Competitive Programmer's Handbook

Antti Laaksonen

Draft February 2017

Contents

Preface

The purpose of this book is to give you a thorough introduction to competitive programming. It is assumed that you already know the basics of programming, but previous background on competitive programming is not needed.

The book is especially intended for students who want to learn algorithms and possibly participate in the International Olympiad in Informatics (IOI) or the International Collegiate Programming Contest (ICPC). Of course, the book is also suitable for anybody else interested in competitive programming.

It takes a long time to become a good competitive programmer, but it is also an opportunity to learn a lot. You can be sure that you will get a good general understanding of algorithms if you spend time reading the book, solving problems and taking part in contests.

The book is under continuous development. You can always send feedback on the book to `ahslaaks@cs.helsinki.fi`.

Helsinki, February 2017
Antti Laaksonen

x

Part I

Basic techniques

Chapter 1

Introduction

Competitive programming combines two topics: (1) the design of algorithms and (2) the implementation of algorithms.

The **design of algorithms** consists of problem solving and mathematical thinking. Skills for analyzing problems and solving them creatively are needed. An algorithm for solving a problem has to be both correct and efficient, and the core of the problem is often how to invent an efficient algorithm.

Theoretical knowledge of algorithms is very important to competitive programmers. Typically, a solution to a problem is a combination of well-known techniques and new insights. The techniques that appear in competitive programming also form the basis for the scientific research of algorithms.

The **implementation of algorithms** requires good programming skills. In competitive programming, the solutions are graded by testing an implemented algorithm using a set of test cases. Thus, it is not enough that the idea of the algorithm is correct, but the implementation has also to be correct.

Good coding style in contests is straightforward and concise. The programs should be written quickly, because there is not much time available. Unlike in traditional software engineering, the programs are short (usually at most some hundreds of lines) and it is not needed to maintain them after the contest.

1.1 Programming languages

At the moment, the most popular programming languages in contests are C++, Python and Java. For example, in Google Code Jam 2016, among the best 3,000 participants, 73 % used C++, 15 % used Python and 10 % used Java [20]. Some participants also used several languages.

Many people think that C++ is the best choice for a competitive programmer, and C++ is nearly always available in contest systems. The benefits in using C++ are that it is a very efficient language and its standard library contains a large collection of data structures and algorithms.

On the other hand, it is good to master several languages and know their strengths. For example, if large integers are needed in the problem, Python can be a good choice, because it contains built-in operations for calculating with large

integers. Still, most problems in programming contests are set so that using a specific programming language is not an unfair advantage.

All example programs in this book are written in C++, and the standard library's data structures and algorithms are often used. The programs follow the C++11 standard, that can be used in most contests nowadays. If you cannot program in C++ yet, now it is a good time to start learning.

C++ template

A typical C++ template for competitive programming looks like this:

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

using namespace std;

int main() {
    // solution comes here
}
```

The `#include` line at the beginning of the code is a feature of the g++ compiler that allows us to include the entire standard library. Thus, it is not needed to separately include libraries such as `iostream`, `vector` and `algorithm`, but they are available automatically.

The `using` line determines that the classes and functions of the standard library can be used directly in the code. Without the `using` line we should write, for example, `std::cout`, but now it suffices to write `cout`.

The code can be compiled using the following command:

```
g++ -std=c++11 -O2 -Wall code.cpp -o code
```

This command produces a binary file `code` from the source code `code.cpp`. The compiler follows the C++11 standard (`-std=c++11`), optimizes the code (`-O2`) and shows warnings about possible errors (`-Wall`).

1.2 Input and output

In most contests, standard streams are used for reading input and writing output. In C++, the standard streams are `cin` for input and `cout` for output. In addition, the C functions `scanf` and `printf` can be used.

The input for the program usually consists of numbers and strings that are separated with spaces and newlines. They can be read from the `cin` stream as follows:

```cpp
int a, b;
string x;
cin >> a >> b >> x;
```

This kind of code always works, assuming that there is at least one space or newline between each element in the input. For example, the above code can read both the following inputs:

```
123 456 monkey
```

```
123    456
monkey
```

The `cout` stream is used for output as follows:

```
int a = 123, b = 456;
string x = "monkey";
cout << a << " " << b << " " << x << "\n";
```

Input and output is sometimes a bottleneck in the program. The following lines at the beginning of the code make input and output more efficient:

```
ios_base::sync_with_stdio(0);
cin.tie(0);
```

Note that the newline "\n" works faster than `endl`, because `endl` always causes a flush operation.

The C functions `scanf` and `printf` are an alternative to the C++ standard streams. They are usually a bit faster, but they are also more difficult to use. The following code reads two integers from the input:

```
int a, b;
scanf("%d %d", &a, &b);
```

The following code prints two integers:

```
int a = 123, b = 456;
printf("%d %d\n", a, b);
```

Sometimes the program should read a whole line from the input, possibly with spaces. This can be accomplished by using the `getline` function:

```
string s;
getline(cin, s);
```

If the amount of data is unknown, the following loop is useful:

```
while (cin >> x) {
    // code
}
```

This loop reads elements from the input one after another, until there is no more data available in the input.

In some contest systems, files are used for input and output. An easy solution for this is to write the code as usual using standard streams, but add the following lines to the beginning of the code:

```
freopen("input.txt", "r", stdin);
freopen("output.txt", "w", stdout);
```

After this, the program reads the input from the file "input.txt" and writes the output to the file "output.txt".

1.3   Working with numbers

Integers

The most used integer type in competitive programming is `int`, that is a 32-bit type with value range $-2^{31} \ldots 2^{31} - 1$ or about $-2 \cdot 10^9 \ldots 2 \cdot 10^9$. If the type `int` is not enough, the 64-bit type `long long` can be used, with value range $-2^{63} \ldots 2^{63} - 1$ or about $-9 \cdot 10^{18} \ldots 9 \cdot 10^{18}$.

The following code defines a `long long` variable:

```
long long x = 123456789123456789LL;
```

The suffix `LL` means that the type of the number is `long long`.

A common error when using the type `long long` is that the type `int` is still used somewhere in the code. For example, the following code contains a subtle error:

```
int a = 123456789;
long long b = a*a;
cout << b << "\n"; // -1757895751
```

Even though the variable `b` is of type `long long`, both numbers in the expression `a*a` are of type `int` and the result is also of type `int`. Because of this, the variable `b` will contain a wrong result. The problem can be solved by changing the type of `a` to `long long` or by changing the expression to `(long long)a*a`.

Usually contest problems are set so that the type `long long` is enough. Still, it is good to know that the g++ compiler also provides an 128-bit type `__int128_t` with value range $-2^{127} \ldots 2^{127} - 1$ or $-10^{38} \ldots 10^{38}$. However, this type is not available in all contest systems.

Modular arithmetic

We denote by $x \bmod m$ the remainder when x is divided by m. For example, $17 \bmod 5 = 2$, because $17 = 3 \cdot 5 + 2$.

Sometimes, the answer to a problem is a very large number but it is enough to output it "modulo m", i.e., the remainder when the answer is divided by m (for

example, "modulo $10^9 + 7$"). The idea is that even if the actual answer may be very large, it suffices to use the types int and long long.

An important property of the remainder is that in addition, subtraction and multiplication, the remainder can be taken before the operation:

$$
\begin{aligned}
(a + b) \bmod m &= (a \bmod m + b \bmod m) \bmod m \\
(a - b) \bmod m &= (a \bmod m - b \bmod m) \bmod m \\
(a \cdot b) \bmod m &= (a \bmod m \cdot b \bmod m) \bmod m
\end{aligned}
$$

Thus, we can take the remainder after every operation and the numbers will never become too large.

For example, the following code calculates $n!$, the factorial of n, modulo m:

