M) {
19      int A, B;
20      cin >> A >> B;
21      adjacent[A] |= 1 << B;
22      adjacent[B] |= 1 << A;
23    }
24    cout << best(0, 0, 0, adjacent) << endl;
25  }
```

As written, this version has the same complexity as the generate-and-test version. Our improvement only generated fewer branches whenever the graph contained edges. In the case where the graph consists of N vertices and no edges, this is not the case. The recursion then degenerates to branching twice N times, resulting in a $\Theta(2^n)$ complexity.

Backtracking works whenever we can construct our solutions iteratively (as in constructing it part by part), and quickly determine whether such a partial solution can possibly be completed to an admissible solution.

Exercise 5.3 — Kattis Exercises

Class Picture – classpicure

Boggle – boggle

Geppetto – geppetto

Map Colouring – mapcolouring

Sudokunique – sudokunique

All Friends – friends

As a general principle, backtracking seems simple enough. Some backtracking solutions require a bit more ingenuity, as in the next problem.

Basin City Surveillance

Nordic Collegiate Programming Contest 2014 – Pål G. Drange and Markus S. Dregi

BASIN CITY is known for her incredibly high crime rates. The police see no option but to tighten security. They want to install traffic drones at different 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 neighbouring intersections. As is usual in many cities, there are no intersections in Basin City with more than four other neighbouring 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 feasible to position exactly this number of drones.

Input

The first line contains an integer k ($0 \leq k \leq 15$), giving the number of drones to position. Then follows one line with $1 \leq n \leq 100\,000$, the total number of intersections in Basin City. Finally follow n lines describing consecutive intersections. The i'th line describes the i'th intersection in the following format: The line starts with one integer d ($0 \leq d \leq 4$) describing the number of intersections neighbouring the i'th one. Then follow d integers denoting the indices of these neighbouring intersections. They will be all distinct and different from i. The intersections are numbered from 1 to n.

Output

If it is possible to position k drones such that no two neighbouring intersections have been assigned a drone, output a single line containing `possible`. Otherwise, output a single line containing `impossible`.

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 vertices is a huge number of intersections! We can make the problem a bit more reasonable with our first insight. If we have a large number of intersections, and every intersection is adjacent to very few other intersection, it is probably very easy to place the drones at appropriate intersections. To formalize this insight, consider what happens when we place a drone at an intersection.

By placing a drone at the intersection marked in black in Figure 5.1, at most five intersections are affected – the intersection we placed the drone at, along with its

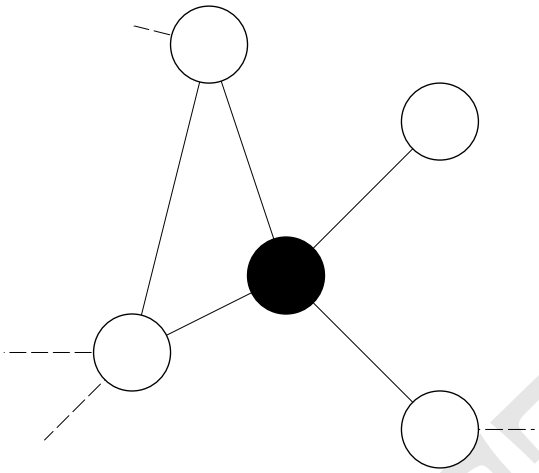


Figure 5.1: The intersections affected by placing a drone at an intersection.

neighbouring 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 neighbourhood, since no further placements will take place.

Therefore, we can assume that the number of vertices 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 we used for the Max Clique problem. We could recursively construct the set of our k drones by, for each intersection, try to either place a drone there or not. If placing a drone at an intersection, we would forbid placing drones at any neighbouring intersection.

Unfortunately, this basically means that we are testing every intersection when placing a certain drone somewhere. This would give us a complexity of $O(n^k)$. More specifically, the execution time $T(n,k)$ would satisfy $T(n,k) \approx T(n-1,k) + T(n-1,k-1)$, which implies $T(n,k) \approx \binom{n}{k} = \Omega(n^k)$ (see Section 13.4 for more details). For $n = 70, k = 15$, this will almost certainly be way too high. 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 achiving 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 $O(c^k)$.

The trick, yet again, comes from Figure 5.1. Assume that we choose to include this

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? Basically, one of its neighbours would have to be included in every optimal solution. Fortunately for us, this gives us just what we need to improve our algorithm – either a given intersection, or one of its neighbours, must 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 will give 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, applying a common principle will speed things up dramatically (even giving us a better time complexity).

First of all, we can partition the intersections into sets that are connected by sequences by roads. These sets are all independent of each other, so we can solve them separately by computing the maximal number of drones we can place in every such set, until we have processed enough sets to place k drones.

With this in hand, we can use the following insight: if we at every step branch on the intersection with the fewest neighbours, we will instead achieve a complexity of $O(4^k)$. After branching on the first drone, there will always be one intersection with at most 3 neighbours – leaving us with only 4 choices we must test when placing a drone. We can prove this easily by contradiction. Assume that after placing a number of drones, every intersection has exactly 4 neighbours. Then, none of these intersections can be the neighbour of an intersection we have removed so far. However, this means the set of intersections removed so far and the remaining intersections are disconnected, a contradiction.

While such an algorithm would be significantly faster than the $O(5^k)$, 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 neighbours must be included in any optimal solution. Can it ever be the case that only *one* of its neighbours are included in an optimal solution, as in Figure 5.2?

Clearly, this is never the case. We can simply move the drone to the intersection from its neighbour, since there is no other neighbour causing problems. Now, we are basically done. For any intersection, there will either be an optimal solution including it, or two of its neighbours. Since an intersection has at most 4 neighbours, it has at most 6 pairs of neighbours. This means our recursion will take time $T(k) = T(k-1) + 6T(k-2)$ in the worst case. This recurrence has the solution 3^k, since $3^{k-1} + 6 \cdot 3^{k-2} = 3^{k-1} + 2 \cdot 3^{k-1} = 3 \cdot 3^{k-1} = 3^k$. A final improvement would be to combine this insight with the independence of the connected subsets of intersections. The second term of

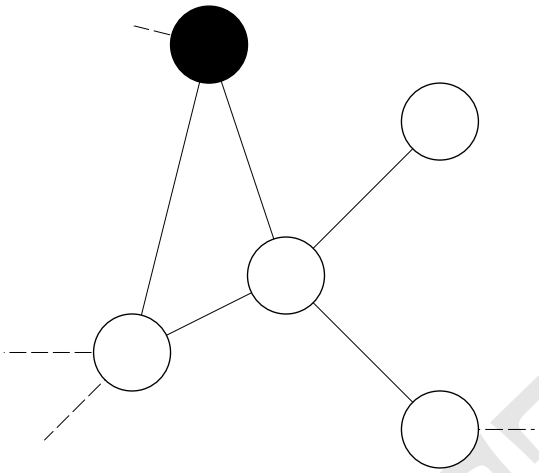


Figure 5.2: Placing a drone at a single neighbour of an intersection.

the time recurrence would then be a 3 instead of a 6 (as 3 neighbours make 3 pairs). Solving this recurrence would give us the complexity $O(2.31^k)$ instead.

The general version of this problem (without the bounded degree) is called *Independent Set*

So, what is the take-away regarding backtracking? First of all, find a way to construct candidate solutions iteratively. Then, try to integrate the process of testing the validity of a complete solution with the iterative construction, in the hope of significantly reducing the number of candidate solutions which 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).

Exercise 5.4 — Kattis Exercises

Domino – domino

Fruit Baskets – fruitbaskets

Infiltration – infiltration

Vase Collection – vase

5.4 Fixing Parameters

The parameter fixing technique is also similar to the generate and test method, but is not used to test the solution set itself. Instead, you perform brute force to *fix* some parameter in the problem by trying possible values they can assume. Hopefully, fixing the correct parameters allows you to solve the remaining problem easily. The intuition is that while any particular choice of parameter may be wrong, testing every choice allows us to assume that we at some point used the correct parameter.

Buying Books

Swedish Olympiad in Informatics 2010, Finals

You are going to buy N books, and are currently checking the different internet M book shops for prices. Each book is sold by at least one book store, and can vary in prices between the different stores. Furthermore, if you order anything from a certain book store, you must pay for postage. Postage may vary between book stores as well, but is the same no matter how many books you decide to order. You may order books from any number of book stores. Compute the smallest amount of money you need to pay for all the books.

Input

The first line contains two integers $1 \leq N \leq 100$ – the number of books, and $1 \leq M \leq 15$ – the number of book stores.

Then, M descriptions of the book stores follow. The description of the i'th store starts with a line containing two integers $0 \leq P_i \leq 1000$ (the postage for this book store), and $1 \leq L_i \leq N$ (the number of books this store sells). The next L lines contains the books sold by the store. The j'th book is described by two integers $0 \leq B_{i,j} < N$ – the (zero-indexed) number of a book being sold here, and $1 \leq C_{i,j} \leq 1000$ – the price of this book at the current book store.

Output

Output a single integer – the smallest amount of money you need to pay for the books.

If we performed naive generate and test on this problem, we would probably get something like 15^{100} solutions (testing every book shop for every book). This is infeasible. So, why can we do better than this? There must be some hidden structure in the problem which makes testing all those possibilities unnecessary. To find this structure, we will analyze a particular candidate solution as given by the naive generate and test method. Can we find a simple criteria for when a solution obvious cannot be optimal?

It turns out that yes, we can. Assume that we, in a particular solution, bought a book from some book store A, but it was available for a lower price at book store B. If this is an optimal solution we must not have bought any books from B.

When this happens, i.e., you notice very simple constraints which immediately disqualify many of the solution candidates from being optimal, it is a good sign you may be able to restructure your brute force in order to avoid evaluating many of them. In our case, this observation hints that making a choice for every book is not particularly good.

We *could* decide to use this fact to turn our generate and test into a backtracking algorithm, by pruning away any solution where we have bought a book which is available at a cheaper price from another book store we also used. Unfortunately, this is easily defeated by giving most of the books equal prices at most of the book stores.

Instead, let us use the observation differently. The entirety of the observation quite plainly told us that we do not really have any choice in where we buy the books! Indeed, once we know what book stores we have bought from, we are forced to buy the book from the cheapest store that is used. At this point, we are basically done. We have reduced the amount of information we need to fix to solve the problem to "which book stores will we purchase from?". This parameter has only 2^{15} possibilities. After fixing it, we can compute the rest of the answer using the "cheapest store for each book" rule. Since we test every possible such parameter, we must also include the optimal one. The pseudo code for this solution is in Algorithm 5.4.

Algorithm 5.4: Buying Books

```
procedure BUYINGBOOKS(books N, stores M, costs C, postages P)
    answer ← ∞
    for every S ⊆ [M] do
        cost ← 0
        for every s ∈ S do
            cost ← cost + P_s
        for every b ∈ [N] do
            cost ← cost + min_{i∈S} C_{i,b}
        answer ← min(answer, cost)
    return answer
```

Alternatively, we could have come to the same insight by simply asking ourselves "can we bruteforce over the number of book shops?". Whenever you have a parameter about this size in such a problem, this is a question worth asking. However, 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.

Integer Equation

Codeforces Round #262, Problem B

Find all integers $0 < x < 10^9$ which satsify the equation

$$x = a \cdot s(x)^b + c$$

where a, b and c are given integers, and $s(x)$ is the digit sum of x.

Input
The input contains the three integers a, b, and c such that

$$|a| \leq 10\,000$$

$$1 \leq b \leq 5$$

$$|c| \leq 10\,000$$

Output
Output a single integer – number of integer solutions x to the equation.

In this problem, the only explicit object we have is x. Unfortunately, 10^9 is a tad too many possibilities. If we study the equation a bit closer, we see that $s(x)$ also varies as a function of x. This is helpful, since $s(x)$ has far fewer options than x. In fact, a number bounded by 10^9 has at most 9 digits, meaning it has a maximum digit sum of $9 \cdot 9 = 81$. Thus, we can solve the problem by looping over all the possibles values of $s(x)$. This uniquely fixes our right hand side, which equals x. Given x, we verify that $s(x)$ has the correct value. Whenever we have such a function, i.e., one with a large domain (like x in the problem) but a small image (like $s(x)$ in the problem), this technique can be used by brute forcing over the image instead.

Exercise 5.5 — Kattis Exercises

Shopping Plan – shoppingplan

5.5 Meet in the Middle

The *meet in the middle* technique is essentially a special case of the parameter fixing technique. The general theme will be to fix half of the parameter space and build some fast structure such that when testing the other half of the parameter space, we can avoid explicitly re-testing the first half. It is a space-time tradeoff, in the sense that we improve the time usage (testing half of the parameter space much faste much faster), by paying with increased memory usage (to save the pre-computed structures).

Subset Sum

Given a set of integers S, is there some subset $A \subseteq S$ with a sum equal to T?

Input

The first line contains an integer N, the size of S, and T. The next line contains N integers $s_1, s_2, ..., s_N$, separated by spaces – the elements of S. It is guaranteed that $s_i \neq s_j$ for $i \neq j$.

Output

Output `possible` if such a subset exists, and `impossible` otherwise.

In this problem, we have N parameters in our search space. 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 $N = 40$.

As it happens, our parameters are to a large extent independent. If we fix e.g. the $\frac{N}{2}$ first parameters, the only constraint they place on the remaining $\frac{N}{2}$ parameters is the sum of the integers in subset they decide. This latter constraint takes $O(2^{\frac{N}{2}})$ time to check for each choice of the first half of parameters if we use brute force. However, the computation performed is essentially the same, only differing in what sum we try to find. We will instead trade away some memory in order to only compute this information once, by first computing all possible sums that the latter half of the elements can form. This information can be inserted into a hash set, which allows us to determine if a sum can be formed by the elements in $\Theta(1)$ instead. Then, the complexity is instead $\Theta(N2^{\frac{N}{2}})$ in total.

Algorithm 5.5: Subset Sum

```
procedure SUBSETSUM(set S, target T)
    N ← |S|
    left ← N/2
    right ← N − left
    Lset ← the left first elements of S
    Rset ← S \ Lset
    Lsums ← new set
    for each L ⊆ Lset do
        Lsums.insert(∑_{l∈L} l)
    for each R ⊆ Rset do
        sum ← ∑_{r∈R} r
        if Lsums.contains(T − sum) then
            output true
```

```
        |   |   return
    output false
```

Exercise 5.6 — Kattis Exercises

Closest Sums – closestsums

Maximum Loot – maxloot

Celebrity Split – celebritysplit

Circuit Counting – countcircuits

Indoorienteering – indoorienteering

Key to Knowledge – keytoknowledge

Knights in Fen – knightsfen

Rubrik's Revenge in ... 2D!? 3D? – rubriksrevenge

5.6 Chapter Notes

write chapter notes

Chapter 6

Greedy Algorithms

In this chapter, we are going to look at another standard technique to solve some categories of search and optimization problems faster than naive bruteforce, by exploiting properties of *local optimality*.

6.1 Optimal Substructure

Most optimization problems we study consist of making a series of sequential choices. They are often be of the following form: Given a weighted, directed acyclic graph (DAG) on N vertices, what is the "best" path from a vertex S to another vertex T? This graph is almost never be given explicitly. Instead, it hides behind the problem as a set of states (the vertices). At each state, we are to make some choice that takes us to another state (traversing an edge). If the path consists of edges $e_1, e_2, \ldots, e_k$, the function we are to maximize will be of the form

$$G(e_1, e_2, \ldots, e_k) = g(e_1, g(e_2, g(\ldots, g(e_k, 0))))$$

where g is a function from $E \times R$ to R. We will denote $B(v)$ as the maximum value of $G(e_1, e_2, \ldots, e_k)$ over all paths from v to T. Often, g will be the negative sum of the edge weights, meaning we look for the shortest path from S to T. If $g(e, x)$ is increasing in x, we say that problems exhibiting this property has *optimal substructure*.

One way to solve the problem would be to evaluate G for every path in the graph. Unfortunately, the number of paths in a general graph can be huge (growing exponentially with the number of vertices). However, the optimal substructure property allows us to make a simplification. Assume that S has neighbors $v_1, v_2, ..., v_m$. Then by the definition of B and g, $B(s) = \max(g(\{s, v_i\}, B(v_i))$. Thus, we can solve the problem by first solving **the same problem** on all vertices v_i instead.

Change-making Problem, Denominations $1, 2, 5$

Given an infinite number of coins of denominations $1, 2, 5$, determine the smallest number of coins needed to sum up to T.

Input

The input contains a single integer $1 \leq T \leq 10^6$.

Output

Output a single integer, the minimum number of coins needed.

We can phrase this problem using the kind of graph we previously discussed. Let the graph have vertices labeled $0, 1, 2, ..., T$, representing the amount of money we wish to sum up to. For a vertex labeled x, we add edges to vertices labeled $x - 1$, $x - 2$, $x - 5$ (if those vertices exist), weighted 1. Traversing such an edge represents adding a coin of denomination 1, 2 or 5. Then, the Change-making Problem can be phrased as computing the shortest path from the vertex T to 0. The corresponding graph for $T = 5$ can be seen in Figure 6.1.

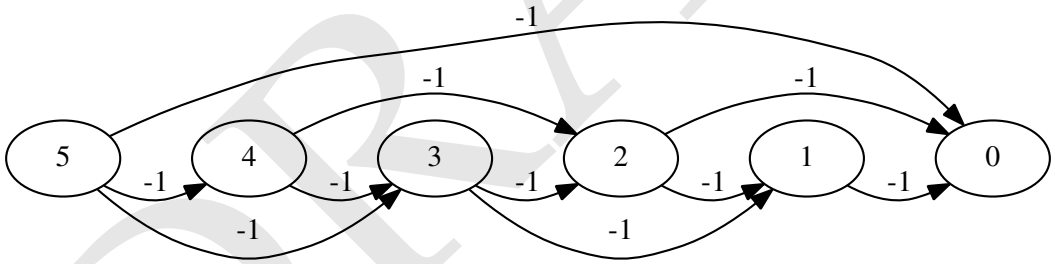


Figure 6.1: The Change-making Problem, formulated as finding the shortest path in a DAG, for $K = 5$.

So, how does this graph formulation help us? Solving the problem on the graph as before using simple recursion would be very slow (with an exponential complexity, even). In Chapter 7 on Dynamic Programming, we will see how to solve such problems in polynomial time. For now, we will settle with solving problems exhibiting yet another property besides having optimal substructure – that of local optimality.

Exercise 6.1

Compute the shortest path from each vertex in Figure 6.1 to T using the optimal substructure property.

6.2 Locally Optimal Choices

Greedy algorithms solve this kind of problem by making what is called a *locally optimal choice*. We generally construct our shortest path iteratively, one edge at a time. We start out at the first vertex, S, and need to choose an edge to traverse. A greedy algorithm chooses the edge which locally looks like a good choice, without any particular thought about future edges.

For example, consider how you would solve the Change-making Problem yourself. You would probably attempt to add the 5 coin as many times as possible, until you need to add less than 5. This makes sense locally, since 5 is the largest amount we may charge at a time. When we need to add less than 5, we would probably instead add coins worth 2, until we need to add either 0 or 1. In the latter case, we would add a final 1 coin.

Intuitively, this makes sense. Adding the highest amount every time *ought* to be the best possible. For this problem, this is actually true. However, if the denominations were different, this could fail (Exercise 6.2).

Exercise 6.2

Prove that the greedy choice may fail if the coins have denominations 1, 6 and 7.

Assume that the optimal solution uses the 1, 2 and 5 coins a, b and c times respectively. We then have that either:

$a = 1, b \leq 1$ If $b \geq 2$, we could exchange one 1 coin and two 2 coins for one 5 coin.

$a = 0, b \leq 2$ If $b \geq 3$, we could exchange three 2 coins one 1 coin and one 5 coin.

If $a \geq 2$, we could instead add a single 2 coin instead of two 1 coins.

This means that the possibilities for a and b are few:

- $a = 0, b = 0$: value 0
- $a = 1, b = 0$: value 1
- $a = 0, b = 1$: value 2
- $a = 1, b = 1$: value 3
- $a = 0, b = 2$: value 4

Now, assume that we use exactly c coins of value 5, so that $T = 5c + r$. We know that $0 \leq r < 5$. Otherwise, we would be summing up to an amount larger than 4 without using any 5 coins, but this is impossible based on the above list. This means c must be as large as possible, i.e. it is optimal to add as many 5 coins as possible – which the greedy choice will. Then, only the cases 0, 1, 2, 3 and 4 remain. Looking at the list of those cases, we see that their optimal solutions correspond to how the greedy algorithm works. Thus, the greedy algorithm will always give the optimal answer.

Competitive Tip

If you have the choice between a greedy algorithm and another algorithm (such as one based on brute force or dynamic programming), use the other algorithm unless you are certain the greedy choice works.

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 problems that are solvable using a greedy algorithm. Take note of the kind of arguments we are going to use – there are a few common types of proofs which are often used in proofs of greedy algorithms.

6.3 Scheduling

Scheduling problems are a class of problems which deals with constructing large subsets of non-overlapping intervals, from some given set of intervals.

The classical Scheduling Problem is the following.

Scheduling Problem

Given is a set of half-open (being open on the right) intervals S. Determine the largest subset $A \subseteq S$ of non-overlapping intervals.

Input

The input contains the set of intervals S, where $|S|$.

Output

The output should contain the subset A.

We will construct the solution iteratively, adding one interval at a time. When looking for greedy choices to perform, extremal cases are often the first ones you should consider. Hopefully, one of these extremal cases can be proved to be included in an optimal solution. For intervals, some extremal cases would be:

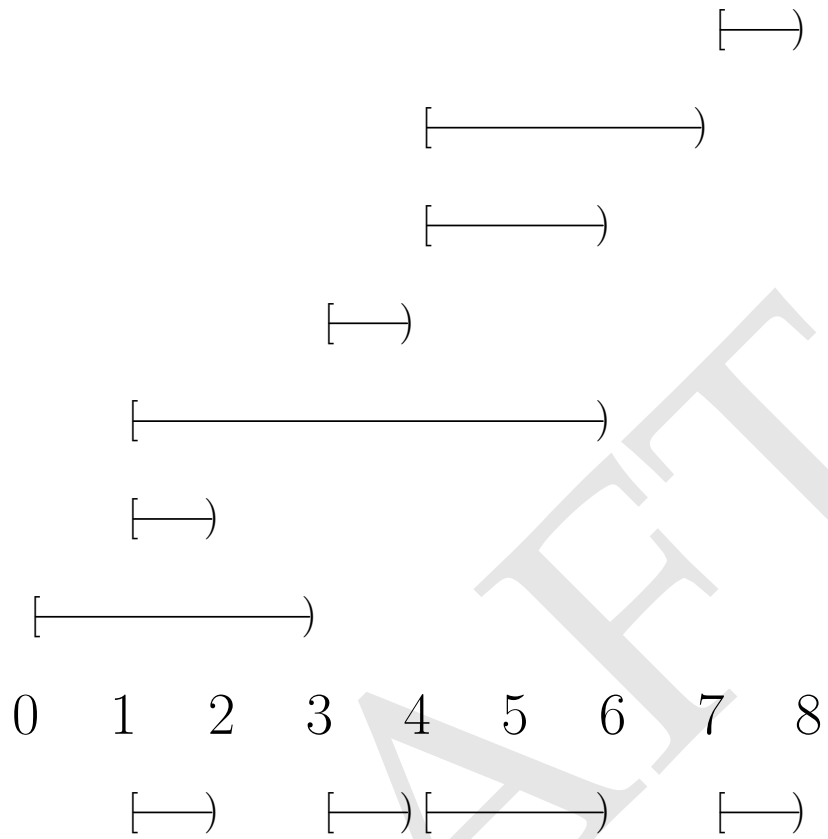


Figure 6.2: An instance of the scheduling problem, with the optimal solution at the bottom.

- a shortest interval,
- a longest interval,
- an interval overlapping as few other intervals as possible,
- an interval with the leftmost left endpoint (and symmetrically, the rightmost right endpoint),
- an interval with the leftmost right endpoint (and symmetrically, the rightmost left endpoint).

As it turns out, we can always select an interval satisfying the fifth case. In the example instance in Figure 6.2, this results in four intervals. First, the interval with the leftmost right endpoint is the interval $[1, 2)$. If we include this in the subset A, 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 $A = \{[1,2), [3,4), [4,6), [7,8)\}$.

Algorithm 6.1: Scheduling

```
procedure SCHEDULING(set S)
    ans ← new set
    Sort S by right endpoint
    highest ← ∞
    for each interval [l, r) ∈ S do
        if l ≥ highest then
            ans.insert([l, r))
            highest ← r
        Lsums.insert(∑_{l∈L} l)
    return ans
```

We can prove that this strategy is optimal using a ***swapping argument***, one of the main greedy proof techniques. In a swapping argument, we attempt to prove that given any solution, we can always modify it in such a way that our greedy choice is no worse. This is what we did in the Change-making Problem, where we argued that an optimal solution had to conform to a small set of possibilities, or else we could swap some set of coins for another (such as two coins worth 1 for a single coin worth 2).

Assume that the optimal solution does not contain the interval $[l, r)$, an interval with the leftmost right endpoint. The interval $[l', r')$ that has the leftmost right endpoint *of the intervals in the solution*, must have $r' > r$. In particular, this means any other interval $[a, b)$ in the solution must have $a \geq r'$ (or else the intervals would overlap). However, this means that the interval $[l, r)$ does not overlap any other interval either, since $a \geq r' > r$ so that $a \geq r$. Then, swapping the interval $[l, r)$ for $[l', r')$ still constitute a valid solution, of the same size. This proves that we could have included the interval $[l, r)$ in the optimal solution. Note that the argument in no way say that the interval $[l, r)$ *must* be in an optimal solution. It is possible for a scheduling problem to have many distinct solutions. For example, in the example in Figure 6.2, we might just as well switch $[4, 6)$ for $[4, 7)$ and still get an optimal solution.

This solution can be implemented in $\Theta(|S| \log |S|)$ time. In Algorithm 6.1, this is accomplished by sort performing a sort (in $\Theta(|S| \log |S|)$), followed by a loop for each interval, where each iteration takes $\Theta(1)$ time.

Exercise 6.3

For each of the first four strategies, find a set of intervals where they fail to find an optimal solution.

We can extend the problem to choosing exactly K disjoint subsets of non-overlapping intervals instead, maximizing the sum of their sizes. The solution is similar to the original problem, in that we always wish to include the interval with the leftmost right endpoint in one of the subsets if possible. The question is then, what subset? Intuitively, we wish to choose a subset there the addition of our new interval causes as little damage as possible. This subset is the one with the rightmost right endpoint that we can place the interval in. Proving this is similar to the argument we used when deciding what interval to choose, and is a good exercise to practice the swapping argument.

Exercise 6.4

Prove that choosing to place the interval in the subset with the rightmost right endpoint is optimal.

Exercise 6.5 — Kattis Exercises

Entertainment Box – entertainmentbox

Disastrous Downtime – downtime

6.4 Chapter 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[12]. It possible to determine what cases can be solved using the greedy algorithm described in polynomial time though[4]. Such a set of denominations is called a *canonical coin system*.

Introduction to Algorithms[5] also treats the scheduling problem in its chapter in greedy algorithms. It also brings up the connection between greedy problems and a concept known as *matroids*, which is well worth studying.

Chapter 7

Dynamic Programming

This chapter will study a technique called *dynamic programming* (often abbreviated DP). In one sense, it is simply a technique to solve the general case of the best path in a directed acyclic graph problem (Section 6.1) in cases where the graph does not admit locally optimal choices, in time approximately equal to the number of edges in the graph. For graphs which are essentially trees with a unique path to each vertex, dynamic programming is no better than brute force. In more interconnected graphs, where many paths lead to the same vertex, the power of dynamic programming shines through. It can also be seen as a way to speed up recursive functions (called *memoization*), which will be our first application.

First, we will see a familiar example – the Change-making problem, with a different set of denominations. Then, we will discuss a little bit of theory, and finally round of with a few concrete examples and standard problems.

7.1 Best Path in a DAG

We promised a generalization that can find the best path in a DAG that exhibit optimal substructure even when locally optimal choices does not lead to an optimal solution. In fact, we already have such a problem – the Change-making Problem from Section 6.1, but with certain other sets of denominations. Exercise 6.2 even 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 can we adapt our solution to this case? The secret lies in the graph formulation of the problem which we constructed (Figure 6.1). For the greedy solution, we essentially performed a recursive search on this graph, except we always knew which edge to pick. When we do not know, the solution ought to be obvious – let us test *all*

the edges.

Algorithm 7.1: Change-making Problem

```
procedure CHANGEMAKING(denominations D, target T)
    if T = 0 then
        return 0
    ans ← ∞
    for denomination d ∈ {1, 6, 7} do
        if T ≥ d then
            ans = min(ans, 1 + ChangeMaking(D, T − d))
    return ans
```

This solution as written is actually exponential in T (it is $\Omega(3^{\frac{T}{7}})$). The recursion tree for the case $T = 10$ can be seen in Figure 7.1.

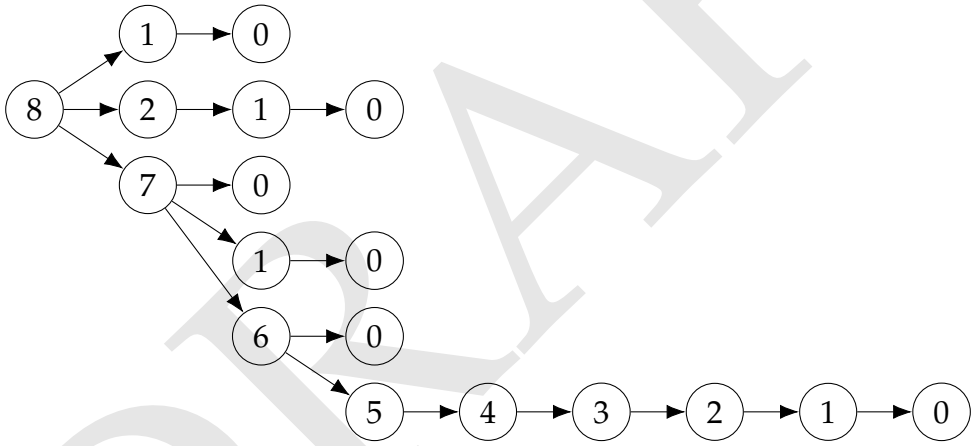


Figure 7.1: The recursion tree for the Change-making problem with $T = 10$.

The key behind the optimal substructure property is that the answer for any particular call in this graph with the same parameter c is the same, independently of the previous calls in the recursion. Right now, we perform calls with the same parameters multiple times. Instead, we can save the result of a call the first time we perform it (Algorithm 7.2).

Algorithm 7.2: Change-making Problem, memoizatoin

```
memo = new int[T + 1]
set memo[i] = −1 for all i
procedure CHANGEMAKING(denominations D, target T)
    if T = 0 then
        return 0
```

```
    if memo[T] ≠ −1 then
        return memo[T]
    ans ← ∞
    for denomination d ∈ D do
        if T ≥ d then
            ans = min(ans, 1 + ChangeMaking(D, T − d)
    memo[T] ← ans
    return ans
```

Theis new algorithm is actually linear in T instead of exponential. The call graph now looks very different (Figure 7.2), since all calls with the same parameter will be merged (as such calls are only evaluated once).

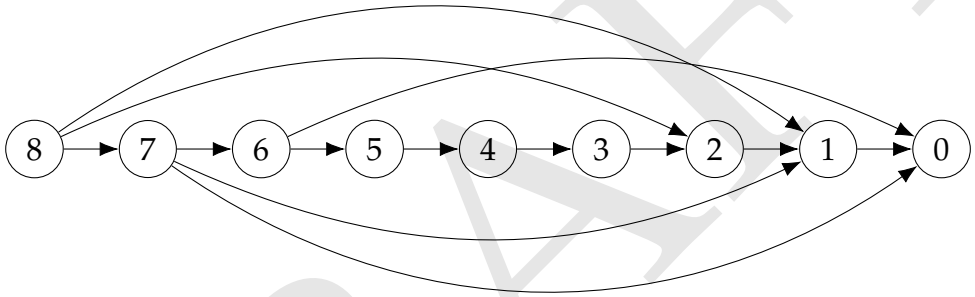


Figure 7.2: The recursion tree for the Change-making problem with $T = 10$, with duplicate calls merged.

Note the similarity between this graph and our previous DAG formulation of the Change-making problem (Figure 6.1).

7.2 Dynamic Programming

With these examples in hand, we are ready to give a more concrete characterization of dynamic programming. In principle, it can be seen as solving the kind of "sequence of choices" problems that we used bruteforce to solve, but where different choices can result in the same situation. For example, in the Change-making Problem, after adding two coins worth 1 and 6, we might just as well have added a 7 coin instead. After we have performed a sequence of choices, *how* we got to the resulting state is no longer relevant – only where we can go from there. Basically, we throw away the information (what exact coins we used) that is no longer needed. This view of dynamic programming problems as having a "forgetful" property, that the exact choices

we have made do not affect the future, is useful in most dynamic programming problems.

Another, more naive view, is that dynamic programming solutions are simple recursions, where we happen to solve the same recursive subproblem a large number of times. In this view, a DP solution is basically nothing more than a recursive solution – find the correct base cases, a fast enough recursion, and memoize the results.

More pragmatically, DP consists of two important parts – the states of the DP, and the computation of a state. Both of these parts are equally important. Fewer states generally mean less computations to make, and a better complexity per state gives a better complexity overall.

7.2.1 Bottom-Up Computation

When applied to a dynamic programming problem, memoization is sometimes called *top-down dynamic programming* instead. The name is inspired from the way we compute the solution to our problem by starting at the largest piece at the top of the recursion tree, and recursively breaking it down to smaller and smaller pieces.

There is an alternative way of implementing a dynamic programming solution, which (not particularly surprisingly) is called *bottom-up dynamic programming*. This method instead constructs the solutions to our sub-problems in the other order, starting with the base case and iteratively computing solutions to larger sub-problems.

For example, we might just as well compute the solution to the Change-making problem the following way:

Algorithm 7.3: Change-making Problem, Bottom-Up

```
procedure CHANGEMAKING(denominations D, target T)
    ans = new int[T + 1]
    set ans[i] = ∞ for all i
    set ans[0] = 0
    for i = 1 to T do
        for denomination d ∈ D do
            if i ≥ d then
                ans[i] = min(ans[i], 1 + ans[i − d])
    return ans[T]
```

How do you choose between bottom-up and top-down? Mostly, it comes down to personal choice. A dynamic programming solution will almost always be fast enough no matter if you code it recursively or iteratively. There are some performance concerns, both ways. A recursive solution is affected by the overhead of recursive

function calls. This problem is not as bad in C++ as in many other languages, but it is still noticeable. When you notice that the number of states in your DP solution is running a bit high, you might want to consider coding it iteratively. Top-down DP, on the other hand, has the upside that only the states reachable from the starting state will be visited. In some DP solutions, the number of unreachable states which are still in some sense "valid" enough to be computed bottom-up is significant enough that excluding them weighs up for the function call overhead. In extreme cases, it might turn out that an entire parameter is uniquely given by other parameters (such as the Ferry Loading problem in Section 7.3). While we probably would notice when this is the case, the top-down DP saves us when we do not.

7.2.2 Order of Computation and Memory Usage

For top-down DP, the memory usage is often quite clear and unavoidable. If a DP solution has N states, it will have an $\Omega(N)$ memory usage. For a bottom-up solution, the situation is quite different.

Firstly, let us consider one of the downsides of bottom-up DP. When coding a top-down DP, you do not need to bother with the structure of the graph you are solving your problem on. For a bottom-up DP, you need to ensure that whenever you solve a subproblem, you have already solved its subproblems too. This requires you to define an *order of computation*, such that if the subproblem a is used in solving subproblem b, a is computed before b.

In most cases, such an order is quite easy to find. Most parameters can simply be increasing or decreasing, using a nested loop for each parameter. When a DP is done over intervals, the order of computation is often over increasing or decreasing length of the interval. DP over trees usually require a post-order traversal of the tree. In terms of the recurring graph representation we often use for DP problems, the order of computation must be a topological ordering of the graph.

While this order of computation business may seem to be nothing but a nuisance that bottom-up users have to deal with, it is related to one of the perks of bottom-up computation. If the order of computation is chosen in a clever way, we need not save every state during our computation. Consider e.g. the Change-making Problem again, which had the following recursion:

$$change(n) = \begin{cases} 0 & \text{if } n = 0 \\ \min(change(n-1), change(n-6), change(n-7)) & \text{if } n > 0 \end{cases}$$

It should be clear that using the order of computation $0, 1, 2, 3, ...$, once we have computed e.g. $change(k)$, the subproblems $change(k-7), change(k-8), ...$ etc. are never used again.

Thus, we only need to save the value of 7 subproblems at a time. This $\Theta(1)$ memory usage is pretty neat compared to the $\Theta(K)$ usage needed to compute $change(K)$ otherwise.

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 will look at, for the few cases where these limits are still relevant.

7.3 Multidimensional DP

Now, we are going to look at a DP problem where our state consists of more than one variable. The example will demonstrate the importance of carefully choosing your DP parameters.

Ferry Loading

Swedish Olympiad in Informatics 2013, Online Qualifiers

A ferry is to be loaded with cars of different lengths, with a long line of cars currently queued up for a place. The ferry consists of four lanes, each of the same length. 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 safety margin of 1 meter between any two parked cars.

Given the length of the ferry and the length of the cars in the queue, compute the maximal amount of cars that can park if they choose the lanes optimally.

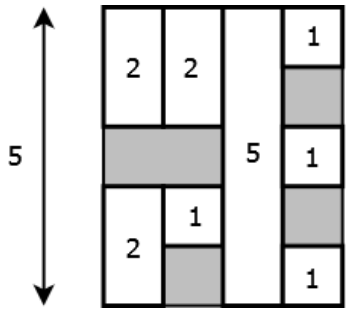


Figure 7.3: An optimal placement on a ferry of length 5 meters, of the cars with lengths $2, 1, 2, 5, 1, 1, 2, 1, 1, 2$ meters. Only the first 8 cars could fit on the ferry.

Input

The first line contains the number of cars $0 \leq N \leq 200$ and the length of the ferry $1 \leq L \leq 60$. The second line contains N integers, the length of the cars $1 \leq a_i \leq L$.

Output
Output a single integer – the maximal amount of cars that can be loaded on the ferry.

The ferry problem looks like a classical DP problem. It consists of a large number of similar choices. Each car has 4 choices – one of the lanes. If a car of length m chooses a lane, the remaining length of the chosen lane is reduced by $m + 1$ (due to the safety margin). After the first c cars have parked on the ferry, the only thing that has changed are the lengths of the ferry. As a simplification, we 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.

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:

Algorithm 7.4: Ferry Loading

```
int dp[201][62][62][62][62] = {-1};

int ferry(int c, vi used, const vi& A) {
  if (c == sz(A)) return 0;
  int& ans = dp[c][left[0]][left[1]][left[2]][left[3]];
  if (ans != -1) return ans;
  rep(i,0,4) {
    if (used[i] + A[i] + 1 > L + 1) continue;
    used[i] += A[i] + 1;
    ans = max(ans, ferry(c + 1, left, A) + 1);
    used[i] -= A[i] + 1;
  }
  return ans;
}
```

Unfortunately, memoizing this procedure would not be sufficient. The size of our memoization array is $200 \cdot 60^4 \approx 2.6 \cdot 10^9$, which needs many gigabytes of memory.

The trick in improving this is basically the same as the fundamental principle of DP. In DP, we reduce a set of choices to a smaller set of information, which represent the effects of those choices. This removes information which turned out to be redundant. In our case, we do not care about what lanes cars chose, only their remaining lengths. Our suggested solution still has some lingering redundancy though.

In Figure 7.3 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. We define $U(c)$ to be the total usage (i.e., lengths plus the safety margin) of the first c cars. Note that $U(c)$ is a strictly increasing function in c, meaning it is in

bijection to $[n]$. Let $u_1(c), u_2(c), u_3(c), u_4(c)$ be the usage of the four lanes individually in some given assignment. Then, we have that $U(c) = u_1(c) + u_2(c) + u_3(c) + u_4(c)$. The four terms on the right are four parameters in our memoization. The left term is not, but it has a bijection with c, which is a parameter in the memoization. Thus, we actually have a redundancy in our parameters. We can eliminate the parameter c, since it is uniquely defined given the values $u_1(c), u_2(c), u_3(c), u_4(c)$. This simplification leaves us with $60^4 \approx 13\,000\,000$ states, which is well within reason.

7.4 Subset DP

Another common theme of DP is subsets, where the state represents a subset of something. The subset DP is used in many different ways. Sometimes (as in Subsection 7.6.3), the problem itself is about sets and subsets. Another common usage is to reduce a solution which requires us to test permutations of something into instead constructing permutations iteratively, using DP to remember only what elements so far used in the permutation, and not the exact assignment.

Amusement Park

Swedish Olympiad in Informatics 2012, Online Qualifiers

Lisa has just arrived at an amusement park, and wants to visit each of the N attractions exactly once. For each attraction, there are two identical facilities at different locations in the park. Given the locations of all the facilities, determine which facility Lisa should choose for each attraction, in order to minimize the total distance she must walk. Originally, Lisa is at the entrance at coordinates $(0, 0)$. Lisa must return to the entrance once she has visited every attraction.

Input

The first line contains the integer $1 \leq N \leq 15$, the number of attractions Lisa wants to visit. Then, N lines follow. The i'th of these lines contains four integers $-10^6 \leq x_1, y_1, x_2, y_2 \leq 10^6$. These are the coordinates (x_1, y_1) and (x_2, y_2) for the two facilities of the i'th attraction.

Output

First, output the smallest distance Lisa must walk. Then, output N lines, one for each attraction. The i'th line should contain two numbers a and f – the i'th attraction Lisa visited (a number between 1 and N), and the facility she visited (1 or 2).

Consider a partial walk, where we have visited a set S of attractions and currently stand at coordinates (x, y). Then, any choice up to this point is irrelevant for the remainder of the problem, which suggests that these parameter S, x, y is a good DP state. Note that (x, y) only have at most 31 possibilities – two for each attraction,

plus the entrance at $(0, 0)$. Since we have at most 15 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 choosing what attraction to visit next. All in all, we get a complexity of $\Theta(N^2 2^N)$. When coding DP over subsets, we generally use bitsets to represent the subset, since these map very cleanly to integers (and therefore indices into a vector):

Algorithm 7.5: Amusement Park

```
double best(int at, int visited) {
  // 2N is the number given to the entrance point
  if (visited == (1<<N) - 1) return dist(at, 2*N);
  double ans = inf;
  rep(i,0,N) {
    if (visited&(1<<N)) continue;
    rep(j,0,2) {
      //2i + j is the number given to the j'th facility of the i'th attraction
      int nat = 2 * i + j;
      ans = min(ans, dist(at + nat) + best(nat, visited | (1<<i)));
    }
  }
  return ans;
}
```

7.5 Digit DP

Digit DP are a class of problems where we count numbers with certain properties that contain a large number of digits, up to a certain 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

Baltic Olympiad in Informatics 2013 – Antti Laaksonen

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, the number 16276 is palindrome-free whereas the number 17276 is not because it 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).

Your task is to calculate the total number of palindrome-free numbers in a given range.

Input
The input contains two numbers $0 \leq a \leq b \leq 10^{18}$.

Output
Your output should contain one integer: the total number of palindrome-free numbers in the range $a, a+1, ..., b-1, b$ (including a and b).

First, a common simplification when solving counting problems on intervals. Instead of computing the answer for the range $a, a+1, ..., b-1, b$, we will solve the problem for the intervals $[0, a)$ and $[0, b+1)$. The answer is then the answer for the second interval with the answer for the first interval removed. Our lower limit will then be 0 rather than a, which simplifies the solution.

Next up, we need an essential observation to turn the problem into a standard application of digit DP. Palindromes as general objects are very unwieldly in our situation. Any kind of iterative construction of numbers would have to bother with 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). This means that when constructing the answer recursively, we only need to care about the last two digits. 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 will solve the problem when the upper limit was $999...999$ – the sequence consisting of n nines. In this case, a simple recursive function will do the trick:

Algorithm 7.6: Palindrome-Free Numbers

```
ll sol(int at, int len, int b1, int b2) {
  if (at == len) return 1; // we have successfully constructed a number
  ll ans = 0;
  rep(d,0,10) {
    // this digit would create a palindrome
    if (d == b2 || d == b1) continue;
    // let -1 represent a leading 0, to avoid the palindrome check
        bool leadingZero = b2 == -1 && d == 0;
    ans += sol(at + 1, len, b2, leadingZero ? -1 : d);
  }
  return ans;
}

// initially, the empty number has length 0 and consists only of leading zeroes
sol(0, n, true, -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. In our case, they are given 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, with each state using a loop iterating only 10 times. Thus, it uses on the order of $1000n$ operations. In our problem, the upper limit has at most 19 digits. Thus, the solution only requires about 20 000 operations.

Once a solution has been formulated for this simple upper limit, extending it to a general upper limit is quite natural. First, we will save the upper limit as a sequence of digits L. Then, we need to differentiate between two cases in our recursive function. The partially constructed number is either equal to the corresponding partial upper limit, or it is less than the limit. In the first case, we are still constrained by the upper limit – the next digit of our number can not exceed the next digit of the upper limit. In the other case, the the upper limit is no longer relevant. If a prefix of our number is strictly lower than the prefix of the upper limit, our number can never exceed the upper limit.

This gives us our final solution:

Algorithm 7.7: Palindrome-Free Numbers, General Case

```
vector<int> L;

ll sol(int at, int len, bool limEq, int b1, int b2) {
    if (at == len) return 1;
    ll ans = 0;
    // we may not exceed the limit for this digit if equal to the prefix of the limit
    rep(d,0,(limEq ? L[at] + 1 : 10)) {
        if (d == b2 || d == b1) continue;
        // the next step will be equal to the prefix if it was now,
        // and our added digit was exactly the limit
        bool limEqNew = limEq && d == L[at];
        bool leadingZero = b2 == -1 && d == 0;
        ans += sol(at + 1, len, limEqNew, b2, leadingZero ? -1 : d);
    }
    return ans;
}

// initially, the number is equal to the prefix limit (both being the empty number)
sol(0, n, true, true, -1, -1);
```

7.6 Standard Problems

Many problems are variations of known DP problems, or have them as parts of their solutions. This section will walk you through a few of them.

7.6.1 Knapsack

The knapsack problem is one of the most common standard DP problem. The problem itself has countless variations. We will look at the "original" knapsack problem, with constraints making it suitable for a dynamic programming approach.

Knapsack

Given is a knapsack with an integer capacity C, and n different objects, each with an integer weight and value. Your task is to select a subset of the items with maximal value, such that the sum of their weights does not exceed the capacity of the knapsack.

Input

The integer C giving the capacity of the knapsack, and an integer n, giving the number of objects. This is followed by the n objects, given by their value v_i and weight w_i.

Output

Output the indicies of the chosen items.

We are now going to attempt to formulate an $O(nC)$ solution. As is often the case when the solution is a subset of something in DP solutions, we solve the problem by looking at the subset as a sequence of choices – to either include an item in the answer or not. In this particular problem, our DP state is rather minimalistic. Indeed, 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

$$K(c,i) = \max \begin{cases} K(c, i-1) \\ K(c - w_i, i-1) + v_i \quad \text{if } w_i \leq c \end{cases}$$

Translating this recursion into a bottom-up solution gives a rather compact algorithm (Algorithm 7.8).

Algorithm 7.8: Knapsack

```
procedure KNAPSACK(capacity C, items n, values V, weights W)
    best ← new int[n + 1][C + 1]
    fill best with −∞
    best[0][C] = 0
    for i from 0 to n − 1 do
        for j from 0 to C do
            if j ≥ W[i] then
```

```
        | | | | best[i + 1][j] ← max(best[i][j], best[i][j − W[i]] + V[i])
    return best
```

However, this only helps us compute the answer. The problem asks us to explicitly construct the subset. This step, i.e., tracing what choices we made to arrive at an optimal solution is called *backtracking*.

For this particular problem, the backtracking is relatively simple. One usually proceeds by starting at the optimal state, and then consider all transitions that lead to this state. Among these, the "best" one is picked. In our case, the transitions correspond to either choosing the current item, or not choosing it. Both lead to two other states which are simple to compute. In the first case, the state we arrived from must have the same value and capacity, while in the second case the value should differ by $V[i]$ and the weight by $W[i]$:

Algorithm 7.9: Knapsack Construction

```
procedure KnapsackConstruct(capacity C, items n, values V, weights W)
    best ← Knapsack(C, n, V, W)
    bestCap ← C
    for i from C to 0 do
        if best[N][i] > best[N][bestCap] then
            bestCap ← i
    for i from N to 1 do
        if W[i] ≤ bestCap then
            newVal ← best[i − 1][bestCap − W[i]]
            if newVal = best[i][bestCap] + V[i] then
                ans.add(i)
                bestCap ← bestCap − W[i]
    output ans
```

Exercise 7.1 — Kattis Exercise

Knapsack – knapsack

Walrus Weights – walrusweights

7.6.2 Longest Common Subsequence

Longest Common Subsequence

A sequence $a_1, a_2, ..., a_n$ has $c_1, c_2, ..., c_k$ as a *subsequence* if there exists indices $p_1 < p_2 < ..., < p_k$ such that $a_{p_i} = c_i$. For example, the sequence $\langle 1, 1, 5, 3, 3, 7, 5 \rangle$,

has $\langle 1, 3, 3, 5 \rangle$ as one of its sub-sequences.

Given two sequences $A = \langle a_1, a_2, ..., a_n \rangle$ and $B = \langle b_1, b_2, ..., b_m \rangle$, find the longest sequence $c_1, ..., c_k$ that is a subsequence of both A and B.

When dealing with DP problems on pairs of sequences, a natural subproblem is to solve the problem for all *prefixes* of A and B. For the subsequence problem, some reasoning about what a common subsequence is leads to a recurrence expressed in this way. Basically, we can do a case analysis on the last letter of the strings A and B. If the last letter of A is not part of a longest increasing subsequence, we can simply ignore it, and solve the problem on the two strings where the last letter of A is removed. A similar case is applicable when the last letter of B is not part of a longest increasing subsequence. A single case remains – when both the last letter of A and the last letter of B are part of a longest increasing subsequence. In this case, we argue that these two letters must correspond to the same letter c_i in the common subsequence. In particular, they must correspond to the final character of the subsequence (by the definition of a subsequence). Thus, whenever the two final letters are equal, we may have the case that they are the last letter of the subsequence, and that the remainder of the subsequence is the longest common subsequence of A and B with the final letter removed.

This yields a simple recursive formulation, which takes $\Theta(|A||B|)$ to evaluate (since each state takes $\Theta(1)$ to evaluate).

$$\text{lcs}(A, B, n, m) = \max \begin{cases} 0 & \text{if } n = 0 \text{ or } m = 0 \\ \text{lcs}(A, B, n-1, m) & \text{if } n > 0 \\ \text{lcs}(A, B, n, m-1) & \text{if } m > 0 \\ \text{lcs}(A, B, n-1, m-1) + 1 & \text{if } a_n = b_m \end{cases}$$

Exercise 7.2 — Kattis Exercise

Longest Increasing Subsequence – longincsubseq

7.6.3 Set Cover

In the *set cover* problem, we are given a family of subsets $S_1, S_2, ..., S_k$ of some larger set S of size n. We seek a minimal choice of subsets $S_{a_1}, S_{a_2}, ..., S_{a_l}$ such that

$$\bigcap_{i=1}^{l} S_{a_i} = S$$

i.e. we want to *cover* the set S by taking the union of as few of the subsets S_i as possible.

For small k and large n, we can solve the problem in $\Theta(n2^k)$, by simply testing each of the 2^k covers. In the case where we have a small n but k can be large, this becomes intractable. Instead, let us 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? Well, 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 computing the union of the current cover with 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 $\Theta(n4^n)$. Such a complexity is unacceptable 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 5.3. 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 will 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 13.4 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 our cover, but they all extend it in the same way. This sounds wasteful, and avoiding it probably the key to getting an asymptotic speedup.

It seems that we are missing some key function which can respond to the question: "is there some 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!

In fact, if we do our summing this time, we get:

$$\begin{aligned}\sum_{m=0}^{n} \binom{n}{m} 2^{n-m} &= \sum_{m=0}^{n} \binom{n}{n-m} 2^{n-m} \\ &= \sum_{m=0}^{n} \binom{n}{m} 2^{m} \\ &= 3^n\end{aligned}$$

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 assumption of our magical "can we extend a cover with a subset A?" function. Sometimes, this function may be quick 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, this 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 7.3 — Kattis Exercises

Square Fields (Easy) – squarefieldseasy

Square Fields (Hard) – squarefieldshard

7.7 Chapter Notes

write chapter notes

Chapter 8

Divide and Conquer

A recursive algorithm solves a problem by reducing it to smaller subproblems, hoping that their solutions can be used to solve the larger problem. So far, the subproblems we have considered have been "almost the same" as the problem at hand. We have usually recursed on a series of choices, where each recursive step made one choice. In particular, our subproblems often overlapped – solving two different subproblems required solving a common, third subproblem. In this chapter, we will take another approach altogether, by splitting our instance into large, disjoint (or almost disjoint parts) parts – dividing it – and combining their solutions – conquering it.

8.1 Inductive Constructions

Inductive construction problems compromise a large class of divide and conquer problems. The goal is often to construct something, such as a tiling of a grid, cycles in a graph and so on. In these cases, divide and conquer algorithms aim to reduce the construction of the whole object to instead constructing smaller parts which can be combined into the final answer. Such constructions are often by-products of mathematical induction proofs of the existence of such a construction. In the following example problems, it is not initially clear that the object we are asked to construct even exists.

Grid Tiling

In a square grid of side length 2^n, one unit square is blocked (represented by coloring it black). Your task is to cover the remaining $4^n - 1$ squares with *triominos*, L-shaped tiles consisting of three squares in the following fashion. The triominos can be rotated by any multiple of 90 deg (Figure 8.1).

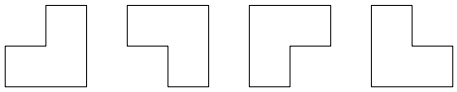


Figure 8.1: The four rotations of a triomino.

The triominos may not overlap each other, nor cover anything outside the grid. A valid tiling for $n = 2$ would be

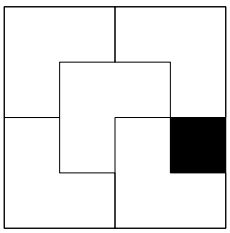


Figure 8.2: A possible tiling for $n = 2$.

Input

The input consists of three integers $1 \leq n \leq 8$, $0 \leq x < 2^n$ and $0 \leq y < 2^n$. The black square has coordinates (x, y).

Output

Output the positions and rotations of any valid tiling of the grid.

When tiling a $2^n \times 2^n$ grid, it is not immediately clear how the divide and conquer principle can be used. To be applicable, we must be able to reduce the problem into smaller instances of the same problem and combine them. The peculiar side length 2^n does hint about a possible solution. Aside from the property that $2^n \times 2^n - 1$ is evenly divisible by 3 (a necessary condition for a tiling to be possible), it also gives us a natural way of splitting an instance, namely into its 4 quadrants.

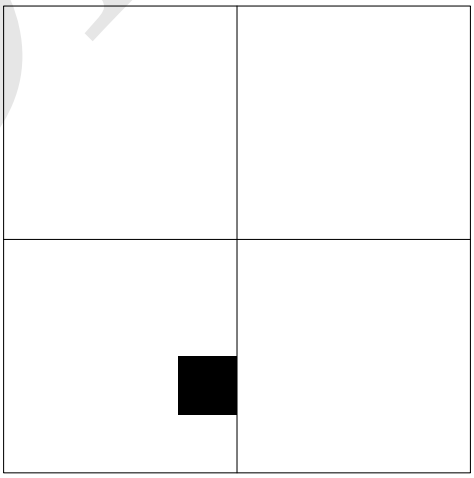


Figure 8.3: Splitting the $n = 3$ case into its four quadrants.

Each of these have the size $2^{n-1} \times 2^{n-1}$, which is also of the form we require of grids in the problem. The crux lies in that these four new grids does not comply with the input specification of the problem. While smaller and disjoint, three of them contain no black square, a requirement of the input. Indeed, a grid of this size without any black squares can not be tiled using triominos.

The solution lies in the trivial solution to the $n = 1$ case, where we can easily reduce the problem to four instances of the $n = 0$ case:

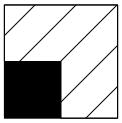


Figure 8.4: A solution to the $n = 1$ case.

In the solution, we use a single triomino which blocked a single square of each of the four quadrants. This gives us four trivial subproblems of size 1×1, where each grid has one blocked square. We can actually place such a triomino in every grid by placing it in the center (the only place where a triomino may cover three quadrants at once).

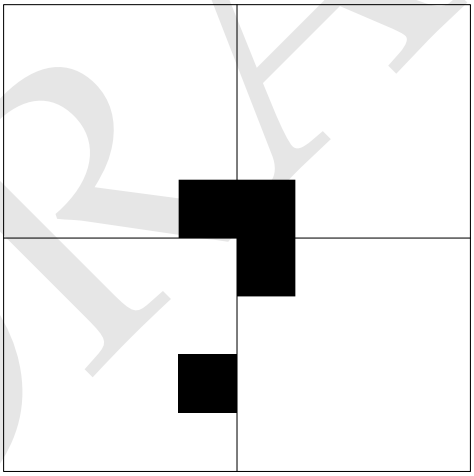


Figure 8.5: Placing a triomino in the corners of the quadrants without a black square.

After this transformation, we can now apply the divide and conquer principle. We split the grid into its four quadrants, each of which now contain one black square. This allows us to recursively solve four new subproblems. At some point, this recursion will finally reach the base case of a 1×1 square, which must already be filled.

Algorithm 8.1: Grid Tiling

procedure TILE(N, (B_x, B_y), (T_x, T_y))

```
if N = 0 then
    return
mid ← 2^(N−1)
blocked ← {(0, 0), (mid − 1, 0), (mid − 1, mid − 1), (0, mid − 1)}
if B_x ≥ mid and B_y ≥ mid then
    blockedQuad ← TOP_RIGHT
if B_x < mid and B_y ≥ mid then
    blockedQuad ← TOP_LEFT
if B_x < mid and B_y < mid then
    blockedQuad ← BOTTOM_LEFT
if B_x ≥ mid and B_y < mid then
    blockedQuad ← BOTTOM_RIGHT
place(T_x + mid, T_y + mid, blockedQuad)
tile(N − 1, blocked[0], T_x + mid, T_y + mid)
tile(N − 1, blocked[1], T_x, T_y + mid)
tile(N − 1, blocked[2], T_x, T_y)
tile(N − 1, blocked[3], T_x + mid, T_y)
```

The time complexity of the algorithm can be computed easily if we use the fact that each call to tile only takes $\Theta(1)$ time except for the four recursive calls. Furthermore, each call places exactly one tile on the board. Since there are $\frac{4^n-1}{3}$ tiles to be placed, the time complexity must be $\Theta(4^n)$.

Exercise 8.1

It is possible to tile such a grid with triominos colored red, blue and green such that no two triominos sharing an edge have the same color. Prove this fact, and give an algorithm to generate such a coloring.

Divisible Subset

Let $k = 2^n$. Given a set A of $2k - 1$ integers, find a subset S of size exactly k such that

$$\sum_{x \in S} x$$

is a multiple of k.

Input

The input contains an integer $1 \leq k \leq 2^{15}$ that is a power of two, followed by the k elements of A.

Output

Output the k elements of S.

When given a problem, it is often a good idea to solve a few small cases by hand. This applies especially to this kind of construction problems, where constructions for small inputs often shows some pattern or insight into how to solve larger instances. The case $k = 1$ is not particularly meaningful, since it is trivially true (any integer is a multiple of 1). When $k = 2$, we get an insight which might not seem particularly interesting, but is key to the problem. We are given $2 \cdot 2 - 1 = 3$ numbers, and seek two numbers whose sum is even. Given three numbers, it must have either two numbers which both are even, or two odd numbers. Both of these cases yield a pair with an even sum.

It turns out that this construction generalizes to larger instances. Generally, it is easier to do the "divide" part of a divide and conquer solution first, but in this problem we will do it the other way around. The recursion will follow quite naturally after we attempt to find a way in combining solutions to the smaller instance to a larger one.

We will lay the the ground work for a reduction of the case $2k$ to k. First, assume that we could solve the problem for a given k. The larger instance then contains $2(2k - 1) = 4k - 1$ numbers, of which we seek $2k$ numbers whose sum is a multiple of $2k$. This situation is essentially the same as for the case $k = 2$, except everything is scaled up by k. Can we scale our solution up as well?

If we have three **sets of** k **numbers** whose respective sums are all **multiples of** k, we can find **two sets of** k **numbers** whose total sum is **divisible by** $2k$. This construction essentially use the same argument as for $k = 2$. If the three subsets have sums ak, bk, ck and we wish to find two whose sum is a multiple of $2k$, this is the same as finding two numbers of a, b, c whose sum is a multiple of 2. This is possible, according to the case $k = 2$.

A beautiful generalization indeed, but we still have some remnants of wishful thinking we need to take care of. The construction assumes that, given $4k - 1$ numbers, we can find three sets of k numbers whose sum are divisible by k. We have now come to the recursive aspect of the problem. By assumption, we could solve the problem for k. This means we can pick any $2k - 1$ of our $4k - 1$ numbers to get our first subset. The subset uses up k of our $4k - 1$ numbers, leaving us with only $3k - 1$ numbers. We keep going, and pick any $2k - 1$ of these numbers and recursively get a second subset. After this, $2k - 1$ numbers are left, exactly how many we need to construct our third subset.

The division of the problem was thus into *four* parts. Three subsets of k numbers, and one set of $k - 1$ which we throw away. Coming up with such a division essentially required us to solve the combination part first with the generalizing of the case $k = 2$.

Algorithm 8.2: Divisible Subset

```
void fillWith(hashset<int>& toFill, hashset<int>& from, int size) {
  while (sz(from) < size) {
    from.insert(toFill.begin());
    toFill.erase(toFill.begin());
  }
}

hashset<int> divisbleSubset(int k, hashset<int> A) {
  if (k == 1) return nums;

  hashset<int> part;
  // Find three subsets of size k/2 with sums divisible by k/2
  fillWith(part, A, k - 1);
  hashset<int> pa = divisibleSubset(k / 2, A);
  A.erase(all(pa));

  fillWith(part, A, k - 1);
  hashset<int> pb = divisibleSubset(k / 2, A);
  A.erase(all(pb));

  fillWith(part, A, k - 1);
  hashset<int> pc = divisibleSubset(k / 2, A);
  A.erase(all(pc));

  // Choose two who sum to k/2
  int as = accumulate(all(pa), 0);
  int bs = accumulate(all(pa), 0);
  int cs = accumulate(all(pa), 0);

  hashset<int> ans;
  if ((as + bs) % k == 0) {
    ans.insert(all(pa));
    ans.insert(all(pb));
  } else if ((as + cs) % k == 0) {
    ans.insert(all(pa));
    ans.insert(all(pc));
  } else {
    ans.insert(all(pb));
    ans.insert(all(pc));
  }
  return ans;
}
```

This complexity is somewhat more difficult to analyze. Now, each call to divisibleSubset takes linear time to k, and makes 3 recursive calls with $k/2$. Thus, the complexity obeys the recurrence $T(k) = 3T(k/2) + \Theta(k)$. By the master theorem, this has complexity $\Theta(k^{\log_2 3}) = \Theta(3^n)$ where $k = 2^n$.

Exercise 8.2

What happens if we, when solving the problem for some k, construct $k - 1$ pairs of integers whose sum are even, throw away the remaining element, and scale the problem down by 2 instead? What is the complexity then?

Exercise 8.3

The problem can be solved using a similar divide and conquer algorithm for *any* k, not just those which are powers of 2[1]. In this case, those k which are prime numbers can be treated as base cases. How is this done for composite k? What is the complexity?

Exercise 8.4

The knight piece in chess can move in 8 possible ways (moving 2 steps in any one direction, and 1 step in one of the two perpendicular directions). A closed tour exists for an 8×8 grid.

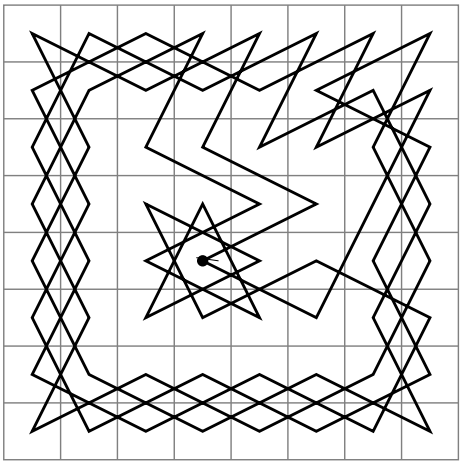


Figure 8.6: A closed tour on an 8×8 grid.

Give an algorithm to construct a tour for any $2^n \times 2^n$ grid with $n \geq 3$.

Exercise 8.5

An n-bit *Gray code* is a sequence of all 2^n bit strings of length n, such that two adjacent bit strings differ in only one position. The first and last strings of the

[1]This result is known as the Erdős–Ginzburg–Ziv theorem

sequence are considered adjacent. Possible Gray codes for the first few n are

$$n = 1\text{: } 0\ 1$$

$$n = 2\text{: } 00\ 01\ 11\ 10$$

$$n = 3\text{: } 000\ 010\ 110\ 100\ 101\ 111\ 011\ 001$$

Give an algorithm to construct an n-bit Gray code for any n.

Exercise 8.6 — Kattis Problems

Bell Ringing – bells

8.2 Merge Sort

Merge sort is a sorting algorithm which uses divide and conquer. It is rather straightforward, and works by recursively sorting smaller and smaller parts of the array. When sorting an array by *dividing it into parts* and combining their solution, there is an obvious candidate for how to perform this partitioning. Namely, splitting the array into two halves and sorting them. When splitting an array in half repeatedly, we will eventually reach a rather simple base case. An array containing a single element *is already sorted*, so it is trivially solved. If we do so recursively, we get the recursion tree in Figure 8.7. Coding this recursive split is easy.

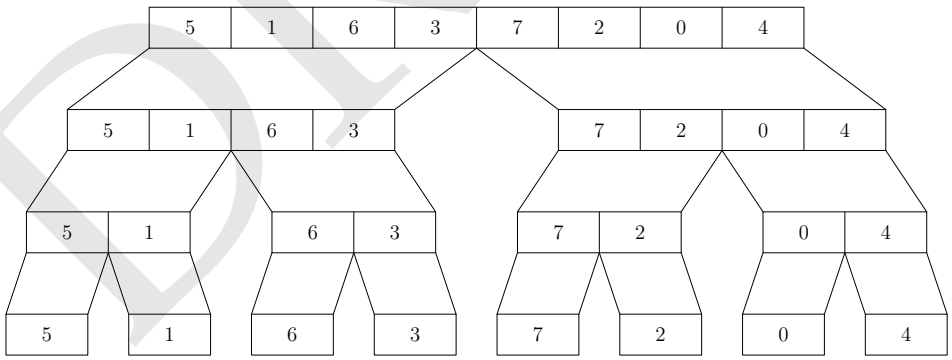


Figure 8.7: The recursion tree given when performing a recursive split of the array $[5, 1, 6, 3, 7, 2, 0, 4]$.

When we have sorted the two halves, we need to combine them to get a sorted version of the entire array. The procedure to do this is based on a simple insight. If an array A is partitioned into two smaller arrays P_1 and P_2, the smallest value of A must be either the smallest value of P_1 or the smallest value of P_2. This insight gives rise to a

simple iterative procedure, where we repeatedly compare the smallest values of P_1 and P_2, extract the smaller one of them, and append it to our sorted array.

Algorithm 8.3: Merge Sort

```
void sortVector(vector<int>& a, int l, int r){
  int m = (l + r)/2;
  sortVector(a, l, m); // sort left half
  sortVector(a, m, r); // sort right half

  //combine the answers
  vector<int> answer;
  int x = l;
  int y = m;
  while(x < m || y < r){
    if(x == m || (y < r && a[y] < a[x])){
      answer.push_back(a[y]);
      y++;
    } else {
      answer.push_back(a[x]);
      x++;
    }
  }
  rep(i,l,r) a[i] = answer[i - l];
}
```

To compute the complexity, consider the recursion tree in Figure 8.7. We make one call with 8 elements, two calls with 4 elements, and so on. Further, the combining procedure takes $\Theta(l)$ time for a call with l elements. In the general case of $n = 2^k$ elements, this means merge sort takes time

$$\sum_{i=0}^{k} 2^i \cdot \Theta(2^{k-i}) = \Theta(k2^k)$$

Since $k = \log_2 n$, this means the complexity is $\Theta(n \log n)$.

Exercise 8.7

Our complexity analysis assumed that the length of the array is a power of 2. The complexity is the same in the general case. Prove this fact.

Exercise 8.8

Given an array A of size n, we call the pair $i < j$ an *inversion* of A if $A[i] > A[j]$.

Adapt the merge sort algorithm to count the number of inversions of an array in $\Theta(n \log n)$.

8.3 Binary Search

The *binary search* is a common component of other solution. Given a number L and a **non-decreasing** function $f : \mathbb{R} \to \mathbb{R}$, we wish to find the greatest x such that $f(x) \leq L$. Additionally, we need two numbers *lo* and *hi*, such that $f(lo) \leq L < f(hi)$.

It is not difficult to see how we solve this problem. Consider the number $mid = \frac{lo+hi}{2}$. If $f(lo) \leq L$, then we know that the answer must lie somewhere in the interval $[lo, hi)$. On the other hand, $L < f(lo)$ gives us a better upper bound on the answer, which must be contained in $[lo, hi)$. Computing $f(mid)$ allowed us to halve the interval in which the answer can be.

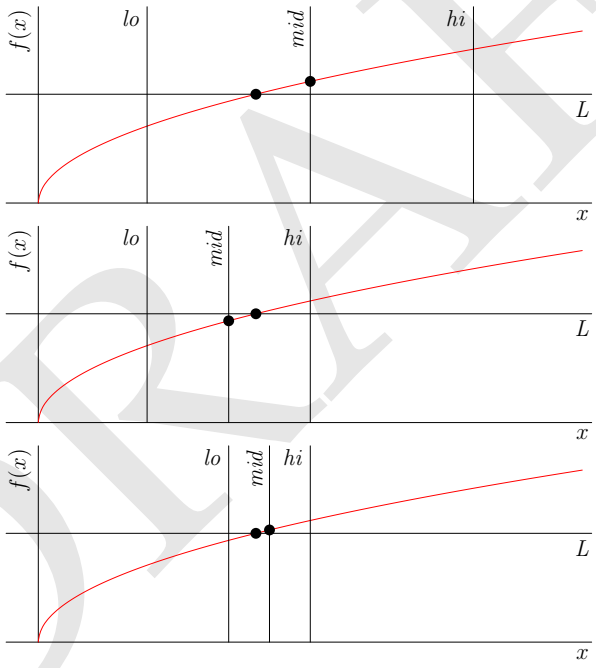


Figure 8.8: Two iterations of binary search.

We can repeat this step until we get close to x.

Algorithm 8.4: Binary Search

```
1  const double precision = 1e-7;
2
3  double binarySearch(double lo, double hi, double lim) {
4    while (hi - lo > precision) {
5      double mid = (lo + hi) / 2;
```

```
    if (lim < f(mid)) hi = mid;
    else lo = mid;
  }
  return lo;
}
```

Notice that we are actually computing an approximation of x within some given precision (10^{-7} in the example implementation), but this is often useful enough.

Competitive Tip

Remember that the double-precision floating point type only have a precision of about 10^{15}. If the limits in your binary search are on the order of 10^x, this means that using a binary search precision of something smaller than 10^{x-15} may cause an infinite loop. This happens because the difference between *lo* and the next possible double is actually larger than your precision.

As an example, the following parameters cause our binary search with precision 10^{-7} to fail.

```
double f(double x) {
  return 0;
}

double lo = 1e12;
double hi = nextafter(lo, 1e100);
binarySearch(lo, hi, 0);
```

An alternative when dealing with limit precision is to perform binary search a fixed number of iterations:

```
double binarySearch(double lo, double hi, double lim) {
  rep(i,0,60) {
    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 we want. 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

$$\frac{hi - lo}{2^c} \leq p$$

$$\frac{hi - lo}{p2^c} \leq 1$$

$$\frac{hi - lo}{p} \leq 2^c$$

$$\log_2 \frac{hi - lo}{p} \leq c$$

For example, if we have an interval of size 10^9 which we wish to binary search down to 10^{-7}, this would require $\log_2 10^{16} = 54$ iterations of binary search.

Now, let us study some applications of binary search.

8.3.1 Optimization Problems

Cutting Hot Dogs

The finals of the Swedish Olympiad in Informatics is soon to be arranged. During the competition the participants must have a steady supply of hot dogs to get enough energy to solve the problems. Being foodies, the organizing committee couldn't just *buy* ready-to-cook hot dogs – they have to make them on their own!

The organizing committee has prepared N long rods of hot dog. These rods have length $a_1, a_2, ..., a_N$ centimeters. It is now time to cut up these rods into actual hot dogs. A rod can be cut into any number k of hot dogs using $k - 1$ cuts. Note that parts of one rod cannot be combined with parts of another rod, due to the difference in taste. It is allowed to leave leftovers from a rod that will not become a hot dog.

In total, the contest has M participants. Each contestant should receive a single hot dog, and all of their hot dogs should be of the same length. What is the maximal hot dog length L the committee can use to cut M hot dogs of length L?

Input

The first line contains two integers $1 \leq N \leq 10\,000$ and $1 \leq M \leq 10^9$.

The next line contains N real numbers $0 < a_1, a_2, ..., \leq 10^6$, the length of the hot dog rods in centimeters.

Output

Output a single real number – the maximal hot dog length possible in centimeters. Any answer with a relative or absolute error of 10^{-6} is acceptable.

This problem is not only an optimization problem, but a *monotone* one. The monotonicity lies in that while we only ask for a certain maximal hot dog length, all lengths below it would also work (in the sense of being able to have M hot dogs cut of this length), while all lengths above the maximal length produce less than M hot dogs. Monotone optimization problems makes it possible to remove the optimization aspect by inverting the problem. Instead of asking ourselves what the maximum length is,

we can instead ask how many hot dogs $f(x)$ can be constructed from a given length x. After this inversion, the problem is now on the form which binary search solves: we wish to find the greatest x such that $f(x) = M$ (replacing $\leq$ with $=$ is equivalent in the cases where we know that $f(x)$ assume the value we are looking for). We know that this length is at most $\max_i a_i \leq 10^6$, which gives us the interval $(0, 10^6]$ to search in.

What remains is to actually compute the function $f(x)$. In our case, this can be done by considering just a single rod. If we want to construct hot dogs of length x, we can get at most $\lfloor \frac{a_i}{x} \rfloor$ hot dogs from a rod of length a_i. Summing this for every rod gives us our solution.

Algorithm 8.5: Cutting Hot Dogs

```
procedure CountRods(lengths A, minimum length M)
    dogs ← 0
    for each l ∈ A do
        dogs ← dogs + ⌊l/M⌋
    return dogs
procedure HotDogs(lengths A, participants M)
    L ← 0, H ← 10^6
    while H − L ← 10^−7 do
        mid ← (L + H)/2
        if CountRods(A, mid) < M then
            H ← mid
        else
            L ← mid
    output L
```

The key to our problem was that the number of hot dogs constructible with a length x was monotonically decreasing with x. It allowed us to perform binary search on the answer, a powerful technique which is a component of many optimization problems. In general, it is often easier to determine *if* an answer is acceptable, rather than computing a maximal answer.

8.3.2 Searching in a Sorted Array

The classical application of binary search is to find the position of an element x in a sorted array A of length n. Applying binary search to this is straightforward. At first, we know nothing about location of the element – its position could be anyone of $[0, n)$. So, we consider the middle element, $\texttt{mid} = \lfloor \frac{n}{2} \rfloor$, and compare $A[\texttt{mid}]$ to x. Since A is sorted, this leaves us with three cases:

- $A[\texttt{mid}] = x$ – and we are done
- $x < A[\texttt{mid}]$ – since the array is sorted, any occurrence of x must be to the left of $\texttt{mid}$
- $A[\texttt{mid}] < x$ – by the same reasoning, x can only lie to the right of $\texttt{mid}$.

The last two cases both halve the size of the sub-array which x could be inside. Thus, after doing this halving $\log_2 n$ times, we have either found x or can conclude that it is not present in the array.

Algorithm 8.6: Search in Sorted Array

```
procedure SEARCH(array A, target x)
    L ← 0, H ← |A|
    while H − L > 0 do
        mid ← ⌊(L + H)/2⌋
        if A[mid] = x then
            return mid
        else if x < A[mid] then
            H = mid
        else
            L = mid + 1
    return −1
```

Competitive Tip

When binary searching over discrete domains, care must be taken. Many bugs have been caused by improper binary searches[2].

The most common class of bugs is related to the endpoints of your interval (i.e. whether they are inclusive or exclusive). Be explicit regarding this, and take care that each part of your binary search (termination condition, midpoint selection, endpoint updates) use the same interval endpoints.

Exercise 8.9 — Kattis Problems

Ballot Boxes – ballotboxes

[2]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`

8.3.3 Generalized Binary Search

Binary search can also be used to find *all* points where a monotone function changes value (or equivalently, all the intervals on which a monotone function is constant). Often, this is used in problems on large sequences (often with $n = 100\,000$ elements), which can be solved by iterating through all contiguous sub-sequences in $\Theta(n^2)$ time.

Or Max

Petrozavodsk Winter Training Camp 2015

Given is an array A of integers. Let

$$B(i,k) = A[i] \mid A[i+1] \mid ... \mid A[i+k-1]$$

i.e. the bitwise or of the k consecutive numbers starting with the i'th,

$$M(i,k) = \max\{A[i], A[i+1], ..., A[i+k-1]\}$$

i.e. the maximum of the k consecutive numbers starting with the i'th, and

$$S(i,k) = B(i,k) + M(i,k)$$

For each $1 \leq k \leq n$, find the maximum of $S(i,k)$.

Input

The first line contains the length $1 \leq n \leq 10^5$ of A.

The next and last line contains the n values of A ($0 \leq A[i] < 2^{16}$), separated by spaces.

Output

Output n integers, the maximum values of $S(i,k)$ for $k = 1, 2, ..., n$.

As an example, consider the array in Figure 8.9. The best answer for $k = 1$ would be $S(0,1)$, with both maximal element and bitwise or 5, totaling 10. For $k = 2$, we have $S(6,2) = 7 + 4 = 11$.

$i=0$	$i=1$	$i=2$	$i=3$	$i=4$	$i=5$	$i=6$	$i=7$	$i=8$	$i=9$
5	1	4	2	2	0	4	3	1	2
101	001	100	010	010	000	100	011	001	010

Figure 8.9: Example array, with the numbers additionally written in binary.

This problem can easily be solved in $\Theta(n^2)$, by computing every $S(i,k)$ iteratively. We can compute all the $B(i,k)$ and $M(i,k)$ using the recursions

$$B(i,k) := \begin{cases} 0 & \text{if } k = 0 \\ B(i,k-1) \mid A[i+k-1] & \text{if } k > 0 \end{cases}$$

$$M(i,k) = \begin{cases} 0 & \text{if } k = 0 \\ \max\{M(i,k-1), A[i+k-1]\} & \text{if } k > 0 \end{cases}$$

by looping over k, once we fix an i. With $n = 100\,000$, this approach is too slow.

The difficulty of the problem lies in $S(i,k)$ consisting of two basically unrelated parts – the maximal element and the bitwise or of a segment. When maximizing sums of unrelated quantities that put constraints on each other, brute force often seems like a good idea. This is basically what we did in the Buying Books problem (Section 5.4), where we minimized the sum of two parts (postage and book costs) which constrained each other (buying a book forced us to pay postage to its store) by brute forcing over one of the parts (the set of stores to buy from). Since the bitwise or is much more complicated than the maximal element – it is decided by an entire interval rather than a single element – we are probably better of doing brute force over the maximal element. Our brute force will consist of fixing which element is our maximal element, by assuming that $A[m]$ is the maximal element.

With this simplification in hand, only the bitwise or remains. We could now solve the problem by looping over all the left endpoints of the interval and all the right endpoints of the interval. At a first glance, this seems to actually worsen the complexity. Indeed, this takes quadratic time for each m (on average), resulting in a cubic complexity.

This is where we use our new technique. It turns out that, once we fix m, there are only a few possible values for the bitwise or of the intervals containing the m'th element, Any such interval $A[l], A[l+1], ..., A[m-1], A[m], A[m+1], ..., A[r-1], A[r]$ can be split into two parts: one to the left, $A[l], A[l+1], ..., A[i-1], A[i]$, and one to the right, $A[i], A[i+1], ..., A[r-1], A[r]$. The bitwise or of either of these two parts is actually a monotone function (in their length), and can only assume at most 16 different values!

Studying Figure 8.10 gives a hint about why. The first row shows the binary values of the array, with $m = 6$ (our presumed maximal element) marked. The second row shows the binary values of the bitwise or of the interval $[i, m]$ or $[m, i]$ (depending on whether m is the right or left endpoint). The third line shows the decimal values of the second row.

For example, when extending the interval $[2, 6]$ (with bitwise or 110) to the left, the new bitwise or will be 110|001. This is the only way the bitwise or can change – when

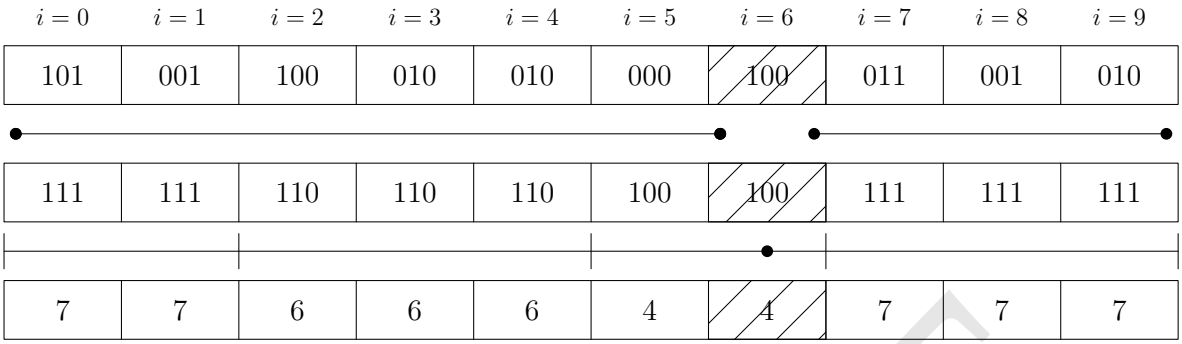


Figure 8.10: The bitwise or of the left and right parts, with an endpoint in $m = 6$

the new value includes bits which so far have not been set. Obviously, this can only happen at most 16 times, since the values in A are bounded by 2^{16}.

For a given m, this gives us a partition of all the elements, by the bitwise or of the interval $[m, i]$. In Figure 8.10, the left elements will be partitioned into $[0, 1]$, $[2, 4]$, $[5, 6]$. The right elements will be partitioned into $[6, 6]$, $[7, 9]$. These partitions are everything we need to compute the final.

For example, if we pick the left endpoint from the part $[2, 4]$ and the right endpoint from the part $[7, 9]$, we would get a bitwise or that is $6 \mid 7 = 7$, of a length between 4 and 8, together with the 4 as the presumed maximal element. For each maximal element, we get at most $16 \cdot 16$ such choices, totaling less than $256N$ such choices. From these, we can compute the final answer using a simple sweep line algorithm.

8.4 Karatsuba's algorithm

Karatsuba's algorithm was developed by Russian mathematician Anatoly Karatsuba and published in the early 1960's. It is one of the earliest examples of a divide and conquer algorithm, and is used to quickly multiply large numbers. While multiplying small numbers (i.e. those that fit in the integer types of your favorite programming language) is considered to be a $\Theta(1)$ operation, this is not the case for arbitrarily large integers. We will look at Karatsuba as a way of multiplying polynomials, but this can easily be extended to multiplying integers.

Polynomial Multiplication

Given two n-degree polynomials (where n can be large) $p(x) = \sum_{i=0}^{n} x^i a_i$ and

$q(x) = \sum_{i=0}^{n} x^i b_i$ compute their product

$$(pq)(x) = \sum_{i=0}^{2n} x^i (\sum_{j=0}^{i} a_j b_{i-j})$$

The naive multiplication algorithm evaluates this using $\Theta(n^2)$ multiplications (e.g. by two nested loops).

It turns out we can do this faster, using a recursive transformation. If we split the numbers p and q into their upper and lower $k = \frac{n}{2}$ coefficients (if n is odd, we pad the polynomials with a leading zero), so that $p(x) = p_l(x)x^k + p_r(x)$ and $q(x) = q_l(x)x^k + q_r(x)$, their product is equal to

$$\begin{aligned}(pq)(x) &= (p_l(x)x^k + p_r(x))(q_l(x)x^k + q_r(x)) \\ &= p_l(x)q_l(x)x^n + (p_l(x)q_r(x) + p_r(x)q_l(x))x^k + p_r(x)q_r(x)\end{aligned}$$

This formula requires multiplying 4 pairs of k-degree polynomials instead, which we can recursively compute, resulting in the time complexity recurrence $T(n) = 4T(\frac{n}{2}) + \Theta(n)$. Using the master theorem gives us the solution $T(n) = \Theta(n^2)$, which is no faster than the naive multiplication.

However, we can compute $p_l(x)q_r(x) + p_r(x)q_l(x)$ using only one multiplication instead of two. Both of these terms are part of the expansion of pq, which is only one multiplication. That particular multiplication is on n-degree polynomials, but it is not difficult to see how we can reduce it to a single k-degree multiplication. We simply throw away the multiplicative factors that makes $p_l x^k + p_r$ and $q_l x^k + q_r$ an n-degree polynomials:

$$(p_l(x) + p_r(x))(q_l(x) + q_r(x)) = p_l(x)q_l(x) + p_l(x)q_r(x) + p_r(x)q_l(x) + p_r(x)q_r(x)$$

so that

$$p_l(x)q_r(x) + p_r(x)q_l(x) = (p_l(x) + p_r(x))(q_l(x) + p_r(x)) - p_l(x)q_l(x) - p_r(x)q_r(x)$$

This means we only need to compute three k-degree multiplications: $(p_l(x)+p_r(x))(q_l(x)+q_r(x)), p_l(x)q_l(x), p_r(x), q_r(x)$ Our time complexity recurrence is then reduced to $T(n) = 3T(\frac{n}{2}) + O(n)$, which by the master theorem is $\Theta(n^{\log_2 3}) \approx \Theta(n^{1.585})$.

Exercise 8.10

Polynomial Multiplication 2 – polymul2

8.5 Chapter Notes

Chapter 9

Data Structures

Solutions to algorithmic problems consists of two constructs – algorithms and ***data structures***. Data structures are used to organize the data that the algorithms operate on. For example, the array is such an algorithm.

Many data structures has been developed to handle particular common operations we need to perform on data quickly. In this chapter, we will study a few such structures.

9.1 Disjoint Sets

In the *Connectivity Problem*, we want to determine whether two vertices in the graph are in the same connected component. This problem can be solved using a Depth-First Search (Section 10.2). For now, we will instead focus on an extension of this problem called the *Dynamic Connectivity Problem, Additions only*[1], where we may also alter the graph by adding edges.

Dynamic Connectivity, Additions Only

Given a graph G which initially consists of V disconnected vertices, you will receive Q queries of two types:

1. take two vertices v and w, and add an edge between them
2. determine whether vertices v and w are currently in the same component

This problem can easily be solved in $O(Q^2)$, by after each query performing a DFS to partition the graph into its connected components. Since the graph has at most Q

[1]The Dynamic Connectivity problem where edges may also be removed can be solved by a data structured called a *Link-Cut tree*, which is not discussed in this book.

edges, a DFS takes $O(Q)$ time.

We can improve this by explicitly computing the connected components at each step. Originally, we have V components of a single vertex each. When adding an edge between two vertices that are in different components, these components will now merge. Note that it is irrelevant which vertices we are adding an edge between – it is only important what components the vertices belong to. When merging two connected components to one, we iterate through all the vertices of one component and add them to the other. Since we make at most V joins of different components, and the components involved contain at most V vertices, these queries only take $O(V^2)$ time. Determining whether two vertices are in the same component can then be done in $O(V)$, leaving us with a total complexity of $O(V(V+Q))$. We can speed this up further by using a simple look-up table $comp[v]$ for the vertices, which stores some identifier for the component a vertex is in. We can then respond to a connectivity query in $\Theta(1)$ by comparing the value of $comp[v]$ and $comp[w]$. Our complexity is then $O(V^2+Q)$ instead.

Finally, we will use a common trick to improve the complexity to $O(V \log V + Q)$ instead. Whenever we merge two components of size a and b, we can do this in $O(\min(a,b))$ instead of $O(a+b)$ by merging the smaller component into the larger component. Any individual vertex can be part of the smaller component at most $O(\log V)$ times. Since the total size of a component is always at least twice the size of the smaller component, this means that if a vertex is merged as part of the smaller component k times the new component size must be at least 2^k. This cannot exceed V, meaning $k \leq \log_2 V$. If we sum this up for every vertex, we arrive at the $O(V \log V + Q)$ complexity.

Algorithm 9.1: Disjoint Set

```
struct DisjointSets {

  vector<vector<int>> components;
  vector<int> comp;
  DisjointSets(int elements) : components(elements), comp(elements) {
    iota(all(comp), 0);
    for (int i = 0; i < elements; ++i) components[i].push_back(i);
  }

  void unionSets(int a, int b) {
    a = comp[a]; b = comp[b];
    if (a == b) return;
    if (components[a].size() < components[b].size()) swap(a, b);
    for (int it : components[b]) {
      comp[it] = a;
      components[a].push_back(it);
    }
  }
```

```
19
20  };
```

A somewhat faster[2] version of this structure instead performs the merges of components lazily. When merging two components, we do not update $comp[v]$ for every vertex in the smaller component. Instead, if a and b are the representative vertices of the smaller and the larger component, we only merge a by setting $comp[a] = b$. However, we still need to perform the merges. Whenever we try to find the component a vertex lies in, we perform all the merges we have stored so far.

Algorithm 9.2: Improved Disjoint Set

```
1  struct DisjointSets {
2
3    vector<int> comp;
4    DisjointSets(int elements) : comp(elements, -1) {}
5
6    void unionSets(int a, int b) {
7      a = repr(a); b = repr(b);
8      if (a == b) return;
9      if (-comp[a] < -comp[b]) swap(a, b);
10     comp[a] += comp[b];
11     comp[b] = a;
12   }
13
14   bool is_repr(int x) { return comp[x] < 0; }
15
16   int repr(int x) {
17     if (is_repr(x)) return x;
18     while (!is_repr(comp[x])) {
19       comp[x] = comp[comp[x]];
20     }
21     return comp[x];
22   }
23
24 };
```

However, it turns out we can sometimes perform many merges at once. Consider the case where we have k merges lazily stored for some vertex v. Then, we can perform *all* the merges of $v, comp[v], comp[comp[v]], \ldots$ at the same time since they all have the same representative: the representative of v.

[2] With regards to actual time, not asymptotically.

Algorithm 9.3: Performing Many Merges

```
int repr(int x) {
  if (comp[x] < 0) return x;
  int par = comp[x];
  comp[x] = repr(par);
  return comp[x];
}
```

Intuitively, this should be faster. After all, we would be performing performing at least $k + (k-1) + (k-2) + \cdots = O(k^2)$ merges in $O(k)$ time, only for those vertices. If there were more vertices part of the components merged, this number grows even more.

It turns out that this change improves the complexity asymptotically.

9.2 Range Queries

Problems often ask us to compute some expression based on some interval of an array. The expressions are usually easy to compute in $\Theta(len)$ where *len* is the length of the interval. We will now study some techniques that trade much faster query responses for a bit of memory and precomputation time.

9.2.1 Prefix Precomputation

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 $S(L, R) = a_L + a_{L+1} + \cdots + a_{R-1}$.

Computing the sums naively would require $\Theta(N)$ worst-case time per query if the intervals are large, for a total complexity of $\Theta(NQ)$. If $Q = \Omega(N)$ we can improve this to $\Theta(N^2 + Q)$ by *precomputing* all the answers. To do this in quadratic time, we use the recurrence

$$S(L, R) = \begin{cases} 0 & \text{if } L = R \\ S(L, R-1) + a_{R-1} & \text{otherwise} \end{cases}$$

Using this recurrence we can compute the sequence $S(L, L), S(L, L+1), S(L, L+2), \ldots, S(L, N)$ in average $\Theta(N)$ time for every L. This gives us the $\Theta(N^2 + Q)$ complexity.

If the function we are computing has an inverse, we can speed this precomputation up a bit. Assume that we have computed the values $P(R) = a_0 + a_1 + \cdots + a_{R-1}$, i.e. the prefix sums of a_i. Since this function is invertible (with inverse $-P(R)$), we can compute $S(L, R) = P(R) - P(L)$. Basically, the interval $[L, R)$ consists of the prefix $[0, R)$ with the prefix $[0, L)$ removed. As addition is invertible, we could simply remove the latter prefix $P(L)$ from the prefix $P(R)$ using subtraction. Indeed, expanding this expression shows us that

$$\begin{aligned} P(R) - P(L) &= (a_0 + a_1 + \cdots + a_{R-1}) - (a_0 + a_1 + \cdots + a_{L-1}) \\ &= a_L + \cdots + a_{R-1} = S(L, R) \end{aligned}$$

Algorithm 9.4: Interval Sums

```
procedure PREFIXES(sequence A)
    P ← new int[|A| + 1]
    for i = 0 to |A| − 1 do
        P[i + 1] ← P[i] + A[i]
    return P
procedure INTERVALSUM(interval [L, R), prefix table P)
    return P[R] − P[L]
```

This same technique works for any invertible operation.

Exercise 9.1

The above technique does not work straight-off for non-commutative operations. How can it be adapted to this case?

9.2.2 Sparse Tables

The case where a function does not have an inverse is a bit more difficult.

Interval Minimum

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 $M(L, R) = min(a_L, a_{L+1}, \ldots, a_{R-1})$.

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 $\Theta(N \log N)$.

Algorithm 9.5: Sparse Table

```
vector<vi> ST(const vi& A) {
  vector<vi> ST(__builtin_popcount(sz(A)), 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

$$[L, L + 2^{k_1})$$

$$[L + 2^{k_1}, L + 2^{k_1} + 2^{k_2})$$

$$\cdots$$

$$[L + 2^{k_1} + \cdots + 2^{k_{l-1}}, L + len)$$

together cover $[L, L + len)$. Thus we can compute the minimum of $[L, L + len)$ as the minimum of $\log_2 len$ intervals.

Algorithm 9.6: 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;
```

```
11 }
```

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(b - a);
  return min(jmp[maxLen][L], jmp[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.

9.2.3 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 8.2). In Figure 9.1, 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*.

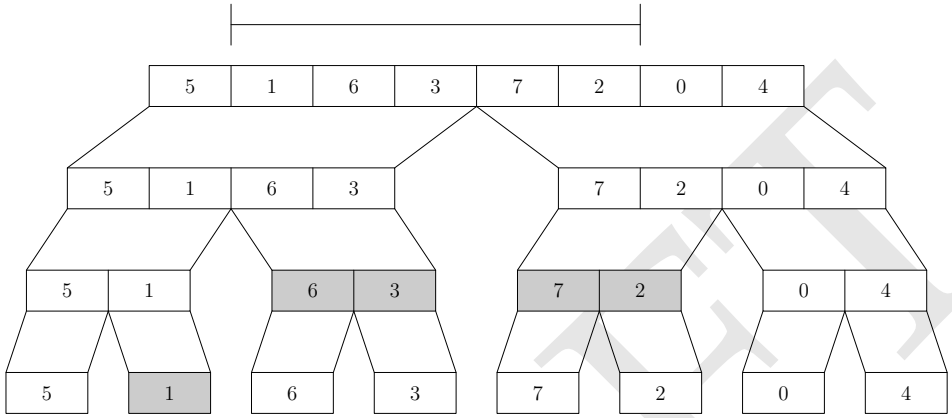


Figure 9.1: The $2N - 1$ intervals to precompute.

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.

Algorithm 9.7: Segment Tree Construction

```
procedure MAKETREE(sequence A)
    tree ← new int[2|N|]
    for i = |N| to 2|N| − 1 do
        tree[i] ← A[i − |N|]
    for i = |N| − 1 to 1 do
        tree[i] ← tree[2 · i] + tree[2 · i + 1]
    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 in Algorithm 9.7.

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.

Algorithm 9.8: Segment Tree Update

```
procedure UpdateTree(tree T, index i, value v)
    index ← i + N
    tree[index] ← v
    while index ≠ 0 do
        index ← index/2
        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 take $\Theta(\log N)$ 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 $O(\log N)$.

Algorithm 9.9: Segment Tree Query

```
procedure QueryTree(tree T, index i, query [L, R), tree interval [L′, R′))
    if R ≤ L′ or L > R′ then
        return 0
    if L = L′ and R = R′ then
        return T[i]
```

$M = (L + R)/2$
$lsum = \text{QueryTree}(T, 2i, [L, \min(R, M)), [L', M))$
$rsum = \text{QueryTree}(T, 2i + 1, [\max(L, M), R), [M, R))$
return $lsum + rsum$

9.3 Chapter Notes

Chapter 10

Graph Algorithms

Graph theory is probably the richest of all algorithmic areas. You are almost guaranteed to see at least one graph problem in any given contest, so it is important to be well versed in the common algorithms that operate on graphs. The most important such algorithms concern shortest paths from some vertex, which will also be our first object of study.

10.1 Breadth-First Search

One of the most common basic graph algorithms is the ***breadth-first search***. Its main usage is to find the distances from a certain vertex in an unweighted graph:

Single-Source Shortest Path, Unweighted Edges
Given an unweighted graph $G = (V, E)$ and a source vertex s, compute the shortest distances $d(s, v)$ for all $v \in V$.

For simplicity, we will first consider the problem on a grid graph, where the unit squares constitute vertices, and vertices which share an edge are connected. Additionally, some squares are blocked (and do not contain a vertex). An example can be seen in Figure 10.1.

Let us solve this problem inductively. First of all, what vertices have distance 0? Clearly, this can only be the source vertex s itself. This seems like a reasonable base case, since the problem is about shortest paths from s. Then, what vertices have distance 1? These are exactly those with a path consisting of a single edge from s, meaning they are the neighbors of s (marked in Figure 10.2.

If a vertex v should have 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

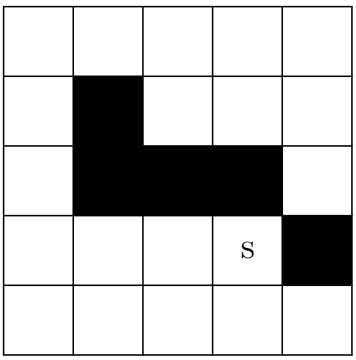


Figure 10.1: An example grid graph, with source marked s.

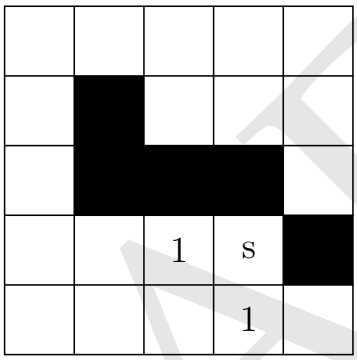


Figure 10.2: The square with distance 1 from the source.

construct a path of length 2 simply by extending the path of any neighbor of distance 1 with the edge (u, v).

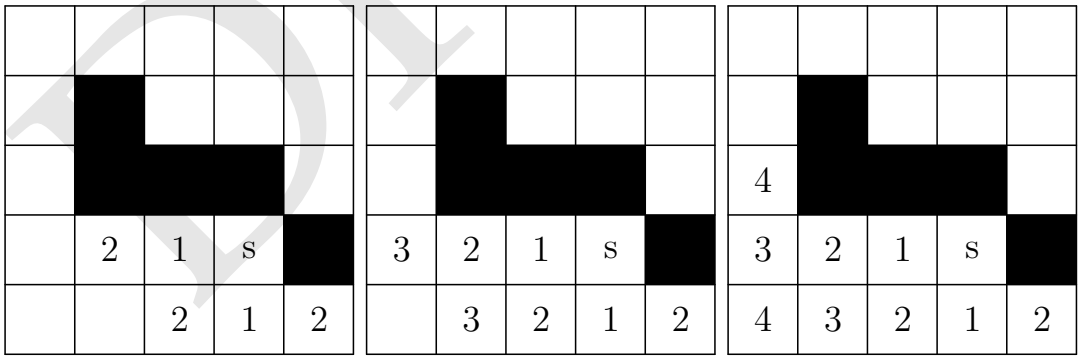


Figure 10.3: The squares with distance 2, 3 and 4.

In fact, this reasoning generalizes to any particular distance, i.e., that all the vertices that have exactly distance k are those that have a neighbor of distance $k-1$ but no neighbor to a vertex with a smaller distance. Using this, we can construct an algorithm to solve the problem. 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 $dist$. This algorithm is called the ***breadth-first search***.

Exercise 10.1

Use the BFS algorithm to compute the distance to every square in the following grid:

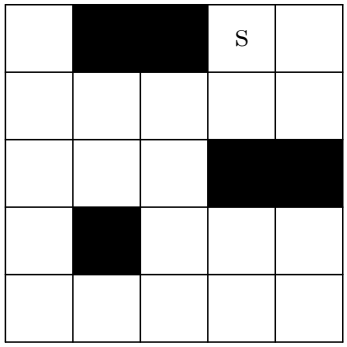


Figure 10.4

We can implement this the following way:

Algorithm 10.1: Breadth-First Search

```
procedure BreadthFirstSearch(vertices V, vertex s)
    distances ← new int[|V|]
    fill distances with ∞
    curDist ← 0
    curDistVertices ← new vector
    curDistVertices.add(s)
    distances[s] = curDist
    while curDistVertices ≠ ∅ do
        nCurDist = curDist + 1
        nCurDistVertices ← new vector
        for from ∈ curDistVertices do
            for v ∈ from.neighbours do
                if distances[v] = ∞ then
                    nCurDistVertices.add(v)
                    distances[v] = nCurDist
        curDist = nCurDist
        curDistVertices = nCurDistVertices
    return distances
```

Each vertex is added to *nCurDistVertices* at most once, since it is only pushed if

$distances[v] = \infty$ whereupon it is immediately set to something else. We then iterate through every neighbor of all these vertices. In total, the number of all neighbours is $2E$, so the algorithm in total uses $\Theta(V + E)$ time.

Usually, the outer loop are often coded in another way. Instead of maintaining two separate vectors, we can merge them into a single queue:

Algorithm 10.2: Breadth-First Search, Variant

```
while curDistVertices ≠ ∅ do
    from ← curDistVertices.front()
    curDistVertices.pop()
    for from ∈ curDistVertices do
        for v ∈ from.neighbours do
            if distances[v] = ∞ then
                curDistVertices.add(v)
                distances[v] = distances[from] + 1
```

The order of iteration is equivalent to the original order.

Exercise 10.2

Prove that the shorter way of coding the BFS loop (Algorithm 10.2) is equivalent to the longer version (Algorithm 10.1).

In many problems the task is to find a shortest path between some pair of vertices where the graph is given *implicitly*:

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.

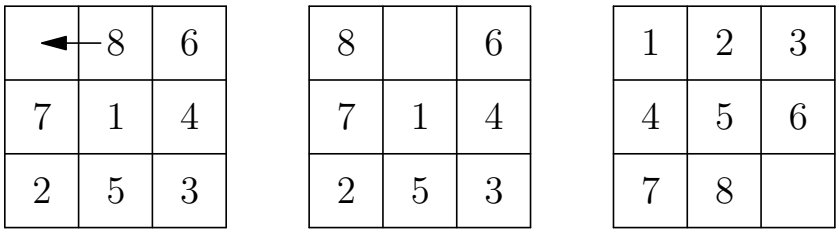


Figure 10.5: An example 8-puzzle, with a valid move. The rightmost puzzle shows the target configuration.

Given a puzzle, determine how many moves are required to solve it, or if it cannot be solved.

This 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 such transitions to some goal state. We can model such problems using a graph. The vertices are then the possible arrangements of the tiles in the grid, and an edge connects two states if the differ by a single move. A sequence of moves from the starting state to the target configuration then represents a path in this graph. The minimum number of moves required is thus the same as the distance between those vertices in the graph, meaning we can use a BFS.

In such a problem, most of the code is usually concerned with the representation of a state as a vertex, and generating the edges that a certain vertex is adjacent to. When an implicit graph is given, we generally do not compute the entire graph explicitly. Instead, we use the states from the problems as-is, and generate the edges of a vertex only when it is being 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 that we can reach from a certain state.

Algorithm 10.3: Generating 8-puzzle Moves

```
typedef vector<vi> Puzzle;

vector<Puzzle> edges(const Puzzle& v) {
  int emptyRow, emptyCol;
  rep(row,0,3)
    rep(col,0,3)
      if (v[row][col] == 0) {
        emptyRow = row;
        emptyCol = col;
      }
  vector<Puzzle> possibleMoves;
  auto makeMove = [&](int rowMove, int colMove) {
    int newRow = row + rowMove;
    int newCol = col + colMove;
    if (newRow >= 0 && newCol >= 0 && newRow < 3 && newCol < 3) {
      Puzzle newPuzzle = v;
      swap(newPuzzle[emptyRow][emptyCol], newPuzzle[newRow][newCol]);
      possibleMoves.spush_back(newPuzzle);
    }
  };
  makeMove(-1, 0);
  makeMove(1, 0);
  makeMove(0, -1);
  makeMove(0, 1);
```

```
25    return possibleMoves;
26  }
```

With the edge generation in hand, the rest of the solution is a normal BFS, slightly modified to account for the fact that our vertices are no longer numbered $0, \ldots, V-1$. We can solve this by using e.g. maps instead.

Algorithm 10.4: 8-puzzle BFS

```
1  int puzzle(const Puzzle& S, const Puzzle& target) {
2    map<Puzzle, int> distances;
3    distances[S] = 0;
4    queue<Puzzle> q;
5    q.push(S);
6    while (!q.empty()) {
7      const Puzzle& cur = q.front(); q.pop();
8      int dist = distances[cur];
9      if (cur == target) return dist;
10     for (const Puzzle& move : edges(cur)) {
11       if (distances.find(move) != distances.end()) continue;
12       distances[move] = dist + 1;
13       q.push(move);
14     }
15   }
16   return -1;
17 }
```

Exercise 10.3 — Kattis Exercises

Button Bashing – buttonbashing

10.2 Depth-First Search

The depth-first search is an analogue to the breadth-first search that visits vertices in another order. Similarly to how the BFS grows the set of visited vertices using a wide frontier around the source vertex, the depth-first search proceeds its search by, at every step, trying to plunge deeper into the graph. This order is called the depth-first order. More precisely, the search starts at some source vertex s. Then, any neighbor of s is chosen to be the next vertex v. Before visiting any other neighbor of s, we first visit any of the neighbours of v, and so on.

Implementing the depth-first search is usually done with a recursive function, using a vector *seen* to keep track of visited vertices:

Algorithm 10.5: Depth-First Search

```
procedure DEPTH-FIRST SEARCH(vertex at, adjacency list G)
    if seen[at] then
        return
    seen[at] = true
    for neighbour ∈ G[at] do
        dfs(neighbour, G)
```

In languages with limited stack space, it is possible to implement the DFS iteratively using a stack instead, keeping the vertices which are currently open in it.

Due to the simplicity of coding the DFS compared to a BFS, it is usually the algorithm of choice in problems where we want to visit all the vertices.

Coast Length

KTH Challenge 2011 – Ulf Lundström

The residents of Soteholm value their coast highly and therefore want to maximize its total length. For them to be able to make an informed decision on their position in the issue of global warming, you have to help them find out whether their coastal line will shrink or expand if the sea level rises. From height maps they have figured out what parts of their islands will be covered by water, under the different scenarios described in the latest IPCC report on climate change, but they need your help to calculate the length of the coastal lines.

You will be given a map of Soteholm as an $N \times M$ grid. Each square in the grid has a side length of 1 km and is either water or land. Your goal is to compute the total length of sea coast of all islands. Sea coast is all borders between land and sea, and sea is any water connected to an edge of the map only through water. Two squares are connected if they share an edge. You may assume that the map is surrounded by sea. Lakes and islands in lakes are not contributing to the sea coast.

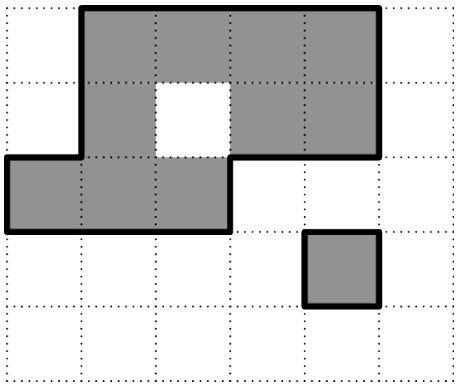


Figure 10.6: Gray squares are land and white squares are water. The thick black line is the sea coast.

We can consider the grid as a graph, where all the water squares are vertices, and two squares have an edge between them if they share an edge. If we surround the entire grid by an water tiles (a useful trick to avoid special cases in this kind of grid problems), the sea consists exactly of those vertices that are connected to these surrounding water tiles. This means we need to compute the vertices which lie in the same connected component as the sea – a typical DFS task[1]. After computing this component, we can determine the coast length by looking at all the squares which belong to the sea. If such a square share an edge with a land tile, that edge contributes 1 km to the coast length.

Algorithm 10.6: Coast Length

```
const vpi moves = {pii(-1, 0), pii(1, 0), pii(0, -1), pii(0, 1)};

int coastLength(const vector<vector<bool>>& G) {
  int H = sz(G) + 4;
    W = sz(G[0]) + 4;
  vector<vector<bool>> G2(H, vector<bool>(W, true));
  rep(i,0,sz(G)) rep(j,0,sz(G[i])) G2[i+2][j+2] = G[i][j];
  vector<vector<bool>> sea(H, vector<bool>(W));

  function<void(int, int)> floodFill = [&](int row, int col) {
    if (row < 0 || row >= H|| col < 0 || col >= W) return;
    if (sea[row][col]) return;
    sea[row][col] = true;
    trav(move, moves) floodFill(row + move.first, col + move.second);
  };
  dfs(0, 0);

  int coast = 0;
  rep(i,1,sz(G)+1) rep(j,1,sz(G[0])+1) {
    if (sea[i][j]) continue;
    trav(move, moves) if (!sea[i + move.first][j + move.second]) coast++;
  }
  return coast;
}
```

Exercise 10.4 — Kattis Exercises

Mårten's DFS – martensdfs

[1]This particular application of DFS, i.e. computing a connected area in a 2D grid, is called a *flood fill*.

10.3 Weighted Shortest Path

The theory of computing shortest paths in the case of weighted graphs is a bit more rich than the unweighted case. Which algorithm to use depends on three factors:

- The number of vertices.
- Whether edge weights are non-negative or not.
- If we seek shortest paths only from a single vertex or between all pairs of vertices.

There are mainly three algorithms used: *Dijkstra's Algorithm*, the *Bellman-Ford* algorithm, and the *Floyd-Warshall* algorithm.

10.3.1 Dijkstra's Algorithm

Dijkstra's Algorithm can be seen as an extension of the breadth-first search that works for weighted graphs as well.

Single-Source Shortest Path, non-negative weights
Given a weighted graph $G = (V, E)$ where all weights are non-negative and a source vertex s, compute the shortest distances $d(s, v)$ for all $v \in V(G)$.

It has a similar inductive approach, where we iteratively compute the shortest distances to all vertices in ordered by distance. The difference lies in that we do not immediately know when we have found the shortest path to a vertex. For example, the shortest path from a neighbour to the source vertex may now use several other vertices in its shortest path to s (Figure **??**).

image

However, can this be the case for *every* vertex adjacent to s? In particular, can the neighbour to s with the smallest edge weight W to s have a distance smaller than w? This is never the case in Figure **??**, and actually holds in general. Assume that this is not the case. In Figure **??**, this hypothetical scenario is shown. Any such path must still pass through some neighbour u of s. By assumption, the weight of the edge (s, u) must be larger than W (which was the minimal weight of the edges adjacent to s). This reasoning at least allows us to find the shortest distance to one other vertex.

10.3.2 Bellman-Ford

Single-Source Shortest Path, non-negative weights

Given a weighted graph $G = (V, E)$ where all weights are non-negative and a source vertex s, compute the shortest distances $d(s, v)$ for all $v \in V(G)$.

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 else in the graph could be used to construct a shorter path to a vertex which was already marked as completed. However, the concept of relaxing an edge is still very much applicable and allows us to construct a slower, inductive solution.

Initially, we know the shortest distances to each vertex, assuming we are allowed to traverse 0 edges from the source. Assuming that we know these values when allowed to traverse up to k edges, can we find the shortest paths that traverse up to $k + 1$ edges? This way of thinking is similar to how to solved the BFS problem. Using a BFS, we computed the vertices at distance $d + 1$ by taking the neighbours of the vertices at distance d Similarly, we can find the shortest path to a vertex v that traverse up to $k + 1$ edges, by attempting to extend the shortest paths using k edges from the neighbours of v. Letting $D(k, v)$ be the shorter distance to v by traversing up to k edges, we arrive at the following recursion:

$$D(k, v) = \min \begin{cases} 0 & \text{if } v = s \\ D(k-1, v) & \text{if } k > 0 \\ \min_{e=(u,v)\in E} D(k-1, u) + W(e) & \text{if } k > 0 \end{cases}$$

Algorithm 10.7: Bellman-Ford

```
procedure BellmanFord(vertices V, edges E, vertex s)
    D ← new int[|V|][|V| + 1]
    fill D with ∞
    D[0][s] ← 0
    for k = 1 to |V| do
        D[k] = D[k − 1]
        for e = (u, v) ∈ E do
            D[k][v] = min(D[k][v], D[k − 1][u] + W(e))
    return D
```

All in all, the states $D(v, i)$ for a particular i takes time $\Theta(|E|)$ to evaluate. To compute the distance $d(s, v)$, we still need to know what the maximum possible k needed to arrive at this shortest path could be. It turns out that this could potentially be infinite, in the case where the graph contains a negative-weight cycle. Such a cycle can be exploited to construct arbitrarily short paths.

However, of no such cycle exists, $k = |V|$ will be sufficient. If a shortest path uses more than $|V|$ edges, it must contain a cycle. If this cycle is not of negative weight, we may simply remove it to obtain a path of at most the same length. Thus, the algorithm takes $O\Theta(|V||E|)$.

Exercise 10.5

Bellman-Ford can be adapted to instead use only $\Theta(V)$ memory, by only keeping a current know shortest path and repeatedly relaxing every edge. Sketch out the pseudo code for such an approach, and prove its correctness.

Exercise 10.6

We may terminate Bellman-Ford earlier without loss of correctness, in case $D[k] = D[k-1]$. How can this fact be used to determine whether the graph in question contains a negative-weight cycle?

10.3.3 Floyd-Warshall

All-Pairs Shortest Paths

Given a weighted graph $G = (V, E)$, compute the shortest distance $d(u, v)$ for **every** pair of vertices u, v.

The Floyd-Warshall algorithm is a remarkably short method to solve the all-pairs shortest paths problem. It basically consists of three nested loops containing a single statement:

Algorithm 10.8: Floyd-Warshall

```
procedure FLOYD-WARSHALL(distance matrix D)
    for k = 0 to |V| − 1 do
        for i = 0 to |V| − 1 do
            for j = 0 to |V| − 1 do
                D[i][j] = min(D[i][j], D[i][k] + D[k][j])
    return D
```

Initially, the distance matrix D contains the distances of all the edges in E, so that $D[i][j]$ is the weight of the edge (i, j) if such an edge exists, ∞ if there is no edge between i and j or 0 if $i = j$. Note that if multiple edges exists between i and j, $D[i][j]$ must be given the minimum weight of them all. Additionally, if there is a self-loop (i.e. an edge from i to i itself) of negative weight, $D[i][i]$ must be set to this value.

To see why this approach works, we can use the following invariant proven by induction. After the k'th iteration of the loop, $D[i][j]$ will be at most the minimum distance between i and j that uses vertices $0, 1, \ldots, k-1$. Assume that this is true for a particular k. After the next iteration, there are two cases for $D[i][j]$. Either there is no shorter path using vertex k than those using only vertices $0, 1, \ldots, k-1$. In this case, $D[i][j]$ will fulfill the condition by the induction assumption. If there is a shorter path between i and j if we use the vertex k, this must have length $D[i][k] + D[k][j]$, since $D[i][k]$ and $D[k][j]$ both contain the shortest paths between i and k, and k and j using vertices $0, 1, \ldots, k-1$. Since we set $D[i][j] = \min(D[i][j], D[i][k] + D[k][j])$ in the inner loop, we will surely find this path too in this iteration. Thus, the statement is true after the $k+1$'th iteration too. By induction, it is true for $k = |V|$, meaning $D[i][j]$ contains at most the minimum distance between i and j using any vertex in the graph.

10.4 Minimum Spanning Tree

Using the depth-first search, we were able to find a subtree of a connected graph which spanned the entire set of vertices. A particularly important such spanning tree is the minimum spanning tree.

Minimum Spanning Tree

We say that the weight of a spanning tree is the sum of the weights of its edges. A ***minimum spanning tree*** is a spanning tree whose weight is minimal.

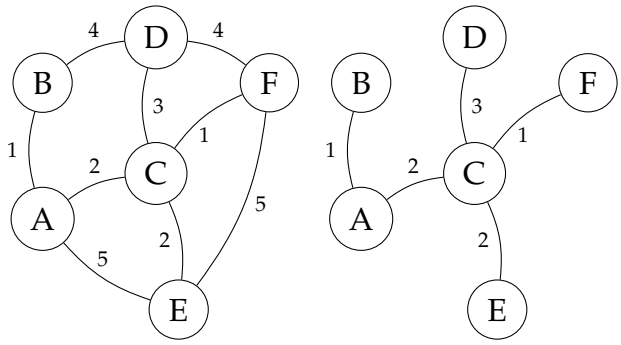


Figure 10.7: A graph with a corresponding minimum spanning tree.

Given a weighted graph, find a minimum spanning tree.

Mainly two algorithms are used to solve the Minimum Spanning Tree (MST) problem: either ***Kruskal's Algorithm*** which is based on the Union-Find data structure, or ***Prim's Algorithm*** which is an extension of Dijkstra's Algorithm. We will demonstrate Kruskal's, since it is by far the most common of the two.

Kruskal's algorithm is based on a greedy, incremental approach that reminds us of the Scheduling problem (Section 6.3). In the Scheduling problem, we tried to find any interval that we could prove to be part of an optimal solution, by considering various extremal cases. Can we do the same thing when finding a minimum spanning tree?

First of all, if we can always find such an edge, we are essentially done. Given an edge (a, b) that we know is part of a minimum spanning tree, we can contract the two vertices a and b to a single vertex ab, with all edges adjacent to the two vertices. Any edges that go between contracted vertices are ignored. An example of this process in action be be seen in Figure 10.8. Note how the contraction of an edge reduces the problem to finding an MST in a smaller graph.

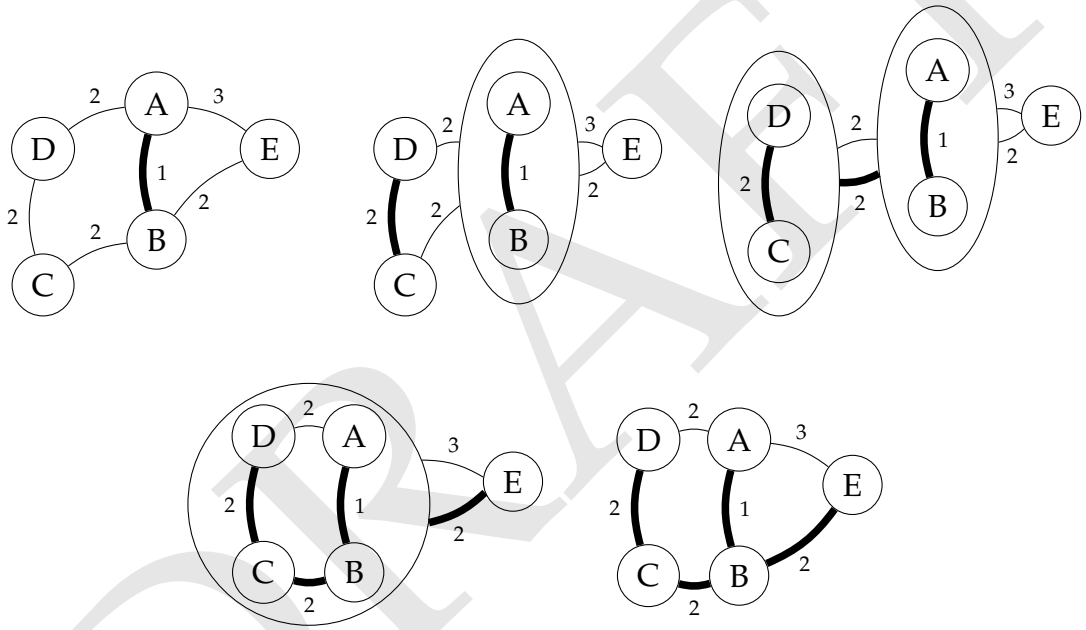


Figure 10.8: Incrementally constructing a minimum spanning tree by merging

A natural extremal case to consider is the edge with the minimum weight. After all, we are trying to minimize the edge sum. Our proof is similar in structure to the Scheduling problem as well by using a swapping argument.

Assume that a minimum-weight edge $\{a, b\}$ with weight w is not part of any minimum spanning tree. Then, consider the spanning tree with this edge appended. The graph will then contain exactly one cycle. In a cycle, any edge can be removed while maintaining connectivity. This means that if any edge $\{c, d\}$ on this cycle have a weight w' larger weight than w, we can erase it. We will thus have replaced the edge $\{c, d\}$ by $\{a, d\}$, while changing the weight of the tree by $w - w' < 0$, reducing the sum of weights. Thus, the tree was improved by using the minimum-weight edge, proving that it could have been part of the tree.

Exercise 10.7

What happens if all edges on the cycle that appears have weight w? Is this a problem for the proof?

When implementing the algorithm, the contraction of the edge added to the minimum spanning tree is generally not performed explicitly. Instead, a disjoint set data structure is used to keep track of which subsets of vertices have been contracted. Then, all the original edges are iterated through in increasing order of weight. An edge is added to the spanning tree if and only if the two endpoints of the edge are not already connected (as in Figure 10.8).

Algorithm 10.9: Minimum Spanning Tree

```
procedure MinimumSpanningTree(vertices V, edges E)
    sort E by increasing weight
    uf ← new DisjointSet(V)
    mst ← new Graph
    for each edge {a, b} ∈ E do
        if not uf.sameSet(a, b) then
            mst.append(a, b)
            uf.join(a, b)
    return mst
```

The complexity of this algorithm is dominated by the sorting (which is $O(E \log V)$), since operations on the disjoint set structure is $O(\log V)$.

10.5 Chapter Notes

Chapter 11

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*).

11.1 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 11.1 — 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 11.1, 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.

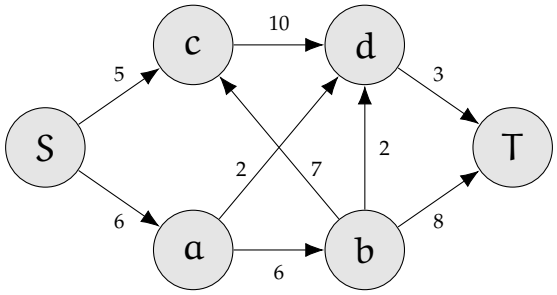


Figure 11.1: An example flow network.

Definition 11.2 — 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 $v \in V \{S, T\}$, $\sum_{e \in in(v)} f(e) = \sum_{e \in out(v)}$ – flow is conserved for each vertex, possibly except the source and sink.

The ***size*** of a flow is defined to be the value

$$\sum_{v \in out(S)} f(v) - \sum_{v \in in(S)} f(v)$$

In a computer network, the flows could e.g. represent the current rate of transfer through each connection.

Exercise 11.1

Prove that the size of a given flow also equals

$$\sum_{v \in in(T)} f(v) - \sum_{v \in out(T)} f(v)$$

i.e. the excess flow out from S must be equal to the excess flow in to T.

In Figure 11.2, flows have been added to the network from Figure 11.1.

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.

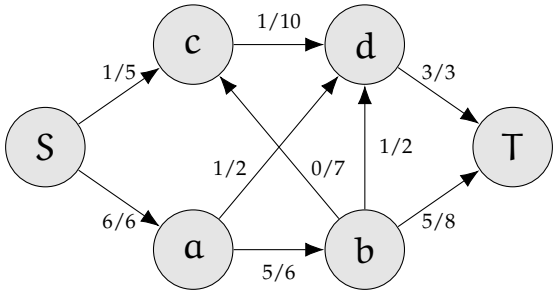


Figure 11.2: An example flow network, where each edge has an assigned flow. The size of the flow is 8.

Maximum Flow

Given a flow network (V, E, c, S, T), construct a maximum flow from S to T.

Input

A flow network.

Output

Output the maximal size of a flow, and the flow assigned to each edge in one such flow.

Exercise 11.2

The flow of the network in Figure 11.2 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.

11.2 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).

11.2.1 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.

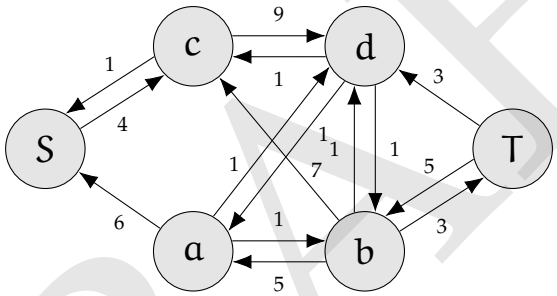


Figure 11.3: The residual flows from the network in Figure 11.2.

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 11.3, 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:

- 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 implemented in pseudo code on Algorithm 11.2.

Algorithm 11.1: Augmenting a Flow

```
procedure AUGMENT(path P)
    inc ← ∞
    for e ∈ P do
        inc ← min(inc, c(e) − f(e))
    for e ∈ P do
        f(e) ← f(e) + inc
        f(b(e)) ← f(b(e)) − inc
    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.

11.2.2 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. 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).

Algorithm 11.2: Finding an Augmenting Path by DFS

```
procedure AUGMENTINGPATH(flow network (V, E, c, f, S, T))
    bool[] seen ← new bool[|V|]
    Stack stack ← new Stack
    found ← DFS(S, T, f, c, seen, stack)
    if found then
        return stack
    return Nil
procedure DFS(vertex at, sink T, flow f, capacity c, path p)
```

```
    p.push(at)
    if at = T then
        return true
    for every out-edge e = (at, v) from at do
        if f(e) < c(e) then
            if DFS(v, T, f, c, p) then
                return true
    p.pop()
    return false
```

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).

11.3 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 11.3

The minimum path cover reduction can be modified slightly to find a *minimum cycle cover* in a directed graph instead. Construct such a reduction.

11.4 Chapter Notes

erence to
per

The Edmonds-Karp algorithm was originally published in 1970 by Yefim Dinitz. The paper by Edmonds and Karp

Chapter 12

Strings

In computing, much of the information we process is text. Therefore, it should not come as a surprise that many common algorithms and problems focus concerns text strings. In this chapter, we will study some of the common string algorithms and data structures.

12.1 Tries

The *trie* (also called a *prefix tree*) is the most common string-related data structure. It represents a set of words as a rooted tree, where every prefix of every word is a vertex, with children from a prefix P to all strings Pc which are also prefixes of a word. If two words have a common prefix, the prefix only appears once as a vertex. The root of the tree is the empty 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.

The most basic operation of the trie is the insertion of strings, which may be implemented as follows.

Algorithm 12.1: Trie

```
1  struct Trie {
2    map<char, Trie> children;
3    bool isWord = false;
4
5    void insert(const string& s, int pos) {
6      if (pos != sz(s)) children[s[pos]].insert(s, pos + 1);
7      else isWord = true;
8    }
9
```

```
10  };
```

We mark those vertices which corresponds to the inserted word using a boolean flag `isWord`. Many problems essentially can be solved by very simple usage of a trie, such as the following IOI problem.

Type Printer

International Olympiad in Informatics 2008

You need to print N words on a movable type printer. 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.

Your task is to output a sequence of operations that prints all the words using the minimum number of operations needed.

Input

The first line contains the number of words $1 \leq N \leq 25\,000$. The next N lines contain the words to be printed, one per line. Each word is at most 20 letters long and consist only of lower case letters `a-z`. All words will be distinct

Output

Output a sequence of operations that prints all the words. The operations should be given in order, one per line, starting with the first. Adding a letter `c` is represented by outputting `c` on a line. Removing the last letter of the current word is represented by a `-`. Printing the current word is done by outputting `P`.

Let us start by solving a variation of the problem, where we are not allowed to leave letters in the printer at the end. First of all, are there 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 never wish to input a sequence that is

not a prefix of a word we wish to type. For example, if we input abcdef and this is not a prefix of any word, we must at some point erase the last letter f, without having printed any words. But then we can erase the entire sequence of operations between inputting the f and erasing the f, without changing what words we print.

On the other hand, every prefix of a word we wish to print must at some point appear on the type writer. Otherwise, we would not be able to reach the word we wish to print. Therefore, the partial words to ever appear on the type printer are exactly the prefixes of the words we wish to print – strongly hinting at a trie-based solution.

If we build the trie of all words we wish to print, it contains as vertices exactly those strings which will appear as partial words on the printer. Furthermore, the additions and removals of letters form a sequence of vertices that are connected by edges in this trie. We can move either from a prefix P to a prefix Pc, or from a prefix Pc to a prefix P, which are exactly the edges of a trie. The goal is then to construct the shortest possible tour starting at the root of the trie and passing through all the vertices of the trie.

Since a trie is a tree, any such trail must pass through every edge of the trie at least twice. If we only passed through an edge once, we can never get back to the root since every edge disconnects the root from the endpoint of the edge further away from the root. It is actually possible to construct a trail which passes through every edge exactly twice (which is not particularly difficult if you attempt this task by hand). As it happens, the depth-first search of a tree passes through an edge exactly twice – once when first traversing the edge to an unvisited vertex, and once when backtracking.

The problem is subtly different once we are allowed to leave some letters in the printer at the end. Clearly, the only difference between an optimal sequence when letters may remain and an optimal sequence when we must leave the printer empty is that we are allowed to skip some trailing removal operations. If the last word we print is S, the difference will be exactly $|S|$ ‘‘-’’ operations. An optimal solution will therefore print the longest word last, in order to “win” as many ‘‘-’’ operations as possible. We would like this last word to be the longest word *of all the ones we print* if possible. In fact, we can order our DFS such that this is possible. First of all, our DFS starts from the root and the longest word is $s_1 s_2 \dots s_n$. When selecting which order the DFS should visit the children of the root in, we can select the child s_1 last. Thus, all words starting with the letter s_1 will be printed last. When visiting s_1, we use the same trick and visit the child $s_1 s_2$ last of the children of s_1, and so on. This guarantees S to be the last word to be printed.

Note that the solution requires no additional data to be stored in the trie – the only modification to our basic trie is the DFS.

Algorithm 12.2: Typewriter

```
struct Trie {
  ...

  void dfs(int depth, const string& longest) {
    trav(it, children)
      if (it->first != longest[depth])
        dfs2(depth, longest, it->first);
    dfs2(depth, longest, longest[depth]);
  }

  void dfs2(int depth, const string& longest, char output) {
    cout << output << endl;
    if (isWord) cout << "P" << endl;
    children[output]->dfs(depth + 1, longest);
    if (longest[depth] != output) {
      cout << "-" << endl;
    }
  }
};
```

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 requires only a small augmentation of a trie, to solve a problem which looks complex.

Rareville

In Rareville, everyone must have a distinct name. When a new-born baby is to be given a name, its parents must first visit *NAME*, the Naming Authority under the Ministry of Epithets, to have its name approved. The authority has a long list of all names assigned to the citizens of Rareville. When deciding whether to approve a name or not, a case officer uses the following procedure. They start at the first name in the list, and read the first letter of it. If this letter matches the first letter of the proposed name, they proceed to read the next letter in the word. This is repeated for every letter of the name *in the list*. After reading a letter from the word, the case officer can sometime determine that this could not possibly be the same name as the proposed one. This happens if either

- the next letter in the proposed name did not match the name in the list
- there was no next letter in the proposed name
- there was no next letter in the name in the list

When this happen, the case officer starts over with the next name in the list, until

exhausting all names in the list. For each letter the case officer reads (or attempts to read) from a name in the list, one second pass.

Currently, there are N people in line waiting to apply for a name. Can you determine how long time the decision process will take for each person?

Input

The first line contains integers $1 \leq D \leq 200\,000$ and $1 \leq N \leq 200\,000$, the size of the dictionary and the number of people waiting in line. The next D lines contains one lowercase name each, the contents of the dictionary. The next N lines contains one lowercase name each, the names the people in line wish to apply with. The total size of the lists is at most 10^6 letters.

Output

For each of the N names, output the time (in seconds) the case officer needs to decide on the application.

The problem clearly relates to prefixes in some way. Given a dictionary word A and an application for a name B, the case officer needs to read letters from A corresponding to the longest common prefix of A and B, plus 1. Hence, our solution will probably be to consider all the prefixes of each proposed name, which is exactly what tries are good at.

Instead of thinking about this process one name a time, we use a common trie technique and look at the transpose of this problem, i.e. for every i, how many names C_i have a longest common prefix of length *at least* i when handling the application for a name S? This way, we have transformed the problem from being about D individual processes to $|S|$ smaller problems which treats the dictionary as unified group of strings. Then, we will have to read $C_0 + C_1 + \cdots + C_{|S|}$ letters.

Now, the solution should be clear. We augment the trie vertex for a particular prefix p with the number of strings P_p in the list that start with this prefix. Initially, an empty trie has $P_p = 0$ for every p. Whenever we insert a new word $W = w_1 w_2 \ldots$ in the trie, we need to increment $P_{w_1}, P_{w_1 w_2}, \ldots$, to keep all the P_p correct, since we have added a new string which have those prefixes. Then, we have that $C_i = P_{s_1 s_2 \ldots s_i}$, so that we can compute all the numbers P_i by following the word S in the trie. The construction of the trie is linear in the number of characters we insert, and responding to a query is linear in the length of the proposed name.

Algorithm 12.3: Rareville

```
1  struct Trie {
2    map<char, Trie> children;
3    int P = 0;
4
5    void insert(const string& s, int pos) {
```

```
6       P++;
7       if (pos != sz(s)) children[s[pos]].insert(s, pos + 1);
8     }
9
10    int query(const string& s, int pos) {
11      int ans = P;
12      if (pos != sz(s)) {
13        auto it = children.find(s[pos]);
14        if (it != children.end) ans += it->second.query(s, pos + 1);
15      }
16      return ans;
17    }
18  };
```

12.2 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 contain 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

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.

Algorithm 12.4: Naive String Matching

```
procedure StringMatching(pattern P, string W)
    answer ← new vector
    for outer: i from 0 to |W| − |P| do
        for j from 0 to |P| − 1 do
            if P[j]! = W[i + j] then
                start next iteration of outer
        answer.append(i)
    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.

Algorithm 12.5: Longest Suffix Computation

```
procedure LONGESTSUFFIXES(pattern P)
    T ← new int[|P| + 1]
    for l from 1 to |P| − 1 do
        matchLen ← 0
```

```
        while l + matchLen ≤ |W| do
            if P[l]! = P[matchLen] then
                break
            matchLen ← matchLen + 1
            T[l + matchLen] = matchLen
    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.

Algorithm 12.6: String Matching using Longest Suffixes

```
procedure StringMatching(pattern P, text W)
    matches ← new vector
    T ← LongestSuffixes(P)
    pos ← 0, match ← 0
    while pos + match < |W| do
        if match < |P| and W[pos + match] = P[match] then
            match ← match + 1
        else if match = 0 then
            pos ← pos + 1
        else
            pos ← pos + match − T[match]
            match ← T[match]
        if match = |P| then
            matches.append(match)
    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:

Algorithm 12.7: Improved Longest Suffix Computation

```
procedure LongestSuffixes(pattern P)
    T ← new int[|P| + 1]
    pos ← 1, match ← 0
    while pos + match < |P| do
        if P[pos + match] = P[match] then
            T[pos + match] ← match + 1
            match ← match + 1
        else if match = 0 then
            pos ← pos + 1
        else
            pos ← pos + match − T[match]
            match ← T[match]
        if match = |P| then
            matches.append(match)
    return T
```

Using the same analysis as for the improved string matching, this precomputation is instead $\Theta(|P|)$. The resulting string matching then takes $\Theta(|P| + |W|)$.

This string matching algorithm is called the ***Knuth-Morris-Pratt*** (KMP) algorithm.

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.

12.3 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

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, 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 first line contains an integer $2 \leq N \leq 5000$, the number of friends on FriendBook. N lines follow, each containing n characters. The c'th character on the r'th line S_{rc} 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`.

Output

Output a single integer, the number of pairs of persons for which the quote holds.

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 12.1 — 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 12.1 — 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 12.1

Prove the properties of Theorem 12.1

Exercise 12.2

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 12.2) 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 $H(S_1), H(s_1 s_2), \dots, H(s_1 s_2 \dots s_n)$ 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. Find the shortest string L, such that S is a substring of the infinite string $T = \dots LLLLL \dots$.

Input

The first and only line of the input contains the string S, with $1 \leq |S| \leq 10^6$.

Output
Output the string L. If there are multiple strings L of the shortest length, you can 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 $L = s_1 s_2 \dots s_l$. 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

$$s_1 = s_{l+1} = s_{2l+1} = \dots$$
$$s_2 = s_{l+2} = s_{2l+2} = \dots$$
$$\dots$$
$$s_l = s_{2l} = s_{3l} = \dots$$

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:

$$s_1 s_2 \dots s_{n-l+1} = s_{l+1} s_{l+2} \dots s_n$$

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.

Algorithm 12.8: 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 $1 + 2 + \dots + k = \Theta(k^2)$, so $k = O(\sqrt{P})$.

Substring Range Matching

Petrozavodsk Winter Training Camp 2015

Given N strings $s_1, s_2, \ldots, s_N$ and a list of 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.

Input

The first line contains $1 \leq N \leq 50\,000$ and the number of queries $0 \leq Q \leq 100\,000$. The next N lines contains the strings $s_1, s_2, \ldots, s_N$, one per line. The next Q lines contains one query each. A query is given by the integers $1 \leq L \leq R \leq N$ and a string S.

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.

Output

For each query L, R, S, output a line with the answer to the query.

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.

Algorithm 12.9: Substring Range Matching

```
1  int countInterval(int upTo, const set<pii>& s) {
2    auto it = s.lower_bound(pii(upTo + 1, 0));
3    if (it == s.begin()) return 0;
4    return (--it)->second;
5  }
6
7  int main() {
8    int N, Q;
```

```
9     cin >> N >> Q;
10    vector<string> s(N);
11    rep(i,0,N) cin >> s[i];
12
13    map<int, set<string>> patterns;
14
15    vector<tuple<int, int, string>> queries;
16    rep(i,0,Q) {
17      int L, R;
18      string S;
19      cin >> L >> R >> S;
20      queries.emplace_back(L, R, S);
21      patterns[sz(s)].insert(S);
22    }
23
24    map<H, set<pii>> hits;
25    trav(pat, patterns) {
26      rep(i,0,N) {
27        vector<H> hashes = rollHash(s[i], pat.first);
28        trav(h, hashes)
29          if (pat.second.count(h))
30            hits[h].emplace(i, sz(hits[h]) + 1);
31      }
32    }
33
34    trav(query, queries) {
35      H h = polyHash(get<2>(query));
36      cout << countInterval(R, hits[h]) - countInterval(L-1, hits[h]) << endl;
37    }
38  }
```

Exercise 12.3

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.

12.3.1 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 12.2 — 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 12.4

Prove that τ_{2i} is a palindrome.

Theorem 12.2 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 $1 \mid \mid H(\overline{\tau_n}) - H(\tau_n)$ 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.

12.3.2 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

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

$$a_{1,j} - p_{1,1} = \cdots = a_{1,j+n-1} - p_{1,n} = c$$

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.

Exercise 12.5 — Kattis Exercise

Chasing Subs – chasingsubs

12.4 Chapter Notes

rabin karp paper

KMP

hashing

Chapter 13

Combinatorics

Combinatorics deals with various discrete structures, such as graphs and permutations. In this chapter, we will mainly study the branch of combinatorics known as *enumerative combinatorics* – the art of counting. We will count the number of ways to choose K different candies from N different candies, the number of distinct seating arrangements around a circular table, the sum of sizes of all subsets of a set and many more objects. Many combinatorial counting problems are based on a few standard techniques which we will learn in this chapter.

13.1 The Addition and Multiplication Principles

The ***addition principle*** states that, given a finite collection of **disjoint** sets $S_1, S_2, \ldots, S_n$, we can compute the size of the union of all sets by simply 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 13.1 Assume 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 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 13.2 Assume that we have the same sets of candies C, G and L as in Example 13.1. We want to compose an entire dinner out of snacks, by choosing one chocolate bar, one bubble gum and a lollipop. The multiplication principles tells us that, modeling a snack dinner as a tuple $(c, g, l) \in C \times G \times L$, we can form our dinner in $5 \cdot 3 \cdot 4 = 60$ ways.

The addition principle is often useful when we solve counting problems by case analysis.

Example 13.3 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__
a_a_
a__a
_aa_
_a_a
__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.

Let us now apply these basic principle sto solve the following problem:

Kitchen Combinatorics

Northwestern Europe Regional Contest 2015 – Per Austrin

The world-renowned Swedish Chef is planning a gourmet three-course dinner for some muppets: a starter course, a main course, and a dessert. His famous Swedish cook-book offers a wide variety of choices for each of these three courses, though some of them do not go well together (for instance, you of course cannot serve chocolate moose and sooted shreemp at the same dinner).

Each potential dish has a list of ingredients. Each ingredient is in turn available from a few different brands. Each brand is of course unique in its own special way, so using a particular brand of an ingredient will always result in a completely different dinner experience than using another brand of the same ingredient.

Some common ingredients such as pølårber may appear in two of the three chosen dishes, or in all three of them. When an ingredient is used in more than one of the three selected dishes, Swedish Chef will use the same brand of the ingredient in all of them.

While waiting for the meecaroo, Swedish Chef starts wondering: how many different dinner experiences are there that he could make, by different choices of dishes and brands for the ingredients?

Input

The input consists of:

- five integers r, s, m, d, n, where $1 \leq r \leq 1\,000$ is the number of different ingredients that exist, $1 \leq s, m, d \leq 25$ are the number of available starter dishes, main dishes, and desserts, respectively, and $0 \leq n \leq 2\,000$ is the number of pairs of dishes that do not go well together.
- r integers $b_1, \ldots, b_r$, where $1 \leq b_i \leq 100$ is the number of different brands of ingredient i.
- $s + m + d$ dishes – the s starter dishes, then the m main dishes, then the d desserts. Each dish starts with an integer $1 \leq k \leq 20$ denoting the number of ingredients of the dish, and is followed by k distinct integers $i_1, \ldots, i_k$, where for each $1 \leq j \leq k$, $1 \leq i_j \leq r$ is an ingredient.
- n pairs of incompatible dishes.

Output

If the number of different dinner experiences Swedish Chef can make is at most 10^{18}, then output that number. Otherwise, output "`too many`".

The solution is a similar addition-multiplication principle combo as used in Example 13.3. First off, 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 $\Theta(1)$ 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 thus to compute this set by taking the union of ingredients for the three included dishes. This step takes $\Theta(k_a + k_b + k_c)$. Once this set is computed, 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.

13.2 Permutations

A ***permutation*** of a set S is an ordering of all the elements in the set. For example, the set $\{1, 2, 3\}$ has 6 permutations:

123	132
213	231
312	321

Our first "real" combinatorial problem will be to count the number of permutations of an n-element set S. When counting permutations, we use the multiplication principle. We will show a procedure that can be used to construct permutations one element at a time. Assume that the permutation is the sequence $\langle a_1, a_2, \ldots, a_n \rangle$. The first element of the permutation, a_1, can be assigned any of the n elements of S. Once this assignment has been made, we have $n - 1$ elements we can choose to be 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}. Using this argument for all n elements of the sequence, we can construct a permutation in $n \cdot (n - 1) \cdots 2 \cdot 1$ ways (by the multiplication principle).

This number is so useful that it has its own name and notation.

Definition 13.1 — Factorial
The ***factorial*** of n, where n is a non-negative integer, 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 thus begin 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. Asypmtotically, the grow as $n^{\Theta(n)}$. More precisely, the well-used *Stirling's formula*[1]

[1]Named after James Stirling (who have other important combinatorial objects named after him too), but stated already by his contemporary Abraham de Moivre.

gives the approximation

$$n! = \sqrt{2\pi n}\left(\frac{n}{e}\right)^n \left(1 + O\left(\frac{1}{n}\right)\right)$$

Exercise 13.1

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?

Exercise 13.2 — Kattis Problems

n'th permutation – nthpermutation

Name That Permutation – namethatpermutation

13.2.1 Permutations as Bijections

The word *permutation* has roots in Latin, meaning "to change completely". We are now going look at permutations in a very different light, which gives some justification to the etymology of the word.

Given a set such as $[5]$, we can fix some 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 permutation as a "rule" which describes how to move – *permute* – the elements.

Such a movement 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 $\langle 1, 2, 3, 4, \ldots, n\rangle$ the *identity permutation*, since the function given by the identity permutation is actually the identity function. As a function, we can also consider

the composition of two permutations. Given two permutations, α and β, their composition $\alpha\beta$ is also a permutation, given by $\alpha\beta(k) = \alpha(\beta(k))$. If we let $\sigma = \langle 5, 4, 3, 2, 1\rangle$ the composition with $\pi = \langle 1, 3, 4, 5, 2\rangle$ from above would then be

$$\begin{array}{r|ccccc} i & 1 & 2 & 3 & 4 & 5 \\ & \downarrow & \downarrow & \downarrow & \downarrow & \downarrow \\ \pi(i) & 1 & 3 & 4 & 5 & 2 \\ & \downarrow & \downarrow & \downarrow & \downarrow & \downarrow \\ \sigma\pi(i) & 5 & 3 & 2 & 1 & 4 \end{array}$$

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 regarding the multiplication of permutations follows from their functional properties, namely their associativity. We have that the permutation $(\alpha\beta)\gamma = \alpha(\beta\gamma)$, so we will take the liberty of dropping the parentheses and writing $\alpha\beta\gamma$.

Exercise 13.3 — Kattis Problems

Permutation Product – permutationproduct

Permutations also have inverses, which are just the inverses of their functions. The permutation $\pi = \langle 1, 3, 4, 5, 2\rangle$ which we looked at in the beginning thus have the inverse given by

$$\pi^{-1}(1) = 1 \qquad \pi^{-1}(3) = 2 \qquad \pi^{-1}(4) = 3 \qquad \pi^{-1}(5) = 4 \qquad \pi^{-1}(2) = 5$$

written in permutation notation as $\langle 1, 5, 2, 3, 4\rangle$. Since this is the functional inverse, we expect $\pi^{-1}\pi = \mathrm{id}$.

$$\begin{array}{r|ccccc} i & 1 & 2 & 3 & 4 & 5 \\ & \downarrow & \downarrow & \downarrow & \downarrow & \downarrow \\ \pi(i) & 1 & 3 & 4 & 5 & 2 \\ & \downarrow & \downarrow & \downarrow & \downarrow & \downarrow \\ \pi^{-1}\pi(i) & 1 & 2 & 3 & 4 & 5 \end{array}$$

Exercise 13.4 — Kattis Problems

Permutation Inverse – permutationinverse

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 we will consider every element exactly once, this procedure is $\Theta(n)$ for n-element permutations.

Exercise 13.5 — Kattis Problems

Cycle Decomposition – cycledecomposition

Given a permutation π, we define its ***order***, denoted $\operatorname{ord}\pi$, as the size of the set $\{\pi, \pi^2, \pi^3, \dots\}$. For all permutations except for the identity permutation, this is the smallest integer $k > 0$ 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 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 $(\pi^l)^k(c_1) = (\pi^{lk})(c_1) = c_1$. Hence, any power of π that is a multiple of l will act 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 $l_1, l_2, \ldots, l_m$, the smallest positive number that is the multiple of every l_i is $\operatorname{lcm}(l_1, l_2, \ldots, l_m)$. 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, a fact that uses the following fact which is left as an exercise:

Exercise 13.6

Prove that if π has a cycle of length l, we must have $l \mid \operatorname{ord} \pi$.

Exercise 13.7 — Kattis Problems

Order of a Permutation – permutationorder

Dance Reconstruction

Nordic Collegiate Programming Contest 2013 – Lukáš Poláček

Marek loves dancing, got really excited when he heard about 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 people and there were N marks on the floor. There was an arrow from each mark 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 2 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.

Another wedding is now coming up a year later, 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 K times during the time the song was played, so people moved K times along the arrows.

Given the two photos, can you help Marek reconstruct the arrows on the floor? On the two photos it can be seen for every person to which position he or she moved. Marek numbered the people in the first photo from 1 to N and then wrote the number of the person whose place they took in the second photo.

Marek's time is running out, so he is interested in any placement of arrows that could produce the two photos.

Input

Two integers $2 \leq N \leq 10\,000$ and $1 \leq K \leq 10^9$. Then, N integers $1 \leq a_1, \ldots, a_N \leq N$, denoting that dancer number i ended up at the place of dancer number a_i.

Every number between 1 and N appears exactly once in the sequence a_i.

Output

If it is impossible to find a placement of arrows such that the dance performed K times would produce the two photos, print "`Impossible`". Otherwise print N numbers on a line, the i'th number denoting to which person the arrow leads from person number i.

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 $\pi^K = a$.

When given permutation problems of this kind, we should probably attack it using cycle decompositions in some way. Since the cycles of π are all independent of each other under multiplication, it is a good guess that the decomposition can simplify the problem. The important question is then how a cycle of π is affected when taking powers. For example, a cycle of 10 elements in π would decompose into two cycles of length 5 in π^2, and five cycles of length 2 in π^5. The general case involves the divisors of l and K:

Exercise 13.8

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.

This suggests our first simplification of the problem: to consider all cycles of π^K partitioned by their lengths. By Exercise 13.8, cycles of different lengths are completely unrelated in the cycle decomposition of π^K.

The result also gives us a way to "*reverse*" the decomposition that happens to the cycles of π. Given $\frac{l}{m}$ cycles of length m in π^K, we can combine them into a l-cycle in π in the case where $m \cdot \gcd(l, K) = l$. By looping over every possible cycle length l (from 1 to N), we can then find all possible ways to combine cycles of π^K into larger cycles of π. This step takes $\Theta(N \log(N + K))$ 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, we want to partition them into sets of certain sizes (given by by previous computation). This step takes $\Theta(a \cdot c)$ ways, if there are c ways to combine a-length cycles.

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 13.8 basically outlines the reverse procedure).

13.3 Ordered Subsets

A variation of the permutation counting problem is to count the number of ordered sequences containing *exactly* k distinct elements, from a set of n. We can compute this by first consider the permutations of the entire set of n elements, and then group 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 we have chosen the first k elements of a permutation, there are $(n-k)!$ ways to order the remaining $n-k$ elements. Thus, we must have divided our $n!$ permutations into one group for each ordered k-length sequence, with each group containing $(n-k)!$ elements. To get the correct total, this means there must be $\frac{n!}{(n-k)!}$ such groups – and k-length sequences.

We call these objects ***ordered*** k***-subsets*** of an n-element set, and denote the number of such ordered sets 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. The difference to the permutation is that we stop after choosing the k'th element, which we can do in $(n-k+1)$ ways.

13.4 Binomial Coefficients

Finally, we are going to do away with the "ordered" part of the ordered k-subsets, and count the number of subsets of size k of an n-element size. This number is called the *binomial coefficient*, and is probably the most important combinatorial number there is.

To compute the number of k-subsets of a set of size n, we 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 we 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	ca	da
ac	bc	cb	db
ad	bd	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 get the number of unordered subsets by dividing 12 with $2!$, giving us the 6 different 2-subsets of $\{a,b,c,d\}$.

ab		
ac	bc	
ad	bd	cd

Definition 13.2 — Binomial Coefficient
The number of k-subsets of an n-set is called the ***binomial coefficient***

$$\binom{n}{k} = \frac{n!}{k!(n-k)!}$$

This is generally read as "n choose k".

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 will compute the binomial coefficient by alternating multiplications and divisions. We will start with storing $1 = \frac{1}{1}$. Then, we multiply with $n - r + 1$ and divide with 1, leaving us with $\frac{n-r+1}{1}$. In the next step we multiply with $n - r + 2$ and divide with 2, having computed $\frac{(n-r+1)\cdot(n-r+2)}{1\cdot 2}$. After doing this r times, we will be left with our binomial coefficient.

There is one big question mark from performing this procedure - why must our intermediate result 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 $\binom{n-r+1}{1}, \binom{n-r+2}{2}, \ldots, \binom{n-1}{r-1}, \binom{n}{r}$. 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:

$$\binom{n}{r} = \frac{n}{r}\binom{n-1}{r-1}$$

Exercise 13.9

Prove this identity combinatorially, by first multiplying both sides with r. (Hint: both sides count the number of ways to do the same two-choice process, but in different order.)

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. Then, we can 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 basically "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 $O(\min\{r, n-r\})$ instead of $O(r)$.

Exercise 13.10 — Kattis Exercises

Binomial Coefficients – binomial

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.

For $N = 6$, this number is 15.

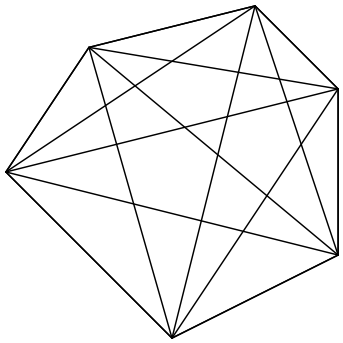


Figure 13.1: A polygon with 4 vertices.

Given N, determine how many pairs of segments intersect.

Input

The integer $3 \leq N \leq 100$.

Output

The number of pairs of segments that intersect.

The problem is a classical counting problem. 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[2] suggests that the answer is the binomial coefficient $\binom{N}{4}$. While this certainly is a legit strategy when solving problems on your own, this approach is usually not applicable at contests where access to the Internet tend to be restricted.

Instead, let us find some kind of bijection between the objects we count (intersections of line segments) with something easier to count. This strategy is one of the basic principles of combinatorial counting. An intersection is defined by two line segments,

[2] `https://oeis.org/A000332`

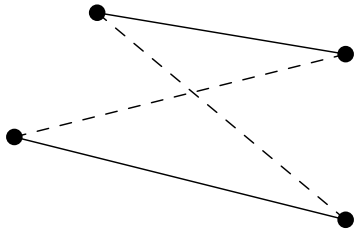


Figure 13.2: Four points taken from Figure 13.1.

of which there are $\binom{N}{2}$. Does every pair of segments intersect? In Figure 13.2, 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 13.2).

Conversely, any intersection of two segments give rise to such a quadrilateral – the one given by the four endpoints of the segments that intersect. Thus there exists a bijection between intersections and quadrilaterals, meaning that there must be an equal number of both. There are $\binom{N}{4}$ such choices of quadrilaterals, meaning there are also $\binom{N}{4}$ points of intersection.

Exercise 13.11

Prove that

1) $\binom{n}{k} = \binom{n-1}{k-1} + \binom{n-1}{k}$
2) $\sum_{k=0}^{n} \binom{n}{k} = 2^n$
3) $\sum_{k=0}^{n} (-1)^k \binom{n}{k} = 0$
4) $\sum_{k=0}^{n} \binom{n}{k} 2^k = 3^n$
5) $\sum_{k=0}^{n} \left[\binom{n}{k} \left(\sum_{l=0}^{k} \binom{k}{l} 2^l \right) \right] = 4^n$

13.4.1 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 this spirit of this chapter, we ask how many Dyck paths there are in a grid

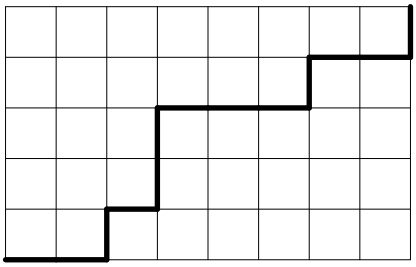


Figure 13.3: A Dyck path on a grid of width 8 and height 5.

of size $W \times H$. The solution is based on two facts: a Dyck path consists of exactly $H+W$ moves, and exactly H of those should be northbound moves, and W eastbound. Conversely, any path consisting of exactly $H + W$ moves where exactly H of those are northbound moves is a Dyck path.

If we consider e.g. the Dyck path in Figure 13.3, we can write down the sequence of moves we made, with the symbol N for northbound moves and E for eastbound moves:

EENENNEEENEEN

Such a sequence must consist of all $H + W$ moves, with exactly H "N"-moves. There are exactly $\binom{H+W}{H}$ such sequences, since this is the number of ways we can choose the subset of positions which should contain the N moves.

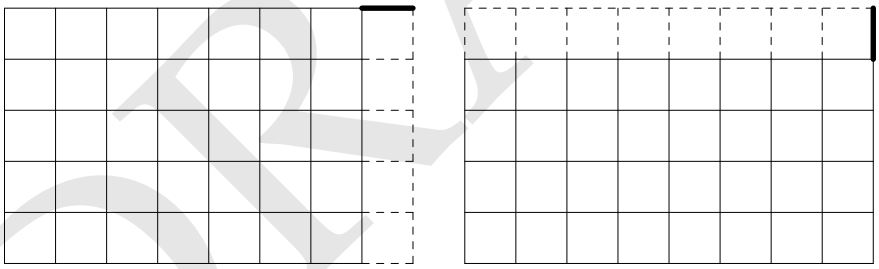


Figure 13.4: The two options for the last possible move in a Dyck path.

If we look at Figure 13.3, 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 13.11).

Exercise 13.12

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

In how many ways can the numbers $0 \leq a_1, a_2, \ldots, a_k$ be chosen such that

$$\sum_{i=1}^{k} a_i = n$$

Input

The integers $0 \leq n \leq 10^6$ and $0 \leq k \leq 10^6$.

Output

Output the number of ways modulo $10^9 + 7$.

Given a Dyck path such as the one in Figure 13.3, 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 13.5 shows this connection explicitly.

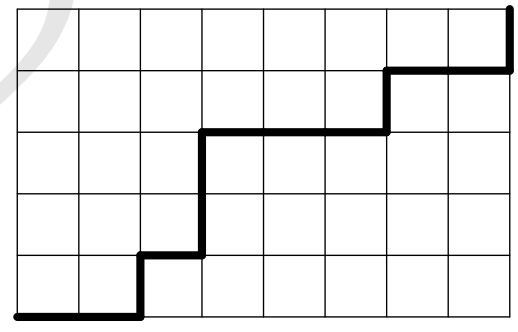


Figure 13.5: A nine-term sum as a Dyck path.

This mapping gives us a bijective mapping between sums of k terms with a sum of n, to Dyck paths on a grid of size $(k-1) \times n$. We already know how many such Dyck paths there are: $\binom{n+k-1}{n}$.

13.4.2 Catalan Numbers

A special case of the Dyck paths are the paths on a square grid that *do not cross the diagonal* of the grid. See Figure 13.6 for an example.

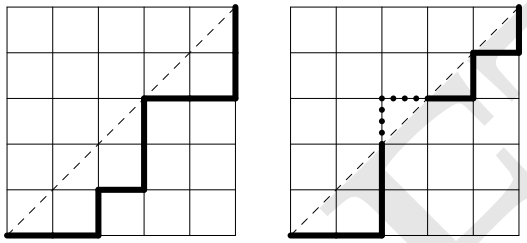


Figure 13.6: A valid path (left) and an invalid path (right).

We are now going to count the number of such paths, the most complex counting problem we have encountered so far. It turns out 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 13.6, 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 13.7.

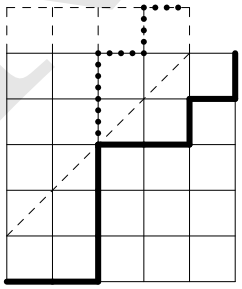


Figure 13.7: Mirroring the part of the Dyck path after its first diagonal crossing.

We claim that when mirroring the remainder of the path in this translated diagonal, we will 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 13.3 — Catalan Numbers
The number of Dyck paths in an $\times n$ grid is called the n'th ***Catalan number***

$$C_n = \binom{2n}{n} - \binom{2n}{n+1} = \binom{2n}{n} - \frac{n}{n+1}\binom{2n}{n} = \frac{1}{n+1}\binom{2n}{n}$$

The first few Catalan numbers[3] are $1, 1, 2, 5, 14, 42, 132, 429, 1430$.

Exercise 13.13 — Kattis Exercises

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.

13.5 The Principle of Inclusion and Exclusion

Often, we wish to compute the size of the union of a collection of sets $S_1, S_2, \dots, S_n$, where these sets are not pairwise disjoint. For this, the *principle of inclusion and exclusion* was developed.

Let us consider the most basic case of the principle, using two sets A and B. If we wish 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. Fortunately, this is easily mitigated: the number of elements in both sets equals $|A \cap B|$ (Figure 13.8). 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

[3] https://oeis.org/A000108

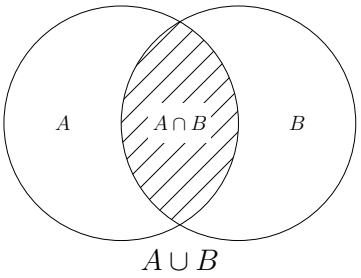


Figure 13.8: The union of two sets A and B.

intersections too many times, so we remove those and get

$$|A| + |B| + |C| - |A \cap B| - |A \cap C| - |B \cap C|$$

This time, however, 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 13.14

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:

$$\left|\bigcup_{i=1}^{n} S_i\right| = \sum_i |S_i| - \sum_{i<j} |S_i \cap S_j| + \sum_{i<j<k} |S_i \cap S_j \cap S_k| - \cdots + (-1)^{n+1} |S_1 \cap S_2 \cap \cdots \cap S_n|$$

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. The principle is based on a very important assumption – that it is easier to compute intersections of sets than their unions. Whenever this is the case, you might want to consider if the principle is applicable.

Derangements

Compute the number of permutations π of length N such that $\pi(i) \neq i$ for every $i = 1 \ldots N$.

This is a typical application of the principle. We will use it on those sets of permutations be 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 is $D_1 \cup D_2 \cup \cdots \cup D_N$. This means we seek $N! - |D_1 \cup \cdots \cup D_N|$. To apply the inclusion and exclusion formula, we must be able to compute the size of intersections of the subsets of D_i. This task is simplified greatly since the intersection of k such subsets is entirely symmetrical (it does not matter for which elements the condition is false, only the number).

If we want to compute the intersection of k such subsets, this means that there are k indices i where $\pi(i) = i$. 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 will evaluate to $\binom{N}{k}(N - k)! = \frac{N!}{k!}$. Thus, the formula can be simplified to

$$\frac{N!}{1!} - \frac{N!}{2!} + \frac{N!}{3!} - \dots$$

Subtracting this from $N!$ means that there are

$$N!(1 - 1 + \frac{1}{2!} - \frac{1}{3!} + \dots)$$

This gives us a $\Theta(N)$ algorithm to compute the answer.

It is possible to simplify this further, using some insights from calculus. We have that

$$e^{-1} = 1 - 1 + \frac{1}{2} - \frac{1}{3} + \dots$$

Then, we expect that the answer should converge to $\frac{N!}{e}$. As it happens, the answer will *always* be $\frac{N!}{e}$ rounded to the nearest integer.

Exercise 13.15

8 persons are to be seated around a circular table. The company is made up of 4 married couples, where the two members of a couple prefer not to be seated next to each other. How many possible seating arrangements are possible, assuming the cyclic rotations of an arrangement are considered equivalent?

13.6 Invariants

Many problems deal with processes which consist of many steps. During such processes, we are often interested in certain properties that never change. We call such a property an *invariant*. For example, consider the binary search algorithm to

find a value in a sorted array. During the execution of the algorithm, we maintain the invariant that the value we are searching for must be contained in some given segment of the array indexed by $[lo, hi)$ at any time. The fact that this property is invariant basically constitutes the entire proof of correctness of binary search. Invariants are tightly attached to greedy algorithms, and is a common tool used in proving correctness of various greedy algorithms. They are also one of the main tools in proving impossibility results (for example when to answer *NO* in decision problems).

Permutation Swaps

Given is a permutation a_i of $\langle 1, 2, ..., N \rangle$. Can you perform exactly K *swaps*, i.e. exchanging pairs of elements of the permutation, to obtain the identity permutation $\langle 1, 2, ..., N \rangle$?

Input

The first line of input contains the size of the permutation $1 \leq N \leq 100\,000$. The next line contains N integers separated, the permutation $a_1, a_2, ..., a_N$.

Output

Output `YES` if it is possible, and `NO` if it is impossible.

First, we need to compute the minimum number of swaps needed.

Assume the cycle decomposition of the permutation consists of C cycles (see 13.2.1 for a reminder of this concept), with lengths $b_1, b_2, .., b_C$. Then, we need at least

$$S = \sum_{i=1}^{C} b_i - 1$$

swaps to return it to the identity permutation, a fact you will be asked to prove in the next section on monovariants. This gives us one necessary condition: $K \geq S$. However, this is not sufficient. A single additional condition is needed – that S and K have the same parity! To prove this, we will look at the number of *inversions* of a permutation, one of the common invariant properties of permutations.

Given a permutation a_i, we say that the pair (i, j) is an inversion if $i < j$, but $a_i > a_j$. Intuitively, it is the number of pairs of elements that are "out of place" in relation to each other.

If we look at Figure 13.9, where we started out with a permutation and performed a number of swaps (transforming it to the identity permutation), we can spot a simple invariant. The parity of the number of swaps and the number of inversions seems to always be the same. This characterization of a permutation is called *odd* and *even* permutations depending on whether the number of inversions is odd or even. Let us prove that this invariant actually holds.

3	5	2	1	4	6 inversions
1	5	2	3	4	3 inversions
1	5	3	2	4	4 inversions
1	5	3	4	2	5 inversions
1	2	3	4	5	0 inversions

Figure 13.9: The number of inversions for permutations differing only by a single swap.

If this is the case, it is obvious why S and K must have the same parity. Since S is the number of swaps needed to transform the identity permutation to the given permutation, it must have the same parity as the number of inversions. By performing K swaps, K must have the same parity as the number of inversions. As K and S must have the same parity as the number of inversions, they must have the same parity as each other.

To see why these two conditions are sufficient, we can, after performing S swaps to obtain the identity permutation, simply swap two numbers with each other the remaining swaps. This can be done since $K - S$ will be an even number due to their equal parity.

13.7 Monovariants

Another similar tool (sometimes called a *monovariant*) instead define some kind of value $p(v)$ to the state v of each step of the process. We choose p such that it is strictly increasing or decreasing. They are mainly used to prove the finiteness of a process, in which either:

- The value function assume e.g. integer values, and is easily bounded in the direction of monotonicity (e.g. an increasing function would have an upper bound).
- The value function assume can assume any real, but there are only finitely many states the process can be in. In this case, the monovariant is used to prove that the process will never return to a previous state since this would contradict the monotonicity of p.

Let us begin with a famous problem of the first kind.

Majority Graph Bipartitioning

Given is a graph G. Find a bipartition of this graph into parts U and V, such that every vertex v has at most $\frac{|N(v)|}{2}$ neighbors in the same part as v itself.

Input

The first line of input contains integers $1 \leq V \leq 100$ and $0 \leq E \leq \frac{V(V-1)}{2}$ – the number of vertices and edges respectively. The next E lines contain two integers $0 \leq a \neq b < V$, the zero-indexed numbers of two vertices that are connected by an edge. No pair of vertices will have two edges.

Output

Output N integers, one for each vertex. The i'th integer should be 1 or 2 if the i'th vertex is in the first or the second part of the partition, respectively.

As an example, consider the valid and invalid partitionings in Figure 13.10. The vertices which does not fulfill the neighbor condition are marked in gray.

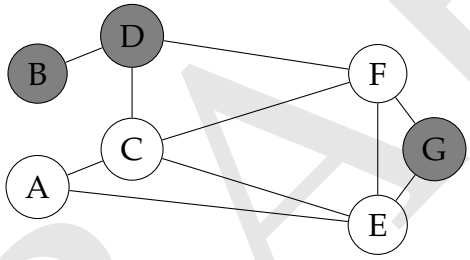


Figure 13.10: An invalid bipartitioning, where vertices B, D, G break the condition.

Problems generally considered greedy algorithms and pure monovariant problems usually differ in that the choice of next action usually has less thought behind it in the monovariant problems. We will often focus not on optimally short sequences of choices as we do with greedy algorithms, but merely finding *any* valid configuration. For example, in the problem above, one might try to construct a greedy algorithm based on for example the degrees of the vertices, which seems reasonable. However, it turns out there is not enough structure in the problem to find any simple greedy algorithm to solve the problem.

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. 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 whenever such a modification is done.

4. Prove that this value cannot decrease 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 $p(s)$. Step 4 usually follows from one of the two value functions discussed previously. 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 will work as follows. First, consider any bipartition of the graph. Assume that this graph does not fulfill the neighbor condition. Then, there must exist a vertex v which has more than $\frac{|N(v)|}{2}$ vertices in the same part as v itself. Whenever such a vertex exists, we move any of them to the other side of the partition. See Figure 13.11 of the this process.

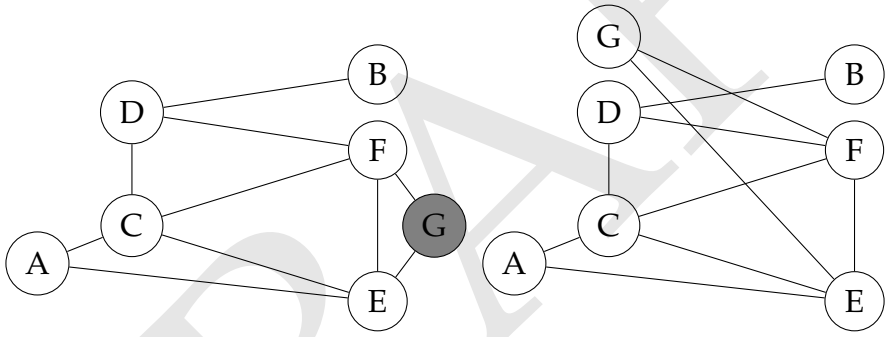


Figure 13.11: Two iterations of the algorithm, which brings the graph to a valid state.

One question remains – why does this move guarantee a finite process? We now have a general framework to prove such things, which suggests that perhaps we should look for a value function $p(s)$ which is either strictly increasing or decreasing as we perform an action. By studying the algorithm in action in Figure 13.11 we might notice that more and more edges tend to go between the two parts. In fact, this number never decreased in our example, and it turns out this is always the case.

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 will no longer be between the two parts, while the a edges to its neighbors in the same part will. This means the number of edges between the parts will change by $a - b > 0$. Thus, we can choose this as our value function. Since this is an integer function with the obvious upper bound of E, we complete step 4 of our proof technique and can thus conclude the final state must be admissible.

In mathematical problem solving, monovariants are usually used to prove that the an

admissible state exists. However, such problems are really algorithmic problems in disguise, since they actually provide an algorithm to construct such an admissible state.

Let us complete our study of monovariants, by also showing a problem using the second value function rule.

Water Pistols

N girls and N boys stand on a large field, with no line going through three different children.

Each girl is equipped with a water pistol, and wants to pick a boy to fire at. While the boys probably will not appreciate being drenched in water, at least the girls are a fair menace – the will only fire at a single boy each. Unfortunately, it may be the case that two girls choose which boys to fire at in such a way that the water from their pistols will cross at some point. If this happens, they will cancel each other out, never hitting their targets.

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.

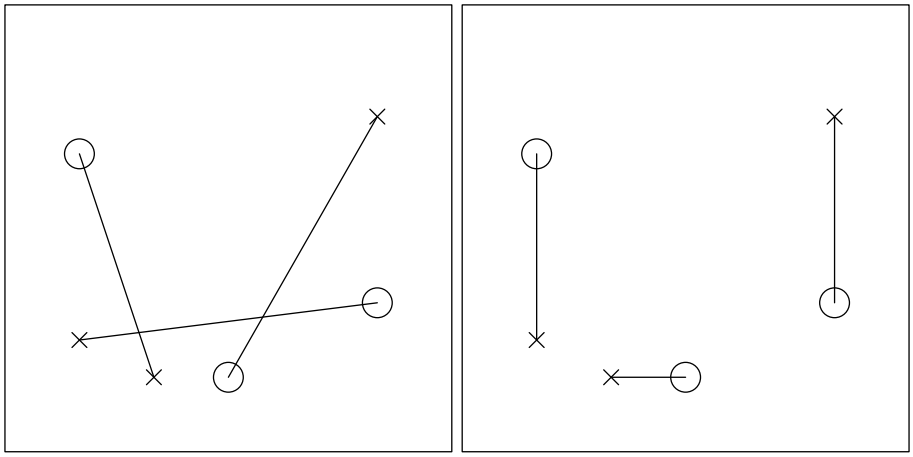


Figure 13.12: An assignment where some beams intersect (left), and an assignment where no beams intersect (right).

Input

The first line contains the integer $N \leq 200$. The next N lines contain two real numbers $-10^6 \leq x, y \leq 10^6$, separated by a space. Each line is the coordinate (x, y) of a girl. The next and final N lines contain the coordinates of the boys, in the same format.

Output

Output N lines. The i'th line should contain the zero-indexed number of the boy which the i'th girl should fire at.

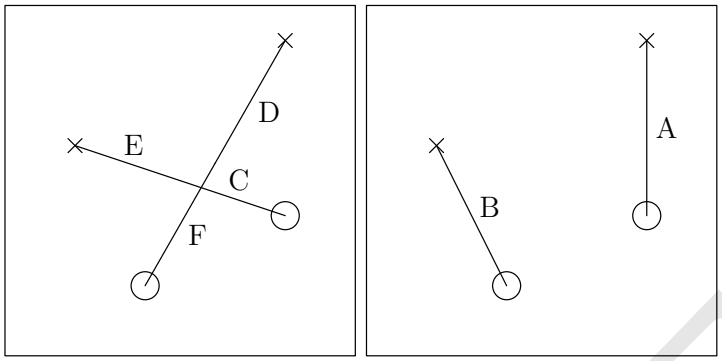


Figure 13.13: Swapping the targets of two intersecting beams.

After seeing the solution to the previous problem, the solution should not come as a surprise. We start by randomly assigning the girls to one boy each, with no two girls shooting at he same boy. If this assignment contains two girls firing water beams which cross, we simply swap their targets.

Unless you are geometrically minded, it may be hard to figure out an appropriate value function. The naive value function of counting the current number of water beams crossing unfortunately fails – and might even increase after a move.

Instead, let us look closer at what happens when we switch the targets of two girls. In Figure 13.13, 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 ($(C + D) + (E + F)$) 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 give the desired result. Thus the sum of all water beam lengths will decrease whenever we perform such a move. As students of algorithmics, we can make the additional note that this means the minimum-cost matching of the complete bipartite graph of girls and boys, with edges given as cost the distance between a particular girl and boy, is a valid assignment. If this was not the case, we would be able to swap two targets and decrease the cost of the matching, contradicting the assumption that it was minimum-cost. Thus, this rather mathematical proof actually ended up giving us a very simple reduction to min-cost matching.

Exercise 13.16 — Kattis Exercise

Army Division – armydivision

Bread Sorting – breadsorting

13.8 Chapter Notes

Chapter 14

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 subfield *computational number theory*. It is within number theory topics such as divisibility and prime numbers belong.

14.1 Divisibility

All of the number theory in this chapter relate to a single property of integers, divisibility.

Definition 14.1 — Divisibility
An integer n is ***divisible*** by an integer d if there is another integer q such that $n = dq$. We also say that d is a ***divisor*** of n.

We denote this fact with $d \mid n$.

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 by 0 with 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 14.1), we will 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 lose little information.

Example 14.1 — 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 14.1

Determine the divisors of 18.

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 – 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 – n is divisible by d if and only if $n \bmod d = 0$. The situation is not as simple for large numbers. Some programming languages, such as Java and Python, have built-in support for dealing with large integers, but e.g. C++ does not. In Section 14.4 on modular arithmetic, we discuss the implementation of the modulo operator on large integers.

Secondly, how do we compute the divisors of a number? 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 $|d| \leq |n|$. 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 bit better though.

Almost Perfect

Baylor Competitive Learning course – David Sturgill

A positive integer p is called a perfect number if all the proper divisors of p sum to p exactly. Perfect numbers are rare; only 10 of them are known. Perhaps the definition of perfection is a little too strict. Instead, we will consider numbers that we'll call almost perfect. A positive integer p is almost perfect if the proper divisors of p sum to a value that differs from p by no more than two.

Input

Input consists of a sequence of up to 500 integers, one per line. Each integer is in the range 2 to 10^9 (inclusive).

Output

For each input value, output the same value and then one of the following: "perfect" (if the number is perfect), "almost perfect" (if it is almost perfect but not perfect), or "not perfect" (otherwise).

In this problem, computing the divisors of the numbers of the input sequence would be way too slow, requiring upwards of 10^{11} operations. Hidden in Example 14.1 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 should not be a surprise, since our definition of divisibility (Definition 14.1)– 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 14.2

Prove that an integer has an odd number of divisors if and only if it is a perfect square (except 0, which has an infinite number of divisors).

Since divisors come in pairs, we can limit ourselves to finding one member of each such pair. Furthermore, one of the elements in each such 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, except for 0, which has $\frac{0}{d} = 0$). 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. You can see the pseudo code for this in Algorithm 14.1

Algorithm 14.1: Almost Perfect

```
procedure DIVISORS(N)
    divisors ← new list
    for i from 1 up to i² ≤ N do
        if N mod i = 0 then
            divisors.add(i)
            if i ≠ N/i then
                divisors.add(N/i)
    return divisors
procedure ALMOSTPERFECT(N)
    divSum ← 0
    for d ∈ Divisors(N) do
        if d ≠ N then
            divSum ← divSum + d
    if divSum = N then
        output "perfect"
    else if |divSum − N| ≤ 2 then
        output "almost perfect"
    else
        output "not perfect"
```

This also 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 $O(n^{\frac{1}{3}})$ being a commonly used bound 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 14.1 Let $d(i)$ be the number of divisors of i. Then,

$$\sum_{i=1}^{n} d(i) \approx n \ln n$$

Proof. There are approximately $\frac{n}{i}$ 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

$$\sum_{j=1}^{n} \frac{n}{j} = n \sum_{j=1}^{n} \frac{1}{j} \approx n \ln n$$

□

This proof also suggest a way to compute the divisors of all the integers $1, 2, ..., n$ in $\Theta(n \ln n)$ time. For each integer i, we find all the numbers divisible by i (in $\Theta(\frac{n}{i})$ time), which are $0i, 1i, 2i, \ldots \lceil \frac{n}{i} \rceil i$. This is an extension of the algorithm commonly known as the ***Sieve of Eratosthenes***, an algorithm to find the objects which are our next topic of study – prime numbers.

Exercise 14.3 — Kattis Exercises

Dividing Sequence – sequence

14.2 Prime Numbers

From the concept of divisibility comes the building blocks of the integers, the famous *prime numbers*. With divisibility, we got factorizations. For example, given the number 12, we could factor it as $2 \cdot 6$, or $3 \cdot 4$, or even $2 \cdot 2 \cdot 3$. This last factorization is special, in that no matter how hard we try, it cannot be factored further. It consists only of prime numbers.

[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`.

Definition 14.2 — Prime Number
An integer $p \geq 2$ is called a ***prime number*** if its only positive divisors are 1 and p.

The numbers that are *not* prime numbers are called ***composite numbers***. There are an infinite number of primes. This be proven by a simple proof by contradiction. 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 (and by extension has no divisors but the trivial ones), so it is not composite. However, P is larger than any prime, so it is not a prime number either, a contradiction.

Example 14.2 The first 10 prime numbers are $2, 3, 5, 7, 11, 13, 17, 19, 23, 29$. Which is the next one?

Since the prime numbers have no other divisors besides the trivial ones, a factorization consisting only of prime numbers is special.

Definition 14.3 — Prime Factorization
The ***prime factorization*** of a positive integer n is a factorization of the form $p_1^{e_1} \cdot p_2^{e_2} \cdot \dots \cdot p_k^{e_k}$, where p_i are all distinct primes. This factorization is unique, except for a reordering of the p_i.

Example 14.3 The prime factorization of 228 is $2 \cdot 2 \cdot 3 \cdot 19$.

A List Game

Spotify Challenge 2010 – Per Austrin

You are playing the following simple game with a friend:

1. The first player picks a positive integer X.
2. The second player gives a list of k positive integers $Y_1, \dots, Y_k$ such that

$$(Y_1 + 1)(Y_2 + 1) \cdots (Y_k + 1) = X$$

and gets k points.

Write a program that plays the second player.

Input

The input consists of a single integer X satisfying $10^3 \leq X \leq 10^9$, giving the number picked by the first player.

Output

Write a single integer k, giving the number of points obtained by the second

player, assuming she plays as good as possible.

The problem seeks the factorization of X that contains the largest number of factors, where every factor is at least 2. This factorization must in fact be the prime factorization of X, which we will prove by contradiction. Assume that the optimal list of integers $Y_1, \ldots, Y_k$ contains a number Y_i that is composite. In this case, it can be further factored into $Y_i = ab$ with $a, b > 1$. Replacing Y_i with the numbers a and b will keep the product of the list invariant (since $Y_i = ab$), but extend the length of the list by 1, thus giving a higher-scoring sequence. The only case when this is impossible is if all the Y_i are primes, so the list we seek must indeed be the prime factorization of X.

This begs the question, how do we compute the prime factorization of a number? Mainly two different algorithms are used when computing prime factorizations: trial division and the Sieve of Eratosthenes[3]. In computing the prime factorization of a few, large numbers, trial division is used. It runs in $O(\sqrt{N})$ time, and use the same insight we used in the improved algorithm to compute the divisors of an algorithm. Since any composite number N must have a divisor less than $\sqrt{N}$, it must also have a prime divisor less than $\sqrt{N}$. Implementing this is straightforward.

The other algorithm is used to factor every number in a large interval.

Product Divisors

Given a sequence of integers $a_1, a_2, \ldots a_n$, compute the number of divisors of $A = \prod_{i=1}^{n} a_i$.

Input

Input starts with an integer $0 \leq n \leq 1\,000\,000$, the length of the sequence a_i. The next line contains the numbers $1 \leq a_1, \ldots, a_n \leq 10^6$.

Output

Output a single integer – the number of divisors of n. Since this number can be very large, output it modulo $10^9 + 7$.

Let A have the prime factorization $A = p_1^{e_1} \cdot p_2^{e_2} \cdot \cdots \cdot p_k^{e_k}$. A divisor of A must be of the form $d = p_1^{e_1'} \cdot p_2^{e_2'} \cdot \cdots \cdot p_k^{e_k'}$, where $0 \leq e_i' \leq e_i$. This can be proven using the uniqueness of the prime factorization, and the fact that $A = dq$ for some integer q. More importantly, *any* number of this form is a divisor of A, by the same cancellation argument used when proving the uniqueness of the prime factorization.

With this in hand, we apply a simple combinatorial argument to count the number of divisors of A. Each number e_i' can take any integer value between 0 and e_i to fulfill the

[3]Many more exist, but are too complicated to be used in problem solving context, although they are very much in use in real-life situations.

conditions needed to make d a divisor of A. 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. We are left with the problem of determining the prime factorization of A. Essentially, this is tantamount to computing the prime factorization of *every* integer between 1 and 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. This is implemented in Algorithm 14.2.

Algorithm 14.2: Product Divisors

```
procedure PRODUCTDIVISORS(sequence A)
    list of lists factorizations ← factorInterval(10^6 + 1)
    map counts ← new map
    for each a in A do
        for each p in factorizations[a] do
            counts[p] ← counts[p] + 1
    ans ← 1
    for each (key, value) in counts do
        ans ← (ans · (value + 1)) mod(10^9 + 7)
    return ans
```

As of now, the crucial step of factoring all integers in $[1..10^6]$ remains. This is the purpose of the Sieve of Eratosthenes. We have already seen the basic idea when computing the divisors of all numbers in the interval $[1...n]$ in Section 14.1. Extending this to factoring is only a matter of restricting the algorithm to finding prime divisors (Algorithm 14.3).

Algorithm 14.3: Sieve of Eratosthenes

```
procedure PRIMESIEVE(limit N)
    list nums ← new list[N]
    for i from 1 to N − 1 do
        nums[i] ← i
    list of lists factorizations ← new list of lists[N]
    for i from 2 to i^2 ≤ N do
        if nums[i]! = i then
            for j ∈ {i · i, i(i + 1), i(i + 2), ...} up to N − 1 do
                while nums[j] mod i = 0 do
                    factorizations[j].add(i)
                    nums[j] = nums[j]/i
    return factors
```

The complexity of this procedure is a bit tricky to analyze. The important part of the sieve is the inner loop, which computes the actual factors. Let us count the number of times a prime p is pushed in this loop. First of all, every p'th integer is divisible by p, which totals $\frac{n}{p}$ iterations. However, every p^2'th integer integer is divisible by p yet again, contributing an additional $\frac{n}{p^2}$ iterations, and so on. Summing this over every p which is used in the sieve gives us the bound

$$\sum_{p \leq \sqrt{n}} \left(\frac{n}{p} + \frac{n}{p^2} + \frac{n}{p^3} + \ldots \right) = n \sum_{p \leq \sqrt{n}} \left(\frac{1}{p} + \frac{1}{p^2} + \frac{1}{p^3} + \ldots \right)$$

Using the formula for the sum of a geometric series ($\frac{1}{p} + \frac{1}{p^2} + \ldots = \frac{p}{p-1}$) gives us the simplification

$$n \sum_{p \leq \sqrt{n}} \frac{1}{p-1} = \Theta \left(n \sum_{p \leq \sqrt{n}} \frac{1}{p} \right)$$

This last sum is out of reach to estimate with our current tools. It turns out that $\sum_{p \leq n} \frac{1}{p} = O(\ln \ln n)$. With this, the final complexity becomes a simple $O(n \ln \ln \sqrt{n}) = O(n \ln \ln n)$.

It is not uncommon for an algorithm to include in its complexity the number of primes up to a certain limit n. This function is known as $\pi(n)$, the prime counting function. We have $\pi(10^3) = 168, \pi(10^6) = 78\,498, \pi(10^9) \approx 51\,000\,000$. In general, $\pi(n) \approx \frac{n}{\ln n}$ for large n.

Exercise 14.4 — Kattis Exercises

Prime Sieve – primesieve

Happy Happy Prime Prime – happyprime

Prime Path – primepathc

Perfect Pth Powers – perfectpowers

Factovisors – factovisors

Divisors – divisors

14.3 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 two different integers, with an extension capable of solving integer equations of the form $ax + by = c$.

Definition 14.4 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 (a, b).

As always, we ask ourselves – how do we compute it?

Greatest Common Divisor
Given to integers a and b, compute (a, b).

We already know of an $O(\sqrt{a}+\sqrt{b})$ algorithm to compute (a, b), namely to enumerate *all* divisors of a and b. Two simple identities are key to the much faster Euclidean algorithm.

$$(a, 0) = |a| \tag{14.1}$$

$$(a, b) = (a - b, b) \tag{14.2}$$

Identity 14.1 is obvious. An integer a cannot have a larger divisor than $|a|$, and this is certainly a divisor of a. Identity 14.2 needs a bit more work. We can prove their 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 $b - a$. Hence the divisors of a and b are also 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 $a > b$, we reduce the problem to a smaller one by instead computing (a, b), we compute $(a - b, b)$. This gives us a smaller problem, in the sense that $a + b$ decrease. Since both a and b are non-negative, this means we must at some point arrive at the situation where either a or b are 0. In this case, we use the base case that is Identity 14.1.

One simple but important step remains before the algorithm is useful. Note how computing $(10^9, 1)$ requires about 10^9 steps right now, since we will do the reductions $(10^9 - 1, 1), (10^9 - 2, 1), (10^9 - 3, 1)$... The fix is easy – the repeated application of subtraction of a number b from a while $a > b$ is exactly the modulo operation, meaning

$$(a, b) = (a \bmod b, b)$$

This last piece of our Euclidean puzzle complete our algorithm, and gives us a remarkably short algorithm, as seen in Algorithm 14.5.

Algorithm 14.4: Greatest Common Divisor

```
procedure GCD(A, B)
    if B = 0 then
        return A
    return GCD(B, A mod B)
```

Competitive Tip

The Euclidean algorithm exists as the built-in function `__gcd(a, b)` in C++.

Whenever dealing with divisors in a problem, the greatest common divisor may be useful. This is the case in the next problem, from the Croatian high school olympiad.

Granica

Croatian Open Competition in Informatics 2007/2008, Contest #6

Given 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.

Input

The first line contains the integer $2 \leq n \leq 100$, the length of the sequence a_i. The second line contains the integers n integers $1 \leq a_1, a_2, \ldots, a_n \leq 10^9$.

Output

Output all such integers d, separated by spaces.

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 $r - r' = 0$ so that $r = r'$ and both remainders were the same. The question then is how we compute

$$\gcd_{1 \leq i < j \leq n} a_i - a_j$$

which seek the greatest common divisor of many rather than just two numbers.

To our rescue comes the prime factor interpretation of divisors, namely that a divisor of a number

$$n = p_1^{e_1} \cdots p_k^{e_k}$$

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 $0 \leq e_i' \leq \min(f_i, e_i)$, with $e_i' = \min(f_i, e_i)$ giving us the GCD.

Using this interpretation of the GCD, we can extend the result can be extended 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 suggests the recursion formula $d = \gcd(b_1, b_2, \ldots) = \gcd(b_1, \gcd(b_2, \ldots))$.

Algorithm 14.5: Greatest Common Divisor of Several Integers

```
procedure MULTIGCD(sequence A)
    gcd ← 0
    for each a ∈ A do
        gcd ← GCD(gcd, a)
    return gcd
```

We only need one more insight to solve the problem, namely that the common divisors d of a and b are exactly the divisors of (a, b).

A related concept is given if we instead of taking the minimum of prime factors of two numbers take the maximum.

Definition 14.5 — Least Common Multiple
The ***least common multiple*** of integers a and b is the smallest integer m such that $a \mid m$ and $b \mid m$.

The computation of the LCM is basically the same as for the GCD.

A multiple d of an integer a of the form

$$a = p_1^{e_1} \cdots p_k^{e_k}$$

must be of the form

$$d = p_1^{e_1'} \cdots p_k^{e_k'}$$

where $e_i \leq e_i'$.

Thus, if d is to be a common multiple of a and

$$b = p_1^{f_1} \cdots p_k^{f_1}$$

it must be that $\max(f_i, e_i) \leq e'_i$, with $e'_i = \max(f_i, e_i)$ giving us the LCM. Since $\max(e_i, f_i) + \min(e_i, f_i) = e_i \cdot + f_i$, we must have that $\text{lcm}(a, b) \cdot \text{gcd}(a, b) = ab$. This gives us the formula $\text{lcm}(a, b) = \frac{a}{\text{gcd}(a,b)} b$ to compute the LCM. The order of operations is chosen to avoid overflows in computing the product ab.

Since max is associative just like min, the LCM operation is too, meaning

$$\text{lcm}(a, b, c, \dots) = \text{lcm}(a, \text{lcm}(b, \text{lcm}(c, \dots)))$$

Next up is the ***extended Euclidean algorithm***.

Diophantine Equation

Given integers a, b, c, find an integer solution x, y to

$$ax + by = (a, b)$$

First of all, it is not obvious such an integer solution exists. This is the case however, which we will prove constructively thus gaining an algorithm to find such a solution at the same time. The trick lies in the exact same identities used when we formulated the Euclidean algorithm. Since $(a, 0) = a$, the case $b = 0$ gives us $ax = (a, b) = a$ with the obvious solution $x = 1, y = 0$. Furthermore, we have $(a, b) = (b, a \bmod b)$. Assume we could compute a solution to $bx + (a \bmod b)y = (b, a \bmod b)$. Some rearrangement of this gives us

$$bx + (a \bmod b)y = (b, a \bmod b)$$

$$bx + (a - \lfloor \frac{a}{b} \rfloor b)y = (b, a \bmod b)$$

$$b(x - \lfloor \frac{a}{b} \rfloor y) + ay = (b, a \bmod b)$$

$$ay + b(x - \lfloor \frac{a}{b} \rfloor y)x = (b, a \bmod b)$$

$$ay + b(x - \lfloor \frac{a}{b} \rfloor y)x = (a, b)$$

meaning $(y, x - \lfloor \frac{a}{b} \rfloor y)$ would be a solution to the equation we are actually interested in.

Given the solution $[x, y]$ to the equation with coefficients $[b, a \bmod b]$ we thus get the solution $[y, x - \lfloor \frac{a}{b} \rfloor y]$ to the equation with coefficients $[a, b]$. The transformation $[b, a \bmod b] \leftrightarrow [a, b]$ happen to be the key step in the Euclidean algorithm. This suggests a recursive solution, where we first perform the ordinary Euclidean algorithm to find (a, b) and obtain the solution $[1, 0]$ to the equation $(a, b)x + 0y = (a, b)$ and then use the above trick to iteratively construct a solution to $ax + by = (a, b)$.

Example 14.4 — Extended Euclidean algorithm
Consider the equation $15x + 11y = 1$. Performing the Euclidean algorithm on these numbers we find that

$$(15, 11) = (11, 15 \bmod 11) =$$

$$(11, 4) = (4, 11 \bmod 4) =$$

$$(4, 3) = (3, 4 \bmod 3) =$$

$$(3, 1) = (1, 3 \bmod 1) =$$

$$(1, 0) = 1$$

Originally, we have the solution $[1, 0]$ to the equation with coefficients $[1, 0]$:

$$1 \cdot 1 + 0 \cdot 0 = (1, 0)$$

Using the transformation we derived gives us the solution $[0, 1 - \lfloor \frac{3}{1} 0 \rfloor] = [0, 1]$ to the equation with coefficients $[3, 1]$:

$$3 \cdot 0 + 1 \cdot 1 = (3, 1)$$

The next application gives us the solution $[1, 0 - \lfloor \frac{4}{3} \rfloor 1] = [1, -1]$ to $[4, 3]$.

$$4 \cdot 1 + 3 \cdot (-1) = (4, 3)$$

$[4, 3] \to [-1, 1 - \lfloor \frac{11}{4} \rfloor (-1)] = [-1, 3]$ which gives us

$$11 \cdot (-1) + 4 \cdot 3 = (11, 4)$$

Finally, $[-1, 3] \to [3, -1 - \lfloor \frac{15}{11} \rfloor 3] = [3, -4]$, the solution we sought

$$15 \cdot 3 + 11 \cdot (-4) = (15, 11)$$

Exercise 14.5
Find an integer solution to the equation $24x + 52y = 2$.

This gives us a single solution, but can we find all solutions? First, assume a and b are co-prime. Then, given two solutions

$$ax_1 + by_1 = 1$$

$$ax_2 + by_2 = 1$$

a simple subtraction gives us that

$$a(x_1 - x_2) + b(y_1 - y_2) = 0$$
$$a(x_1 - x_2) = b(y_2 - y_1)$$

Since a and b are co-prime, we have that

$$b \mid x_1 - x_2$$

Thus means $x_2 = x_1 + kb$ for some k. Inserting this gives us

$$a(x_1 - (x_1 + kb)) = b(y_2 - y_1)$$
$$-akb = b(y_2 - y_1)$$
$$-ak = y_2 - y_1$$
$$y_1 - ak = y_2$$

Thus, any solution must be of the form

$$(x_1 + kb, y_1 - ka) \text{ for } k \in \mathbb{Z}$$

That these are indeed solutions to the original equation can be verified by substituting x and y for these values. This result is called ***Bezout's identity***.

14.4 Modular Arithmetic

When first learning division, one is often introduced to the concept of *remainders*. For example, when diving 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 14.5 — Division with remainders
Consider division (with remainders) by 4 of the numbers $0, \ldots, 6$ We have that

$$\frac{0}{4} = 0, \text{ remainder } 0$$
$$\frac{1}{4} = 0, \text{ remainder } 1$$
$$\frac{2}{4} = 0, \text{ remainder } 2$$
$$\frac{3}{4} = 0, \text{ remainder } 3$$

$$\frac{4}{4} = 1, \text{ remainder } 0$$

$$\frac{5}{4} = 1, \text{ remainder } 1$$

$$\frac{6}{4} = 1, \text{ remainder } 2$$

Note how the remainder always increase by 1 when the nominator increased. As you might remember from Chapter 2 on C++ (or from your favorite programming language), there is an operator which compute this remainder called the *modulo operator*. Modular arithmetic is then the computation on numbers, where every number is taken modulo some integer n. Under such a scheme, we have that e.g. 3 and 7 are basically 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 14.6 — 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 14.6
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	$3 \equiv 0$	$4 \equiv 1$

*	**0**	**1**	**2**
0	0	0	0
1	0	1	2
2	0	2	$4 \equiv 1$

When we wish 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 14.7 — 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. If we instead consider multiplication in $\mathbb{Z}_4$, the situation is quite different.

*	**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! To determine when an inverse exists – and if so, computing the inverse – we will make use of the extended Euclidean algorithm. If $aa^{-1} \equiv 1 \pmod{n}$, we have $n \mid aa^{-1} - 1$, meaning $aa^{-1} - 1 = nx$ for some integer x. Rearranging this equation gives us $aa^{-1} - nx = 1$. We know from Section 14.3 that this has a solution if and only if $(a, n) = 1$. In this case, we can use the extended Euclidean algorithm to compute a^{-1}. Note that be Bezout's identity, a^{-1} is actually unique modulo n.

Just like the reals, modular arithmetic has a cancellation law regarding

Theorem 14.2 Assume $a \perp n$. Then $ab \equiv ac \pmod{n}$ implies $b \equiv c \pmod{n}$.

Proof. Since $a \perp n$, there is a number a^{-1} such that $aa^{-1} \equiv 1 \pmod{n}$.

$$ab \equiv ac \pmod{n}$$

with a^{-1} results in

$$aa^{-1}b \equiv aa^{-1}c \pmod{n}$$

Simplifying aa^{-1} gives us

$$b \equiv c \pmod{n}$$

□

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 essentially 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 $\Theta(m)$ again.

14.5 Chinese Remainder Theorem

The Chinese Remainder Theorem is probably the most useful theorem in algorithmic problem solving. It gives us a way of solving certain systems of linear equations.

Theorem 14.3 — Chinese Remainder Theorem
Given a system of equations

$$\begin{aligned} x &\equiv a_1 \pmod{m_1} \\ x &\equiv a_2 \pmod{m_2} \\ &\dots \\ x &\equiv a_m \pmod{m_n} \end{aligned}$$

where the numbers $m_1, \dots, m_n$ are pairwise relatively prime, there is a *unique* integer $x \pmod{\prod_{i=1}^{n} m_i}$ that satisfy such a system.

Proof. We will prove the theorem inductively. The theorem is clearly true for $n = 1$,

with the unique solution $x = a_1$. Now, consider the two equations

$$x \equiv a_1 \pmod{m_1}$$
$$x \equiv a_2 \pmod{m_2}$$

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 $m_1 \perp m_2$ by assumption. We then have that $x = a_1 \cdot m_2 \cdot (m_2^{-1} \bmod m_1) + a_2 \cdot m_1 \cdot (m_1^{-1} \bmod m_2) \equiv a_1 \cdot m_2 \cdot (m_2^{-1} \bmod m_1) \equiv a_i \pmod{m_1}$.

Since a solution exist for every a_1, a_2, this solution must be unique by the pigeonhole principle – 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 the theorem is true for $k - 1$ equations. Then, we can replace the equations

$$x \equiv a_1 \pmod{m_1}$$
$$x \equiv a_2 \pmod{m_2}$$

with another equation

$$x \equiv x* \pmod{m_1 m_2}$$

where $x*$ is the solution to the first two equations. We just proved those two equations are equivalent with regards to x. This reduces the number of equations to $k - 1$, which by assumption the theorem holds for. Thus, it also holds for k equations. □

Note that the theorem used an explicit construction of the solution, allowing us to find what the unique solution to such a system is.

Radar

KTH Challenge 2014

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

- within distance y_3 of x_3 modulo m_3

Input
The integers $0 \leq m_1, m_2, m_3 \leq 10^6$. The integers $0 \leq x_1, x_2, x_3 \leq 10^6$. The integers $0 \leq y_1, y_2, y_3 \leq 300$.

Output
The integer z.

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 reduce the number of times we need to use the CRT to $600^2 = 360\,000$ times, a modest amount well within the time limit.

14.6 Euler's totient function

Now that we have talked about modular arithmetic, we can give the numbers which are *not* divisors to some integer n their well-deserved attention. This discussion will start with the ϕ-function.

Definition 14.8 Two integers a and b are said to be ***relatively prime*** if their only (and thus greatest) common divisor is 1. If a and b are relatively prime, we write that $a \perp b$.

Example 14.6 The numbers 74 and 22 are not relatively prime, since they are both divisible by 2.

The numbers 72 and 65 are relatively prime. The prime factorization of 72 is $2 \cdot 2 \cdot 2 \cdot 3 \cdot 3$, and the factorization of 65 is $5 \cdot 13$. Since these numbers have no prime factors in common, they have no divisors other than 1 in common.

Given an integer n, we ask ourselves how many of the integers $1, 2, \ldots, n$ which are relatively prime to n.

Definition 14.9 — Euler's totient function
Euler's totient function $\phi(n)$ is defined as the number if integers $k \in [1, n]$ such that $(k, n) = 1$, i.e. those positive integers less than n which are co-prime to n.

Example 14.7 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 $\phi(n)$ has a property which is highly useful in computing certain number theoretical functions – it is ***multiplicative***, meaning

$$\phi(ab) = \phi(a)\phi(b) \text{ if } a \perp b$$

For multiplicative functions, we can reduce the problem of computing arbitrary values of the function to finding a formula only for prime powers. The reasoning behind the multiplicativity of ϕ is quite simple. Let $a' = a - \phi(a)$, i.e. the number of integers which do share a factor with a, and similarly $b' = b - \phi(b)$. Then, there will be ab' numbers between 1 and ab which share a factor with b. If x is one of the b' numbers sharing a factor with b, then so are $x, x + b, x + 2b, ..., x + (a-1)b$. Similarly, there will be $a'b$ numbers between 1 and ab sharing a factor with a. However, there may be some numbers sharing both a factor with a and b. Consider two such numbers $x + ib = y + ja$, which gives $ib - ja = y - x$. By Bezout's identity, this have a single solution (i, j) modulo ab, meaning every number $x + ib$ equals exactly one number $y + ja$. Thus, there were $a'b'$ numbers sharing a factor with both a and b. This means there are $ab' + a'b - a'b'$ numbers sharing a factor with either a and b, so

$$\phi(ab) = ab - ab' - a'b + a'b' = (a - a')(b - b') = \phi(a)\phi(b)$$

and we are done.

Using the multiplicativity of ϕ we get the simple formula

$$\phi(p_1^{e_1} \dots p_k^{e_k}) = \phi(p_1^{e_1}) \cdots \phi(p_k^{e_k}) = p_1^{e_1-1}(p_1 - 1) \cdots p_k^{e_k-1}(p_k - 1)$$

Computing ϕ for a single value can thus be done as quickly as factoring the number. If we wish to compute ϕ for an interval $[1, n]$ we can use the Sieve of Eratosthenes.

This seemingly convoluted function might seem useless, but is of great importance via the following theorem:

Theorem 14.4 — 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 will be to consider the product of the $\phi(n)$ positive integers less than n which are relatively prime to n. We will call these $x_1, x_2, \dots, x_{\phi(n)}$. Since these are all distinct integers between 1 and n, they are incongruent modulo n. We call such a set of $\phi(n)$ numbers, all incongruent modulo n a ***complete residue system*** (CRS) modulo n.

Next, we will prove that $ax_1, ax_2, \dots, ax_\phi$ also form a CRS modulo n. We need to show two properties for this:

1. All numbers are relatively prime to n
2. All numbers are incongruent modulo n

We will start with the first property. 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. The second property requires us to make use of the cancellation property of modular arithmetic (Theorem 14.2). 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 $ax_1, \dots, ax_{\phi(n)}$ 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. □

For primes p we get a special case of Euler's theorem when since $\phi(p) = p - 1$.

Corollary 14.1 Fermat's Theorem For a prime p and an integer $a \perp p$, we have

$$a^{p-1} \equiv 1 \pmod{p}$$

Competitive Tip

By Fermat's Theorem, we also have $a^{p-2} \equiv a^{-1} \pmod{p}$ when p is prime (and $a \perp p$). This is often an easier way of computing modular inverses modulo primes than using the extended Euclidean algorithm, in particular if you already coded modular exponentiation.

Exponial

Nordic Collegiate Programming Contest 2016

Define the *exponial* of n as the function

$$\text{exponial}(n) = n^{(n-1)^{(n-2)^{\cdots^{2^1}}}}$$

Compute $\text{exponial}(n) \pmod{m}$.

Input

The input contains the integers $1 \leq n, m \leq 10^9$.

Output

Output a single integer, the value of $\text{exponial}(n) \pmod{m}$.

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 compute $e = (n-1)^{\cdots}$ modulo $\phi(m)$ and only then compute $n^e \pmod{m}$? Alas, this is only useful when

$n \perp m$, since this is a necessary precondition for Euler's theorem. When working modulo some integer m with a prime factorization of $p_1^{e_1} \cdots p_k^{e_k}$, a helpful approach is to instead work modulo its prime powers $p_i^{e_i}$ and then combine the results using the Chinese remainder theorem. Since the prime powers of a prime factorization is relatively prime, the remainder theorem applies.

Let us apply this principle to Euler's theorem. When computing $n^e \bmod p^k$ we have two cases. Either $p \mid n$, in which case $n^e \equiv 0 \pmod{p^k}$ whenever $e \geq k$. Otherwise, $p \perp n$, and $n^e \equiv n^{e \bmod \phi(p^k)} \pmod{p^k}$ by Euler's theorem.

This suggests that if $e \geq \max(e_1, \ldots, e_k)$, we have that n^e is indeed periodic. Furthermore, e_i is bounded by $\log_2 n$, since

$$\begin{aligned} p_i^{e_i} &\leq n \Rightarrow \\ e_i \log_2(p_i) &\leq \log_2(N) \Rightarrow \\ e_i &\leq \frac{\log_2(N)}{\log_2(p_i)} \leq \log_2(N) \end{aligned}$$

As $p_i \geq 2$, we know that $\log_2(p_i) \geq 1$, which we used in the final inequality. Since $\log_2(10^9) \approx 30$ and $4^{3^{2^1}} \geq 30$, we can use the periodicity of n^e whenever $n \geq 5$.

$$n^{\phi(m)+\text{exponial}(n-1) \bmod \phi(m)} \bmod m$$

For $n = 4$, the exponial equals only 262144, meaning we can compute it immediately.

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 modulo 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. That this sequence decrease very quickly is based on two facts. For even m, $\phi(m) = \phi(2)\phi(\frac{m}{2}) = \phi(\frac{m}{2}) \leq \frac{m}{2}$. 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 $\phi(\phi(m)) \leq \frac{m}{2}$ for $m > 1$ (1 is neither even nor contains an odd prime factor). This means the modulo will become 1 in a logarithmic number of iterations, completing our algorithm.

14.7 Chapter Notes

A highly theoretical introduction to classical number theory can be found in *An Introduction to the Theory of Numbers*[10] While devoid of exercises and examples, it is

very comprehensive.

A Computational Introduction to Number Theory and Algebra[20] instead takes a more applied approach, and is freely available under a Creative Commons license at the authors home page.[4].

[4] http://www.shoup.net/ntb/

Chapter 15

Competitive Programming 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.

15.1 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.

15.1.1 Strategy

Very few contestants manage to solve every problem fully during an IOI contest. There is a very high probability you are 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 if 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.

15.1.2 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

[1] `https://people.ksp.sk/~misof/ioi-syllabus/`
[2] `http://hsin.hr/coci/`
[3] `http://main.edu.pl/en/archive/oi`

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, 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.

15.2 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 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.

15.2.1 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 which 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 reasonable 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.

15.2.2 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, Codeforceshas 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.

http://
codeforces.c

Appendix A

Mathematics

This appendix reviews some basic mathematics. Without a good grasp on the foundations of mathematics, algorithmic problem solving is basically impossible. When we analyze the efficiency of algorithms, we use sums, recurrence relations and a bit of algebra. One of the most common topics of algorithms is graph theory, an area within discrete mathematics. Without some basic topics, such as set theory, some of the proofs – and even problems – in this book will be hard to understand.

This mathematical preliminary touches lightly upon these topics and is meant to complement a high school education in mathematics in preparation for the remaining text. While you can probably get by with the mathematics from this chapter, we highly recommend that you (at some point) delve deeper into discrete mathematics.

We do assume that you are familiar with basic proof techniques, such as proofs by induction or contradiction, and mathematics that is part of a pre-calculus course (trigonometry, polynomials, etc). Some more mathematically advanced parts of this book will go beyond these assumptions, but this is only the case in very few places.

A.1 Logic

In mathematics, we often deal with truths and falsehoods in the form of theorems, proofs, counter-examples and so on. Logic is a very exact discipline, and a precise language has been developed to help us deal with logical statements.

For example, consider the statements

1. an integer is either odd or even,
2. programming is more fun than mathematics,

3. if there exists an odd number divisible by 6, every integer is even,
4. x is negative if and only if x^3 is negative,
5. every apple is blue,
6. there exists an integer.

The first statement uses the conjunction *or*. It connects two statements, and requires only one of them to be true in order for the whole statement to be true. Since any integer is either odd or even, the statement is true.

The second statement is not really a logical statement. While we might have a personal conviction regarding the entertainment value of such topics, it is hard to consider the statements as having a truth value.

The third statement complicates matters by introducing an *implication*. It is a two-part statement, which only makes a claim regarding the second part if the first part is true. Since no odd number divisible by 6 exists, it makes no statement about the evenness of every integer. Thus, this implication is true.

The fourth statement tells us that two statements are equivalent – one is true exactly when the other is. This is also a true statement.

The fifth statement concerns *every* object if some kind. It is a false statement, a fact that can be proved by exhibiting e.g., a green apple.

Finally, the last statement is true. It asks whether something exists, a statement we can prove by presenting an integer such as 42.

To express such statements, a language has been developed where all these logical operations such as existence, implication and so on have symbols assigned to them. This enables us to remove the ambiguity inherent in the English language, which is of utmost importance when dealing with the exactness required by logic.

The ***disjunction*** (a is true **or** b is true) is a common logical connective. It is given the symbol $\vee$, so that the above statement is written as $a \vee b$. The other common connective, the ***conjunction*** (a is true **and** b is true) is assigned the symbol $\wedge$. For example, we write that $a \wedge b$ for the statement that both a and b are true.

An ***implication*** is a statement of the form "if a is true, then b must also be true". This is a statement on its own, which is true whenever a is false (meaning it does not say anything of b), or when a is true and b is true. We use the symbol $\rightarrow$ for this, writing the statement as $a \rightarrow b$. The fourth statement would hence be written as

$$(\exists p : p \text{ is prime} \wedge p \text{ is divisible by } 6) \rightarrow \forall \text{ prime } p : p \text{ is even}$$

The next statement introduced the ***equivalence***, a statement of the form "a is true if, and only if, b is true". This is the same as $a \rightarrow b$ (the only if part) and $b \rightarrow a$ (the if

part). We use the symbol $\leftrightarrow$, which follows naturally for this reason. The statement would then be written as

$$x < 0 \leftrightarrow x^3 < 0$$

Logical also contains ***quantifiers***. The fifth statement, that every apple is blue, actually makes a large number of statements – one for each apple. This concept is captured using the ***universal quantifier*** $\forall$, read as "for every". For example, we could write the statement as

$$\forall \text{ apple } a : a \text{ is blue}$$

In the final statement, another quantifier was used, which speaks of the existence of something; the ***existential quantifier*** $\exists$, which we read as "there exists". We would write the second statement as

$$\exists x : x \text{ is an integer}$$

The ***negation*** operator $\neg$ inverts a statement. The statement "no penguin can fly" would thus be written as

$$\neg(\exists \text{ penguin } p : p \text{ can fly})$$

or, equivalently

$$\forall \text{ penguin } p : \neg p \text{ can fly}$$

Exercise A.1

Write the following statements using the logical symbols, and determine whether they are true or false:

1) If a and b are odd integers, $a + b$ is an even integer,

2) a and b are odd integers if and only if $a + b$ is an even integer,

3) Whenever it rains, the sun does not shine,

4) ab is 0 if and only if a or b is 0

Our treatment of logic ends here. Note that much is left unsaid – it is the most rudimentary walk-through in this chapter. This section is mainly meant to give you some familiarity with the basic symbols used in logic, since they will appear later. If you wish to gain a better understanding of logic, you can follow the references in the chapter notes.

A.2 Sets and Sequences

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 would 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 would write

$$\{p \mid p \text{ is prime}\}$$

Certain 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 $\{1, 2, \ldots, n\}$,
- $\emptyset$ – the empty set.

Exercise A.2

1) Use the set builder notation to describe the set of all odd integers.
2) Use the set builder notation to describe the set of all negative integers.
3) Compute the elements of the set $\{k \mid k \text{ is prime and } k^2 \leq 30\}$.

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 $A \subseteq B$. 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 $\emptyset \subseteq S$ and $S \subseteq S$. Whenever a set A is not a subset of another set B, we write that $A \not\subseteq B$. For example,

$$\{2, \pi\} \not\subseteq \mathbb{Q}$$

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.

Exercise A.3

1) List all subsets of the set $\{1, 2, 3\}$.
2) How many subsets does a set containing n elements have?
3) Determine which of the following sets are subsets of each other:
   - $\emptyset$
   - $\mathbb{Z}$
   - $\mathbb{Z}_+$
   - $\{2k \mid k \in \mathbb{Z}\}$
   - $\{2, 16, 12\}$

Sets also 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, i.e.,

$$x \in A \cap B \Leftrightarrow x \in A \wedge x \in B$$

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 those elements which are members of **either** set.

Example A.1 Let

$$X = \{1, 2, 3, 4\}, Y = \{4, 5, 6, 7\}, Z = \{1, 2, 6, 7\}$$

Then,

$$X \cap Y = \{4\}$$
$$X \cap Y \cap Z = \emptyset$$
$$X \cup Y = \{1, 2, 3, 4, 5, 6, 7\}$$
$$X \cup Z = \{1, 2, 3, 4, 6, 7\}$$

Exercise A.4

Compute the intersection and union of:

1) $A = \{1, 4, 2\}, B = \{4, 5, 6\}$
2) $A = \{a, b, c\}, B = \{d, e, f\}$
3) $A = \{\text{apple}, \text{orange}\}, B = \{\text{pear}, \text{orange}\}$

A *sequence* is an **ordered** collection of values (predominantly numbers) such a $1, 2, 1, 3, 1, 4, \ldots$. 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.3 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 $\ldots$ 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

$$\sum_{i=1}^{6} 2i$$

Exercise A.5

Compute the sum

$$\sum_{i=-2}^{4} 2 \cdot i - 1$$

Many useful sums have closed forms – expressions in which we do not need sums of a variable number of terms.

Exercise A.6

Prove the following identities:

$$\sum_{i=1}^{n} c = cn$$

$$\sum_{i=1}^{n} i = \frac{n(n+1)}{2}$$

$$\sum_{i=1}^{n} i^2 = \frac{n(n+\frac{1}{2})(n+1)}{3}$$

$$\sum_{i=0}^{n} 2^i = 2^{n+1} - 1$$

The sum of the inverses of the first n natural numbers happen to have a very neat approximation, which we will occasionally make use of later on:

$$\sum_{i=1}^{n} \frac{1}{n} \approx \ln n$$

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

$$\prod_{i=1}^{n} 2i - 1$$

Exercise A.7

Prove that

$$(n+2)\prod_{i=1}^{n} i + \frac{n}{n+1}\prod_{i=1}^{n+2} i = \prod_{i=1}^{n+2} i$$

A.4 Graphs

Graphs are one of the most common objects of study in algorithmic problem solving. They are an abstract way of representing various types of relations, such as roads between cities, friendships, computer networks and so on. Formally, we say that a graph consists of two things – a set V of ***vertices***, and a set E of ***edges***. An edge consists of a pair of vertices $\{u, v\}$, which are called the ***endpoints*** of the edge. We say that the two endpoints of the edge are ***connected*** by the edge.

A graph lends itself naturally to a graphical representation. For example, the graph given by $V = \{1, 2, 3, 4, 5\}$ and $E = \{\{1, 2\}, \{3, 1\}, \{4, 2\}, \{4, 1\}\}$ can be drawn as in Figure A.1.

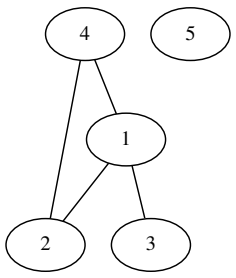


Figure A.1: The graph given by $V = \{1, 2, 3, 4, 5\}$ and $E = \{\{1, 2\}, \{3, 1\}, \{4, 2\}, \{4, 1\}\}$.

Graphs bring a large vocabulary with them. While you might find it hard to remember all of them now, you will become intimately familiar with them when studying graph theoretical algorithms later on.

A *path* is a sequence of *distinct* vertices p_0, p_1, ..., p_{l-1}, p_l such that $\{p_i, p_{i+1}\} \in E$ $(i = 0, 1, ..., l-1)$. This means any two vertices on a path must be connected by an edge. We say that this path has ***length*** l, since it consists of l edges. In Figure A.1, the sequence $3, 1, 4, 2$ is a path of length 3.

If we relax the constraint that a path must contain only distinct vertices, we instead get a ***walk***. A walk which only contain *distinct edges* is called a ***trail***. The graph in Figure A.1 contains the walk $1, 3, 1, 2$ (which is not a trail, since the edge $\{1, 3\}$ is used twice). $3, 1, 4, 2, 1$ on the other hand is a trail.

If a path additionally satisfy that $\{p_0, p_l\} \in E$, we may append this edge to make the path cyclical. This is called a *cycle* . Similarly, a walk with starts and ends at the same vertex is called a ***closed walk***. If a trail starts and ends at the same vertex, we call it a ***closed trail***.

A graph where any pair of vertices have a path between then is called a ***connected*** graph. The (maximal) subsets of a graph which *are* connected form the ***connected components*** of the graph. In Figure A.1, the graph consists of two components, $\{1, 2, 3, 4\}$ and $\{5\}$.

A ***tree*** is a special kind of graph – a **connected** graph which does not contain any cycle. The graph in Figure A.1 is not a tree, since it contains the cycle $1, 2, 4, 1$. The graph in Figure A.2 on the other hand, contains no cycle.

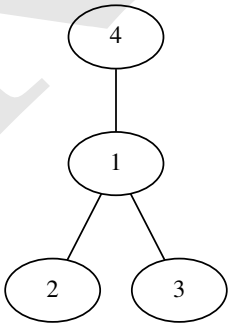


Figure A.2: The tree given by $V = \{1, 2, 3, 4\}$ and $E = \{\{1, 2\}, \{3, 1\}, \{4, 1\}\}$.

Exercise A.8

Prove that a tree of n vertices have exactly $n - 1$ edges.

So far, we have only considered the ***undirected*** graphs, in which an edge simply

connects two vertices. Often, we want to use graphs to model asymmetric relations, in which an edge should be given a direction – it should go *from* a vertex *to* another vertex. This is called a ***directed*** graph. In this case, we will write edges as ordered pairs of vertices (u, v), where the edge goes from u to v. When representing directed graphs graphically, edges will be arrows, with the arrowhead pointing from u to v (Figure A.3).

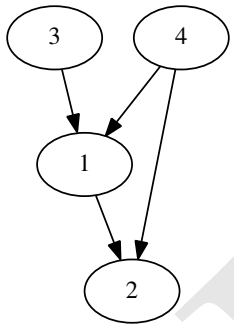


Figure A.3: The graph given by $V = \{1, 2, 3, 4\}$ and $E = \{(1, 2), (3, 1), (4, 2), (4, 1)\}$.

Terms such as cycles, paths, trails and walks translate to directed graphs in a natural way, except we must follow edges in their direction. Thus, a path is a sequence of edges $(p_0, p_1), (p_1, p_2), ..., (p_{l-1}, p_l)$. In the graph in Figure A.3, you may notice that there is no directed cycle. We call such a graph a ***directed acyclic graph*** (DAG). This type of graph will be a recurring topic, so it is a term important to be well acquainted with.

Another augmentation of a graph is the ***weighted graph***. In a weighted graph, each edge $e \in E$ is assigned a weight $w(e)$. This will often represent a length or a cost of the edge. For example, when using a graph to model a network of roads, we may associate each road $\{u, v\}$ between two locations u and v with the length of the road $w(\{u, v\})$.

So far, we have only allowed E to be a set. Additionally, an edge has always connected to different vertices (in contrast to a self-loop, an edge from a vertex to itself). Such a graph is sometimes called a ***simple*** graph. Generally, our examples of graphs will be simple graphs (even though most of our algorithms can handle duplicate edges and self-loops). A graph where E may be a multiset, or contain self-loops is called a *multigraph*.

A.5 Chapter Notes

If you need a refresher on some more basic mathematics, such as single-variable calculus, *Calculus* [22] 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.

Logic in computer science [11] 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.

For a gentle introduction to discrete mathematics, *Discrete and Combinatorial Mathematics: An Applied Introduction* [9] by Ralph Grimaldi is a nice book with a lot of breadth.

One of the best works on discrete mathematics ever produced for the aspiring algorithmic problem solver is *Concrete Mathematics* [7], 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.

Graph Theory [6] by Reinhard Diestel is widely acknowledged as the go-to book on more advanced graph theory concepts. The author is freely available for viewing at the book's home page[1].

[1] `http://diestel-graph-theory.com/`

Bibliography

[1] Sanjeev Arora and Boaz Barak. *Computational Complexity: A Modern Approach*. Cambridge University Press, 2009.

[2] David Beazley and Brian K. Jones. *Python Cookbook*. O'Reilly, 2013.

[3] Joshua Bloch. *Effective Java*. Pearson Education, 2008.

[4] 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.

[5] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. *Introduction to Algorithms*. The MIT Press, 3rd edition, 2009.

[6] Reinhard Diestel. *Graph Theory*. Springer, 2016.

[7] Oren Patashnik Donald E. Knuth and Ronald Graham. *Concrete Mathematics: A Foundation for Computer Science*. Addison-Wesley, 1994.

[8] Philippe Flajolet and Robert Sedgewick. *An Introduction to the Analysis of Algorithms*. Addison-Wesley, 2013.

[9] Ralph P. Grimaldi. *Discrete and Combinatorial Mathematics: An Applied Introduction*. Pearson Education, 2003.

[10] G.H Hardy and E.M Wright. *An Introduction to the Theory of Numbers*. Oxford University Press, 2008.

[11] Michael Huth. *Logic in Computer Science*. Cambridge University Press, 2004.

[12] George S. Lueker. *Two NP-complete Problems in Nonnegative Integer Programming*. Princeton University. Department of Electrical Engineering, 1975.

[13] Robert C. Martin. *Clean Code: A Handbook of Agile Software Craftsmanship*. Pearson Education, 2009.

[14] Steve McConnell. *Code Complete: A Practical Handbook of Software Construction*. Microsoft Press, 2004.

[15] Scott Meyers. *Effective STL*. O'Reilly, 2001.

[16] Scott Meyers. *Effective C++*. O'Reilly, 2005.

[17] Scott Meyers. *Effective Modern C++*. O'Reilly, 2014.

[18] Christos Papadimitriou. *Computational Complexity*. Addison-Wesley, 1994.

[19] Charles Petzold. *CODE*. Microsoft Press, 2000.

[20] Victor Shoup. *A Computational Introduction to Number Theory and Algebra*. Cambridge University Press, 2008.

[21] Brett Slatkin. *Effective Python*. Addison-Wesley, 2015.

[22] Michael Spivak. *Calculus*. Springer, 1994.

[23] Bjarne Stroustrup. *The C++ Programming Language*. Addison-Wesley, 2013.

[24] Jeffrey Ullman and John Hopcroft. *Introduction to Automata Theory, Languages, and Computation*. Pearson Education, 2014.

Index