```
long long x = 1;
for (int i = 2; i <= n i++) {
    x = (x*i)%m;
}
cout << x << "\n";
```

Usually the remainder should be always be between $0 \ldots m - 1$. However, in C++ and other languages, the remainder of a negative number is either zero or negative. An easy way to make sure there are no negative remainders is to first calculate the remainder as usual and then add m if the result is negative:

```
x = x%m;
if (x < 0) x += m;
```

However, this is only needed when there are subtractions in the code and the remainder may become negative.

Floating point numbers

The usual floating point types in competitive programming are the 64-bit double and, as an extension in the g++ compiler, the 80-bit long double. In most cases, double is enough, but long double is more accurate.

The required precision of the answer is usually given in the problem statement. An easy way to output the answer is to use the printf function and give the number of decimal places in the formatting string. For example, the following code prints the value of x with 9 decimal places:

```
printf("%.9f\n", x);
```

A difficulty when using floating point numbers is that some numbers cannot be represented accurately as floating point numbers, but there will be rounding errors. For example, the result of the following code is surprising:

```
double x = 0.3*3+0.1;
printf("%.20f\n", x); // 0.99999999999999988898
```

Due to a rounding error, the value of x is a bit smaller than 1, while the correct value would be 1.

It is risky to compare floating point numbers with the == operator, because it is possible that the values should be equal but they are not because of rounding. A better way to compare floating point numbers is to assume that two numbers are equal if the difference between them is ε, where ε is a small number.

In practice, the numbers can be compared as follows ($\varepsilon = 10^{-9}$):

```
if (abs(a-b) < 1e-9) {
    // a and b are equal
}
```

Note that while floating point numbers are inaccurate, integers up to a certain limit can be still represented accurately. For example, using double, it is possible to accurately represent all integers whose absolute value is at most 2^{53}.

1.4 Shortening code

Short code is ideal in competitive programming, because programs should be written as fast as possible. Because of this, competitive programmers often define shorter names for datatypes and other parts of code.

Type names

Using the command typedef it is possible to give a shorter name to a datatype. For example, the name long long is long, so we can define a shorter name ll:

```
typedef long long ll;
```

After this, the code

```
long long a = 123456789;
long long b = 987654321;
cout << a*b << "\n";
```

can be shortened as follows:

```
ll a = 123456789;
ll b = 987654321;
cout << a*b << "\n";
```

The command typedef can also be used with more complex types. For example, the following code gives the name vi for a vector of integers and the name pi for a pair that contains two integers.

```
typedef vector<int> vi;
typedef pair<int,int> pi;
```

Macros

Another way to shorten the code is to define **macros**. A macro means that certain strings in the code will be changed before the compilation. In C++, macros are defined using the command #define.

For example, we can define the following macros:

```
#define F first
#define S second
#define PB push_back
#define MP make_pair
```

After this, the code

```
v.push_back(make_pair(y1,x1));
v.push_back(make_pair(y2,x2));
int d = v[i].first+v[i].second;
```

can be shortened as follows:

```
v.PB(MP(y1,x1));
v.PB(MP(y2,x2));
int d = v[i].F+v[i].S;
```

A macro can also have parameters which makes it possible to shorten loops and other structures. For example, we can define the following macro:

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

After this, the code

```
for (int i = 1; i <= n; i++) {
    search(i);
}
```

can be shortened as follows:

```
REP(i,1,n) {
    search(i);
}
```

1.5 Mathematics

Mathematics plays an important role in competitive programming, and it is not possible to become a successful competitive programmer without having good mathematical skills. This section discusses some important mathematical concepts and formulas that are needed later in the book.

Sum formulas

Each sum of the form

$$\sum_{x=1}^{n} x^k = 1^k + 2^k + 3^k + \ldots + n^k,$$

where k is a positive integer, has a closed-form formula that is a polynomial of degree $k + 1$. For example,

$$\sum_{x=1}^{n} x = 1 + 2 + 3 + \ldots + n = \frac{n(n+1)}{2}$$

and

$$\sum_{x=1}^{n} x^2 = 1^2 + 2^2 + 3^2 + \ldots + n^2 = \frac{n(n+1)(2n+1)}{6}.$$

An **arithmetic progression** is a sequence of numbers where the difference between any two consecutive numbers is constant. For example,

$$3, 7, 11, 15$$

is an arithmetic progression with constant 4. The sum of an arithmetic progression can be calculated using the formula

$$\frac{n(a+b)}{2}$$

where a is the first number, b is the last number and n is the amount of numbers. For example,

$$3 + 7 + 11 + 15 = \frac{4 \cdot (3 + 15)}{2} = 36.$$

The formula is based on the fact that the sum consists of n numbers and the value of each number is $(a + b)/2$ on average.

A **geometric progression** is a sequence of numbers where the ratio between any two consecutive numbers is constant. For example,

$$3, 6, 12, 24$$

is a geometric progression with constant 2. The sum of a geometric progression can be calculated using the formula

$$\frac{bx - a}{x - 1}$$

where a is the first number, b is the last number and the ratio between consecutive numbers is x. For example,

$$3 + 6 + 12 + 24 = \frac{24 \cdot 2 - 3}{2 - 1} = 45.$$

This formula can be derived as follows. Let

$$S = a + ax + ax^2 + \cdots + b.$$

By multiplying both sides by x, we get

$$xS = ax + ax^2 + ax^3 + \cdots + bx,$$

and solving the equation

$$xS - S = bx - a$$

yields the formula.

A special case of a sum of a geometric progression is the formula

$$1 + 2 + 4 + 8 + \ldots + 2^{n-1} = 2^n - 1.$$

A **harmonic sum** is a sum of the form

$$\sum_{x=1}^{n} \frac{1}{x} = 1 + \frac{1}{2} + \frac{1}{3} + \ldots + \frac{1}{n}.$$

An upper bound for a harmonic sum is $\log_2(n) + 1$. Namely, we can modify each term $1/k$ so that k becomes the nearest power of two that does not exceed k. For example, when $n = 6$, we can estimate the sum as follows:

$$1 + \frac{1}{2} + \frac{1}{3} + \frac{1}{4} + \frac{1}{5} + \frac{1}{6} \le 1 + \frac{1}{2} + \frac{1}{2} + \frac{1}{4} + \frac{1}{4} + \frac{1}{4}.$$

This upper bound consists of $\log_2(n) + 1$ parts (1, $2 \cdot 1/2$, $4 \cdot 1/4$, etc.), and the value of each part is at most 1.

Set theory

A **set** is a collection of elements. For example, the set

$$X = \{2, 4, 7\}$$

contains elements 2, 4 and 7. The symbol $\emptyset$ denotes an empty set, and $|S|$ denotes the size of a set S, i.e., the number of elements in the set. For example, in the above set, $|X| = 3$.

If a set S contains an element x, we write $x \in S$, and otherwise we write $x \notin S$. For example, in the above set

$$4 \in X \quad \text{and} \quad 5 \notin X.$$

New sets can be constructed using set operations:

- The **intersection** $A \cap B$ consists of elements that are both in A and B. For example, if $A = \{1, 2, 5\}$ and $B = \{2, 4\}$, then $A \cap B = \{2\}$.

- The **union** $A \cup B$ consists of elements that are in A or B or both. For example, if $A = \{3, 7\}$ and $B = \{2, 3, 8\}$, then $A \cup B = \{2, 3, 7, 8\}$.

- The **complement** $\bar{A}$ consists of elements that are not in A. The interpretation of a complement depends on the **universal set** that contains all possible elements. For example, if $A = \{1, 2, 5, 7\}$ and the universal set is $\{1, 2, \ldots, 10\}$, then $\bar{A} = \{3, 4, 6, 8, 9, 10\}$.

- The **difference** $A \setminus B = A \cap \bar{B}$ consists of elements that are in A but not in B. Note that B can contain elements that are not in A. For example, if $A = \{2, 3, 7, 8\}$ and $B = \{3, 5, 8\}$, then $A \setminus B = \{2, 7\}$.

If each element of A also belongs to S, we say that A is a **subset** of S, denoted by $A \subset S$. A set S always has $2^{|S|}$ subsets, including the empty set. For example, the subsets of the set $\{2, 4, 7\}$ are

$$\emptyset, \{2\}, \{4\}, \{7\}, \{2, 4\}, \{2, 7\}, \{4, 7\} \text{ and } \{2, 4, 7\}.$$

Often used sets are $\mathbb{N}$ (natural numbers), $\mathbb{Z}$ (integers), $\mathbb{Q}$ (rational numbers) and $\mathbb{R}$ (real numbers). The set $\mathbb{N}$ can be defined in two ways, depending on the situation: either $\mathbb{N} = \{0, 1, 2, \ldots\}$ or $\mathbb{N} = \{1, 2, 3, \ldots\}$.

We can also construct a set using a rule of the form

$$\{f(n) : n \in S\},$$

where $f(n)$ is some function. This set contains all elements of the form $f(n)$, where n is an element in S. For example, the set

$$X = \{2n : n \in \mathbb{Z}\}$$

contains all even integers.

Logic

The value of a logical expression is either **true** (1) or **false** (0). The most important logical operators are $\neg$ (**negation**), $\wedge$ (**conjunction**), $\vee$ (**disjunction**), $\Rightarrow$ (**implication**) and $\Leftrightarrow$ (**equivalence**). The following table shows the meaning of these operators:

A	B	$\neg A$	$\neg B$	$A \wedge B$	$A \vee B$	$A \Rightarrow B$	$A \Leftrightarrow B$
0	0	1	1	0	0	1	1
0	1	1	0	0	1	1	0
1	0	0	1	0	1	0	0
1	1	0	0	1	1	1	1

The expression $\neg A$ has the opposite value of A. The expression $A \wedge B$ is true if both A and B are true, and the expression $A \vee B$ is true if A or B or both are true. The expression $A \Rightarrow B$ is true if whenever A is true, also B is true. The expression $A \Leftrightarrow B$ is true if A and B are both true or both false.

A **predicate** is an expression that is true or false depending on its parameters. Predicates are usually denoted by capital letters. For example, we can define a predicate $P(x)$ that is true exactly when x is a prime number. Using this definition, $P(7)$ is true but $P(8)$ is false.

A **quantifier** connects a logical expression to the elements of a set. The most important quantifiers are $\forall$ (**for all**) and $\exists$ (**there is**). For example,

$$\forall x (\exists y (y < x))$$

means that for each element x in the set, there is an element y in the set such that y is smaller than x. This is true in the set of integers, but false in the set of natural numbers.

Using the notation described above, we can express many kinds of logical propositions. For example,

$$\forall x ((x > 1 \wedge \neg P(x)) \Rightarrow (\exists a (\exists b (x = ab \wedge a > 1 \wedge b > 1))))$$

means that if a number x is larger than 1 and not a prime number, then there are numbers a and b that are larger than 1 and whose product is x. This proposition is true in the set of integers.

Functions

The function $\lfloor x \rfloor$ rounds the number x down to an integer, and the function $\lceil x \rceil$ rounds the number x up to an integer. For example,

$$\lfloor 3/2 \rfloor = 1 \quad \text{and} \quad \lceil 3/2 \rceil = 2.$$

The functions $\min(x_1, x_2, \ldots, x_n)$ and $\max(x_1, x_2, \ldots, x_n)$ give the smallest and largest of values $x_1, x_2, \ldots, x_n$. For example,

$$\min(1, 2, 3) = 1 \quad \text{and} \quad \max(1, 2, 3) = 3.$$

The **factorial** $n!$ can be defined

$$\prod_{x=1}^{n} x = 1 \cdot 2 \cdot 3 \cdot \ldots \cdot n$$

or recursively

$$\begin{aligned} 0! &= 1 \\ n! &= n \cdot (n-1)! \end{aligned}$$

The **Fibonacci numbers** arise in many situations. They can be defined recursively as follows:

$$\begin{aligned} f(0) &= 0 \\ f(1) &= 1 \\ f(n) &= f(n-1) + f(n-2) \end{aligned}$$

The first Fibonacci numbers are

$$0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, \ldots$$

There is also a closed-form formula for calculating Fibonacci numbers:

$$f(n) = \frac{(1+\sqrt{5})^n - (1-\sqrt{5})^n}{2^n \sqrt{5}}.$$

Logarithms

The **logarithm** of a number x is denoted $\log_k(x)$, where k is the base of the logarithm. According to the definition, $\log_k(x) = a$ exactly when $k^a = x$.

A useful property of logarithms is that $\log_k(x)$ equals the number of times we have to divide x by k before we reach the number 1. For example, $\log_2(32) = 5$ because 5 divisions are needed:

$$32 \to 16 \to 8 \to 4 \to 2 \to 1$$

Logarithms are often used in the analysis of algorithms, because many efficient algorithms halve something at each step. Hence, we can estimate the efficiency of such algorithms using logarithms.

The logarithm of a product is

$$\log_k(ab) = \log_k(a) + \log_k(b),$$

and consequently,

$$\log_k(x^n) = n \cdot \log_k(x).$$

In addition, the logarithm of a quotient is

$$\log_k \left(\frac{a}{b} \right) = \log_k(a) - \log_k(b).$$

Another useful formula is

$$\log_u(x) = \frac{\log_k(x)}{\log_k(u)},$$

and using this, it is possible to calculate logarithms to any base if there is a way to calculate logarithms to some fixed base.

The **natural logarithm** $\ln(x)$ of a number x is a logarithm whose base is $e \approx 2{,}71828$.

Another property of logarithms is that the number of digits of an integer x in base b is $\lfloor \log_b(x) + 1 \rfloor$. For example, the representation of 123 in base 2 is 1111011 and $\lfloor \log_2(123) + 1 \rfloor = 7$.

1.6 Contests

IOI

The International Olympiad in Informatics (IOI) [25] is an annual programming contests for secondary school students. Each country is allowed to send a team of

four students to the contest. There are usually about 300 participants from 80 countries [26].

The IOI consists of two five-hour long contests. In both contests, the participants are asked to solve three algorithm tasks of various difficulty. The tasks are divided into subtasks, each of which has an assigned score. Even if the contestants are divided into teams, they compete as individuals.

The IOI syllabus [27] regulates the topics that may appear in IOI tasks. This book covers almost all topics in the IOI syllabus.

The participants for the IOI are selected through national contests. Before the IOI, many regional contests are organized, such as the Baltic Olympiad in Informatics (BOI), the Central European Olympiad in Informatics (CEOI) and the Asia-Pacific Informatics Olympiad (APIO).

Some countries organize online practice contests for future IOI participants, such as the Croatian Open Competition in Informatics (COCI) and the USA Computing Olympiad (USACO). In addition, many problems from Polish contests are available online [33].

ICPC

The International Collegiate Programming Contest (ICPC) [24] is an annual programming contest for university students. Each team in the contest consists of three students, and unlike in the IOI, the students work together; there is even only one computer available for each team.

The ICPC consists of several stages, and finally the best teams are invited to the World Finals. While there are tens of thousands of participants in the contest, there are only 128 final slots available, so even advancing to the finals is a great achievement in some regions.

In each ICPC contest, the teams have five hours time to solve about ten algorithm problems. A solution to a problem is accepted only if it solves all test cases efficiently. During the contest, the teams see the results of the other teams, but for the last hour the scoreboard is frozen and it is not possible to see the results of the last submissions.

The topics that may appear at the ICPC are not so well specified as those at the IOI. In any case, it is clear that more knowledge is needed at the ICPC, especially more mathematical skills.

Online contests

There are also many online contests that are open for everybody. At the moment, the most active contest site is Codeforces that organizes contests about weekly. In Codeforces, participants are divided into two divisions: beginners compete in Div2 and more experienced programmers in Div1. Other contest sites include AtCoder, CS Academy, HackerRank and Topcoder.

Some companies organize online contests with onsite finals. Examples of such contests are Facebook Hacker Cup, Google Code Jam and Yandex.Algorithm. Of

course, companies also use those contests for recruiting: performing well in a contest is a good way to prove one's skills.

1.7 Books

Competitive programming books

There are already some books (besides this book) that concentrate on competitive programming and algorithmic problem solving:

- S. Halim and F. Halim: *Competitive Programming 3: The New Lower Bound of Programming Contests*, 2013

- S. S. Skiena and M. A. Revilla: *Programming Challenges: The Programming Contest Training Manual*, Springer, 2003

- *Looking for a Challenge? The Ultimate Problem Set from the University of Warsaw Programming Competitions*, 2012

The first two books are intended for beginners, whereas the last book contains advanced material.

General algorithm books

Of course, general algorithm books are also suitable for competitive programmers. Some good books are:

- T. H. Cormen, C. E. Leiserson, R. L. Rivest and C. Stein: *Introduction to Algorithms*, MIT Press, 2009 (3rd edition)

- J. Kleinberg and É. Tardos: *Algorithm Design*, Pearson, 2005

- S. S. Skiena: *The Algorithm Design Manual*, Springer, 2008 (2nd edition)

Chapter 2

Time complexity

The efficiency of algorithms is important in competitive programming. Usually, it is easy to design an algorithm that solves the problem slowly, but the real challenge is to invent a fast algorithm. If the algorithm is too slow, it will get only partial points or no points at all.

The **time complexity** of an algorithm estimates how much time the algorithm will use for some input. The idea is to represent the efficiency as an function whose parameter is the size of the input. By calculating the time complexity, we can find out whether the algorithm is good enough without implementing it.

2.1 Calculation rules

The time complexity of an algorithm is denoted $O(\cdots)$ where the three dots represent some function. Usually, the variable n denotes the input size. For example, if the input is an array of numbers, n will be the size of the array, and if the input is a string, n will be the length of the string.

Loops

A common reason why an algorithm is slow is that it contains many loops that go through the input. The more nested loops the algorithm contains, the slower it is. If there are k nested loops, the time complexity is $O(n^k)$.

For example, the time complexity of the following code is $O(n)$:

```
for (int i = 1; i <= n; i++) {
    // code
}
```

And the time complexity of the following code is $O(n^2)$:

```
for (int i = 1; i <= n; i++) {
    for (int j = 1; j <= n; j++) {
        // code
    }
}
```

Order of magnitude

A time complexity does not tell us the exact number of times the code inside a loop is executed, but it only shows the order of magnitude. In the following examples, the code inside the loop is executed $3n$, $n+5$ and $\lceil n/2 \rceil$ times, but the time complexity of each code is $O(n)$.

```
for (int i = 1; i <= 3*n; i++) {
    // code
}
```

```
for (int i = 1; i <= n+5; i++) {
    // code
}
```

```
for (int i = 1; i <= n; i += 2) {
    // code
}
```

As another example, the time complexity of the following code is $O(n^2)$:

```
for (int i = 1; i <= n; i++) {
    for (int j = i+1; j <= n; j++) {
        // code
    }
}
```

Phases

If the code consists of consecutive phases, the total time complexity is the largest time complexity of a single phase. The reason for this is that the slowest phase is usually the bottleneck of the code.

For example, the following code consists of three phases with time complexities $O(n)$, $O(n^2)$ and $O(n)$. Thus, the total time complexity is $O(n^2)$.

```
for (int i = 1; i <= n; i++) {
    // code
}
for (int i = 1; i <= n; i++) {
    for (int j = 1; j <= n; j++) {
        // code
    }
}
for (int i = 1; i <= n; i++) {
    // code
}
```

Several variables

Sometimes the time complexity depends on several factors. In this case, the time complexity formula contains several variables.

For example, the time complexity of the following code is $O(nm)$:

```
for (int i = 1; i <= n; i++) {
    for (int j = 1; j <= m; j++) {
        // code
    }
}
```

Recursion

The time complexity of a recursive function depends on the number of times the function is called and the time complexity of a single call. The total time complexity is the product of these values.

For example, consider the following function:

```
void f(int n) {
    if (n == 1) return;
    f(n-1);
}
```

The call $f(n)$ causes n function calls, and the time complexity of each call is $O(1)$. Thus, the total time complexity is $O(n)$.

As another example, consider the following function:

```
void g(int n) {
    if (n == 1) return;
    g(n-1);
    g(n-1);
}
```

In this case each function call generates two other calls, except for $n = 1$. Hence, the call $g(n)$ causes the following calls:

parameter	number of calls
g(n)	1
g($n-1$)	2
...	...
g(1)	2^{n-1}

Based on this, the time complexity is

$$1 + 2 + 4 + \cdots + 2^{n-1} = 2^n - 1 = O(2^n).$$

2.2 Complexity classes

The following list contains common time complexities of algorithms:

$O(1)$ The running time of a **constant-time** algorithm does not depend on the input size. A typical constant-time algorithm is a direct formula that calculates the answer.

$O(\log n)$ A **logarithmic** algorithm often halves the input size at each step. The running time of such an algorithm is logarithmic, because $\log_2 n$ equals the number of times n must be divided by 2 to get 1.

$O(\sqrt{n})$ A **square root algorithm** is slower than $O(\log n)$ but faster than $O(n)$. A special property of square roots is that $\sqrt{n} = n/\sqrt{n}$, so the square root $\sqrt{n}$ lies in some sense in the middle of the input.

$O(n)$ A **linear** algorithm goes through the input a constant number of times. This is often the best possible time complexity, because it is usually needed to access each input element at least once before reporting the answer.

$O(n \log n)$ This time complexity often indicates that the algorithm sorts the input, because the time complexity of efficient sorting algorithms is $O(n \log n)$. Another possibility is that the algorithm uses a data structure where each operation takes $O(\log n)$ time.

$O(n^2)$ A **quadratic** algorithm often contains two nested loops. It is possible to go through all pairs of the input elements in $O(n^2)$ time.

$O(n^3)$ A **cubic** algorithm often contains three nested loops. It is possible to go through all triplets of the input elements in $O(n^3)$ time.

$O(2^n)$ This time complexity often indicates that the algorithm iterates through all subsets of the input elements. For example, the subsets of $\{1,2,3\}$ are $\emptyset$, $\{1\}$, $\{2\}$, $\{3\}$, $\{1,2\}$, $\{1,3\}$, $\{2,3\}$ and $\{1,2,3\}$.

$O(n!)$ This time complexity often indicates that the algorithm iterates through all permutations of the input elements. For example, the permutations of $\{1,2,3\}$ are $(1,2,3)$, $(1,3,2)$, $(2,1,3)$, $(2,3,1)$, $(3,1,2)$ and $(3,2,1)$.

An algorithm is **polynomial** if its time complexity is at most $O(n^k)$ where k is a constant. All the above time complexities except $O(2^n)$ and $O(n!)$ are polynomial. In practice, the constant k is usually small, and therefore a polynomial time complexity roughly means that the algorithm is *efficient*.

Most algorithms in this book are polynomial. Still, there are many important problems for which no polynomial algorithm is known, i.e., nobody knows how to solve them efficiently. **NP-hard** problems are an important set of problems for which no polynomial algorithm is known [19].

2.3 Estimating efficiency

By calculating the time complexity of an algorithm, it is possible to check before implementing the algorithm that it is efficient enough for the problem. The starting point for estimations is the fact that a modern computer can perform some hundreds of millions of operations in a second.

For example, assume that the time limit for a problem is one second and the input size is $n = 10^5$. If the time complexity is $O(n^2)$, the algorithm will perform about $(10^5)^2 = 10^{10}$ operations. This should take at least some tens of seconds, so the algorithm seems to be too slow for solving the problem.

On the other hand, given the input size, we can try to guess the required time complexity of the algorithm that solves the problem. The following table contains some useful estimates assuming that the time limit is one second.

input size (n)	required time complexity
$n \le 10^{18}$	$O(1)$ or $O(\log n)$
$n \le 10^{12}$	$O(\sqrt{n})$
$n \le 10^6$	$O(n)$ or $O(n \log n)$
$n \le 5000$	$O(n^2)$
$n \le 500$	$O(n^3)$
$n \le 25$	$O(2^n)$
$n \le 10$	$O(n!)$

For example, if the input size is $n = 10^5$, it is probably expected that the time complexity of the algorithm is $O(n)$ or $O(n \log n)$. This information makes it easier to design the algorithm, because it rules out approaches that would yield an algorithm with a worse time complexity.

Still, it is important to remember that a time complexity is only an estimate of efficiency, because it hides the **constant factors**. For example, an algorithm that runs in $O(n)$ time may perform $n/2$ or $5n$ operations. This has an important effect on the actual running time of the algorithm.

2.4 Maximum subarray sum

There are often several possible algorithms for solving a problem such that their time complexities are different. This section discusses a classic problem that has a straightforward $O(n^3)$ solution. However, by designing a better algorithm it is possible to solve the problem in $O(n^2)$ time and even in $O(n)$ time.

Given an array of n integers $x_1, x_2, \ldots, x_n$, our task is to find the **maximum subarray sum**[1], i.e., the largest possible sum of numbers in a contiguous region in the array. The problem is interesting when there may be negative numbers in the array. For example, in the array

1	2	3	4	5	6	7	8
-1	2	4	-3	5	2	-5	2

[1] Bentley's book *Programming Pearls* [7] made this problem popular.

the following subarray produces the maximum sum 10:

1	2	3	4	5	6	7	8
−1	2	4	−3	5	2	−5	2

Algorithm 1

A straightforward algorithm to the problem is to go through all possible ways to select a subarray, calculate the sum of numbers in each subarray and maintain the maximum sum. The following code implements this algorithm:

```
int p = 0;
for (int a = 1; a <= n; a++) {
    for (int b = a; b <= n; b++) {
        int s = 0;
        for (int c = a; c <= b; c++) {
            s += x[c];
        }
        p = max(p,s);
    }
}
cout << p << "\n";
```

The code assumes that the numbers are stored in an array x with indices $1 \ldots n$. The variables a and b determine the first and last number in the subarray, and the sum of the numbers is calculated to the variable s. The variable p contains the maximum sum found during the search.

The time complexity of the algorithm is $O(n^3)$, because it consists of three nested loops and each loop contains $O(n)$ steps.

Algorithm 2

It is easy to make the first algorithm more efficient by removing one loop from it. This is possible by calculating the sum at the same time when the right end of the subarray moves. The result is the following code:

```
int p = 0;
for (int a = 1; a <= n; a++) {
    int s = 0;
    for (int b = a; b <= n; b++) {
        s += x[b];
        p = max(p,s);
    }
}
cout << p << "\n";
```

After this change, the time complexity is $O(n^2)$.

Algorithm 3

Surprisingly, it is possible to solve the problem in $O(n)$ time, which means that we can remove one more loop. The idea is to calculate for each array position the maximum sum of a subarray that ends at that position. After this, the answer for the problem is the maximum of those sums.

Condider the subproblem of finding the maximum-sum subarray that ends at position k. There are two possibilities:

1. The subarray only contains the element at position k.

2. The subarray consists of a subarray that ends at position $k-1$, followed by the element at position k.

Our goal is to find a subarray with maximum sum, so in case 2 the subarray that ends at position $k-1$ should also have the maximum sum. Thus, we can solve the problem efficiently when we calculate the maximum subarray sum for each ending position from left to right.

The following code implements the algorithm:

```
int p = 0, s = 0;
for (int k = 1; k <= n; k++) {
    s = max(x[k],s+x[k]);
    p = max(p,s);
}
cout << p << "\n";
```

The algorithm only contains one loop that goes through the input, so the time complexity is $O(n)$. This is also the best possible time complexity, because any algorithm for the problem has to examine all array elements at least once.

Efficiency comparison

It is interesting to study how efficient algorithms are in practice. The following table shows the running times of the above algorithms for different values of n in a modern computer.

In each test, the input was generated randomly. The time needed for reading the input was not measured.

array size n	algorithm 1	algorithm 2	algorithm 3
10^2	0,0 s	0,0 s	0,0 s
10^3	0,1 s	0,0 s	0,0 s
10^4	> 10,0 s	0,1 s	0,0 s
10^5	> 10,0 s	5,3 s	0,0 s
10^6	> 10,0 s	> 10,0 s	0,0 s
10^7	> 10,0 s	> 10,0 s	0,0 s

The comparison shows that all algorithms are efficient when the input size is small, but larger inputs bring out remarkable differences in running times of

the algorithms. The $O(n^3)$ time algorithm 1 becomes slow when $n = 10^4$, and the $O(n^2)$ time algorithm 2 becomes slow when $n = 10^5$. Only the $O(n)$ time algorithm 3 processes even the largest inputs instantly.

Chapter 3

Sorting

Sorting is a fundamental algorithm design problem. Many efficient algorithms use sorting as a subroutine, because it is often easier to process data if the elements are in a sorted order.

For example, the problem "does the array contain two equal elements?" is easy to solve using sorting. If the array contains two equal elements, they will be next to each other after sorting, so it is easy to find them. Also the problem "what is the most frequent element in the array?" can be solved similarly.

There are many algorithms for sorting, and they are also good examples of how to apply different algorithm design techniques. The efficient general sorting algorithms work in $O(n \log n)$ time, and many algorithms that use sorting as a subroutine also have this time complexity.

3.1 Sorting theory

The basic problem in sorting is as follows:

> Given an array that contains n elements, your task is to sort the elements in increasing order.

For example, the array

1	2	3	4	5	6	7	8
1	3	8	2	9	2	5	6

will be as follows after sorting:

1	2	3	4	5	6	7	8
1	2	2	3	5	6	8	9

$O(n^2)$ algorithms

Simple algorithms for sorting an array work in $O(n^2)$ time. Such algorithms are short and usually consist of two nested loops. A famous $O(n^2)$ time sorting

algorithm is **bubble sort** where the elements "bubble" in the array according to their values.

Bubble sort consists of $n-1$ rounds. On each round, the algorithm iterates through the elements of the array. Whenever two consecutive elements are found that are not in correct order, the algorithm swaps them. The algorithm can be implemented as follows for an array $t[1], t[2], \ldots, t[n]$:

```
for (int i = 1; i <= n-1; i++) {
    for (int j = 1; j <= n-i; j++) {
        if (t[j] > t[j+1]) swap(t[j],t[j+1]);
    }
}
```

After the first round of the algorithm, the largest element will be in the correct position, and in general, after k rounds, the k largest elements will be in the correct positions. Thus, after $n-1$ rounds, the whole array will be sorted.

For example, in the array

1	2	3	4	5	6	7	8
1	3	8	2	9	2	5	6

the first round of bubble sort swaps elements as follows:

1	2	3	4	5	6	7	8
1	3	2	8	9	2	5	6

1	2	3	4	5	6	7	8
1	3	2	8	2	9	5	6

1	2	3	4	5	6	7	8
1	3	2	8	2	5	9	6

1	2	3	4	5	6	7	8
1	3	2	8	2	5	6	9

Inversions

Bubble sort is an example of a sorting algorithm that always swaps consecutive elements in the array. It turns out that the time complexity of such an algorithm

is *always* at least $O(n^2)$, because in the worst case, $O(n^2)$ swaps are required for sorting the array.

A useful concept when analyzing sorting algorithms is an **inversion**: a pair of elements $(t[a], t[b])$ in the array such that $a < b$ and $t[a] > t[b]$, i.e., the elements are in the wrong order. For example, in the array

1	2	3	4	5	6	7	8
1	2	2	6	3	5	9	8

the inversions are $(6,3)$, $(6,5)$ and $(9,8)$. The number of inversions tells us how much work is needed to sort the array. An array is completely sorted when there are no inversions. On the other hand, if the array elements are in the reverse order, the number of inversions is the largest possible:

$$1 + 2 + \cdots + (n-1) = \frac{n(n-1)}{2} = O(n^2)$$

Swapping a pair of consecutive elements that are in the wrong order removes exactly one inversion from the array. Hence, if a sorting algorithm can only swap consecutive elements, each swap removes at most one inversion and the time complexity of the algorithm is at least $O(n^2)$.

$O(n \log n)$ **algorithms**

It is possible to sort an array efficiently in $O(n \log n)$ time using algorithms that are not limited to swapping consecutive elements. One such algorithm is **mergesort** that is based on recursion.

Mergesort sorts a subarray $t[a, b]$ as follows:

1. If $a = b$, do not do anything, because the subarray is already sorted.

2. Calculate the position of the middle element: $k = \lfloor (a+b)/2 \rfloor$.

3. Recursively sort the subarray $t[a, k]$.

4. Recursively sort the subarray $t[k+1, b]$.

5. *Merge* the sorted subarrays $t[a, k]$ and $t[k+1, b]$ into a sorted subarray $t[a, b]$.

Mergesort is an efficient algorithm, because it halves the size of the subarray at each step. The recursion consists of $O(\log n)$ levels, and processing each level takes $O(n)$ time. Merging the subarrays $t[a, k]$ and $t[k+1, b]$ is possible in linear time, because they are already sorted.

For example, consider sorting the following array:

1	2	3	4	5	6	7	8
1	3	6	2	8	2	5	9

The array will be divided into two subarrays as follows:

1	2	3	4
1	3	6	2

5	6	7	8
8	2	5	9

Then, the subarrays will be sorted recursively as follows:

1	2	3	4
1	2	3	6

5	6	7	8
2	5	8	9

Finally, the algorithm merges the sorted subarrays and creates the final sorted array:

1	2	3	4	5	6	7	8
1	2	2	3	5	6	8	9

Sorting lower bound

Is it possible to sort an array faster than in $O(n \log n)$ time? It turns out that this is *not* possible when we restrict ourselves to sorting algorithms that are based on comparing array elements.

The lower bound for the time complexity can be proved by considering sorting as a process where each comparison of two elements gives more information about the contents of the array. The process creates the following tree:
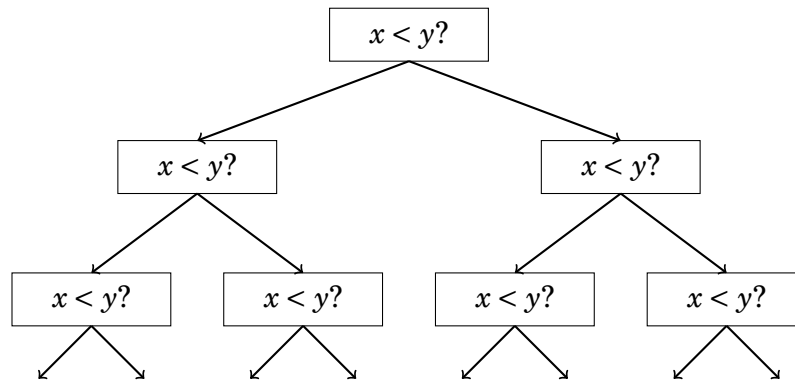
Here "$x < y$?" means that some elements x and y are compared. If $x < y$, the process continues to the left, and otherwise to the right. The results of the process are the possible ways to sort the array, a total of $n!$ ways. For this reason, the height of the tree must be at least

$$\log_2(n!) = \log_2(1) + \log_2(2) + \cdots + \log_2(n).$$

We get an lower bound for this sum by choosing last $n/2$ elements and changing the value of each element to $\log_2(n/2)$. This yields an estimate

$$\log_2(n!) \geq (n/2) \cdot \log_2(n/2),$$

so the height of the tree and the minimum possible number of steps in a sorting algorithm in the worst case is at least $n \log n$.

Counting sort

The lower bound $n \log n$ does not apply to algorithms that do not compare array elements but use some other information. An example of such an algorithm is **counting sort** that sorts an array in $O(n)$ time assuming that every element in the array is an integer between $0 \ldots c$ where c is a small constant.

The algorithm creates a *bookkeeping* array whose indices are elements in the original array. The algorithm iterates through the original array and calculates how many times each element appears in the array.

For example, the array

1	2	3	4	5	6	7	8
1	3	6	9	9	3	5	9

corresponds to the following bookkeeping array:

1	2	3	4	5	6	7	8	9
1	0	2	0	1	1	0	0	3

For example, the value at position 3 in the bookkeeping array is 2, because the element 3 appears 2 times in the original array (positions 2 and 6).

The construction of the bookkeeping array takes $O(n)$ time. After this, the sorted array can be created in $O(n)$ time because the number of occurrences of each element can be retrieved from the bookkeeping array. Thus, the total time complexity of counting sort is $O(n)$.

Counting sort is a very efficient algorithm but it can only be used when the constant c is so small that the array elements can be used as indices in the bookkeeping array.

3.2 Sorting in C++

It is almost never a good idea to use a self-made sorting algorithm in a contest, because there are good implementations available in programming languages. For example, the C++ standard library contains the function `sort` that can be easily used for sorting arrays and other data structures.

There are many benefits in using a library function. First, it saves time because there is no need to implement the function. In addition, the library implementation is certainly correct and efficient: it is not probable that a self-made sorting function would be better.

In this section we will see how to use the C++ `sort` function. The following code sorts a vector in increasing order:

```
vector<int> v = {4,2,5,3,5,8,3};
sort(v.begin(),v.end());
```

After the sorting, the contents of the vector will be $[2,3,3,4,5,5,8]$. The default sorting order is increasing, but a reverse order is possible as follows:

```
sort(v.rbegin(),v.rend());
```

An ordinary array can be sorted as follows:

```
int n = 7; // array size
int t[] = {4,2,5,3,5,8,3};
sort(t,t+n);
```

The following code sorts the string s:

```
string s = "monkey";
sort(s.begin(), s.end());
```

Sorting a string means that the characters in the string are sorted. For example, the string "monkey" becomes "ekmnoy".

Comparison operators

The function sort requires that a **comparison operator** is defined for the data type of the elements to be sorted. During the sorting, this operator will be used whenever it is needed to find out the order of two elements.

Most C++ data types have a built-in comparison operator, and elements of those types can be sorted automatically. For example, numbers are sorted according to their values and strings are sorted in alphabetical order.

Pairs (pair) are sorted primarily by their first elements (first). However, if the first elements of two pairs are equal, they are sorted by their second elements (second):

```
vector<pair<int,int>> v;
v.push_back({1,5});
v.push_back({2,3});
v.push_back({1,2});
sort(v.begin(), v.end());
```

After this, the order of the pairs is $(1,2)$, $(1,5)$ and $(2,3)$.

In a similar way, tuples (tuple) are sorted primarily by the first element, secondarily by the second element, etc.:

```
vector<tuple<int,int,int>> v;
v.push_back(make_tuple(2,1,4));
v.push_back(make_tuple(1,5,3));
v.push_back(make_tuple(2,1,3));
sort(v.begin(), v.end());
```

After this, the order of the tuples is $(1,5,3)$, $(2,1,3)$ and $(2,1,4)$.

User-defined structs

User-defined structs do not have a comparison operator automatically. The operator should be defined inside the struct as a function `operator<` whose parameter is another element of the same type. The operator should return `true` if the element is smaller than the parameter, and `false` otherwise.

For example, the following struct `P` contains the x and y coordinate of a point. The comparison operator is defined so that the points are sorted primarily by the x coordinate and secondarily by the y coordinate.

```
struct P {
    int x, y;
    bool operator<(const P &p) {
        if (x != p.x) return x < p.x;
        else return y < p.y;
    }
};
```

Comparison functions

It is also possible to give an external **comparison function** to the `sort` function as a callback function. For example, the following comparison function sorts strings primarily by length and secondarily by alphabetical order:

```
bool cmp(string a, string b) {
    if (a.size() != b.size()) return a.size() < b.size();
    return a < b;
}
```

Now a vector of strings can be sorted as follows:

```
sort(v.begin(), v.end(), cmp);
```

3.3 Binary search

A general method for searching for an element in an array is to use a `for` loop that iterates through the elements in the array. For example, the following code searches for an element x in the array `t`:

```
for (int i = 1; i <= n; i++) {
    if (t[i] == x) // x found at index i
}
```

The time complexity of this approach is $O(n)$, because in the worst case, it is needed to check all elements in the array. If the array may contain any elements,

this is also the best possible approach, because there is no additional information available where in the array we should search for the element x.

However, if the array is *sorted*, the situation is different. In this case it is possible to perform the search much faster, because the order of the elements in the array guides the search. The following **binary search** algorithm efficiently searches for an element in a sorted array in $O(\log n)$ time.

Method 1

The traditional way to implement binary search resembles looking for a word in a dictionary. At each step, the search halves the active region in the array, until the target element is found, or it turns out that there is no such element.

First, the search checks the middle element of the array. If the middle element is the target element, the search terminates. Otherwise, the search recursively continues to the left or right half of the array, depending on the value of the middle element.

The above idea can be implemented as follows:

```
int a = 1, b = n;
while (a <= b) {
    int k = (a+b)/2;
    if (t[k] == x) // x found at index k
    if (t[k] > x) b = k-1;
    else a = k+1;
}
```

The algorithm maintains a range $a \ldots b$ that corresponds to the active region of the array. Initially, the range is $1 \ldots n$, the whole array. The algorithm halves the size of the range at each step, so the time complexity is $O(\log n)$.

Method 2

An alternative method for implementing binary search is based on an efficient way to iterate through the elements in the array. The idea is to make jumps and slow the speed when we get closer to the target element.

The search goes through the array from left to right, and the initial jump length is $n/2$. At each step, the jump length will be halved: first $n/4$, then $n/8$, $n/16$, etc., until finally the length is 1. After the jumps, either the target element has been found or we know that it does not appear in the array.

The following code implements the above idea:

```
int k = 1;
for (int b = n/2; b >= 1; b /= 2) {
    while (k+b <= n && t[k+b] <= x) k += b;
}
if (t[k] == x) // x was found at index k
```

The variables k and b contain the position in the array and the jump length. If the array contains the element x, the position of x will be in the variable k after the search. The time complexity of the algorithm is $O(\log n)$, because the code in the `while` loop is performed at most twice for each jump length.

Finding the smallest solution

In practice, it is seldom needed to implement binary search for searching elements in an array, because we can use the standard library. For example, the C++ functions `lower_bound` and `upper_bound` implement binary search, and the data structure `set` maintains a set of elements with $O(\log n)$ time operations.

However, an important use for binary search is to find the position where the value of a function changes. Suppose that we wish to find the smallest value k that is a valid solution for a problem. We are given a function $\text{ok}(x)$ that returns `true` if x is a valid solution and `false` otherwise. In addition, we know that $\text{ok}(x)$ is `false` when $x < k$ and `true` when $x \geq k$. The situation looks as follows:

x	0	1	$\cdots$	$k-1$	k	$k+1$	$\cdots$
$\text{ok}(x)$	false	false	$\cdots$	false	true	true	$\cdots$

Now, the value k can be found using binary search:

```
int x = -1;
for (int b = z; b >= 1; b /= 2) {
    while (!ok(x+b)) x += b;
}
int k = x+1;
```

The search finds the largest value of x for which $\text{ok}(x)$ is `false`. Thus, the next value $k = x+1$ is the smallest possible value for which $\text{ok}(k)$ is `true`. The initial jump length z has to be large enough, for example some value for which we know beforehand that $\text{ok}(z)$ is `true`.

The algorithm calls the function ok $O(\log z)$ times, so the total time complexity depends on the function ok. For example, if the function works in $O(n)$ time, the total time complexity is $O(n \log z)$.

Finding the maximum value

Binary search can also be used to find the maximum value for a function that is first increasing and then decreasing. Our task is to find a value k such that

- $f(x) < f(x+1)$ when $x < k$, and

- $f(x) > f(x+1)$ when $x \geq k$.

The idea is to use binary search for finding the largest value of x for which $f(x) < f(x+1)$. This implies that $k = x+1$ because $f(x+1) > f(x+2)$. The following code implements the search:

```
int x = -1;
for (int b = z; b >= 1; b /= 2) {
    while (f(x+b) < f(x+b+1)) x += b;
}
int k = x+1;
```

Note that unlike in the ordinary binary search, here it is not allowed that consecutive values of the function are equal. In this case it would not be possible to know how to continue the search.

Chapter 4

Data structures

A **data structure** is a way to store data in the memory of the computer. It is important to choose an appropriate data structure for a problem, because each data structure has its own advantages and disadvantages. The crucial question is: which operations are efficient in the chosen data structure?

This chapter introduces the most important data structures in the C++ standard library. It is a good idea to use the standard library whenever possible, because it will save a lot of time. Later in the book we will learn more sophisticated data structures that are not available in the standard library.

4.1 Dynamic arrays

A **dynamic array** is an array whose size can be changed during the execution of the program. The most popular dynamic array in C++ is the `vector` structure, that can be used almost like an ordinary array.

The following code creates an empty vector and adds three elements to it:

```
vector<int> v;
v.push_back(3); // [3]
v.push_back(2); // [3,2]
v.push_back(5); // [3,2,5]
```

After this, the elements can be accessed like in an ordinary array:

```
cout << v[0] << "\n"; // 3
cout << v[1] << "\n"; // 2
cout << v[2] << "\n"; // 5
```

The function `size` returns the number of elements in the vector. The following code iterates through the vector and prints all elements in it:

```
for (int i = 0; i < v.size(); i++) {
    cout << v[i] << "\n";
}
```

A shorter way to iterate trough a vector is as follows:

```
for (auto x : v) {
    cout << x << "\n";
}
```

The function `back` returns the last element in the vector, and the function `pop_back` removes the last element:

```
vector<int> v;
v.push_back(5);
v.push_back(2);
cout << v.back() << "\n"; // 2
v.pop_back();
cout << v.back() << "\n"; // 5
```

The following code creates a vector with five elements:

```
vector<int> v = {2,4,2,5,1};
```

Another way to create a vector is to give the number of elements and the initial value for each element:

```
// size 10, initial value 0
vector<int> v(10);
```

```
// size 10, initial value 5
vector<int> v(10, 5);
```

The internal implementation of the vector uses an ordinary array. If the size of the vector increases and the array becomes too small, a new array is allocated and all the elements are moved to the new array. However, this does not happen often and the average time complexity of `push_back` is $O(1)$.

The `string` structure is also a dynamic array that can be used almost like a vector. In addition, there is special syntax for strings that is not available in other data structures. Strings can be combined using the + symbol. The function `substr(k,x)` returns the substring that begins at position k and has length x, and the function `find(t)` finds the position of the first occurrence of a substring `t`.

The following code presents some string operations:

```
string a = "hatti";
string b = a+a;
cout << b << "\n"; // hattihatti
b[5] = 'v';
cout << b << "\n"; // hattivatti
string c = b.substr(3,4);
cout << c << "\n"; // tiva
```

4.2 Set structures

A **set** is a data structure that maintains a collection of elements. The basic operations in a set are element insertion, search and removal.

C++ contains two set implementations: set and unordered_set. The structure set is based on a balanced binary tree and the time complexity of its operations is $O(\log n)$. The structure unordered_set uses a hash table, and the time complexity of its operations is $O(1)$ on average.

The choice which set implementation to use is often a matter of taste. The benefit in the set structure is that it maintains the order of the elements and provides functions that are not available in unordered_set. On the other hand, unordered_set is often more efficient.

The following code creates a set that consists of integers, and shows some of the operations. The function insert adds an element to the set, the function count returns the number of occurrences of an element, and the function erase removes an element from the set.

```
set<int> s;
s.insert(3);
s.insert(2);
s.insert(5);
cout << s.count(3) << "\n"; // 1
cout << s.count(4) << "\n"; // 0
s.erase(3);
s.insert(4);
cout << s.count(3) << "\n"; // 0
cout << s.count(4) << "\n"; // 1
```

A set can be used mostly like a vector, but it is not possible to access the elements using the [] notation. The following code creates a set, prints the number of elements in it, and then iterates through all the elements:

```
set<int> s = {2,5,6,8};
cout << s.size() << "\n"; // 4
for (auto x : s) {
    cout << x << "\n";
}
```

An important property of sets that all the elements are *distinct*. Thus, the function count always returns either 0 (the element is not in the set) or 1 (the element is in the set), and the function insert never adds an element to the set if it is already there. The following code illustrates this:

```
set<int> s;
s.insert(5);
s.insert(5);
s.insert(5);
cout << s.count(5) << "\n"; // 1
```

C++ also contains the structures `multiset` and `unordered_multiset` that work otherwise like `set` and `unordered_set` but they can contain multiple instances of an element. For example, in the following code all three instances of the number 5 are added to a multiset:

```
multiset<int> s;
s.insert(5);
s.insert(5);
s.insert(5);
cout << s.count(5) << "\n"; // 3
```

The function `erase` removes all instances of an element from a multiset:

```
s.erase(5);
cout << s.count(5) << "\n"; // 0
```

Often, only one instance should be removed, which can be done as follows:

```
s.erase(s.find(5));
cout << s.count(5) << "\n"; // 2
```

4.3 Map structures

A **map** is a generalized array that consists of key-value-pairs. While the keys in an ordinary array are always the consecutive integers $0, 1, \ldots, n-1$, where n is the size of the array, the keys in a map can be of any data type and they do not have to be consecutive values.

C++ contains two map implementations that correspond to the set implementations: the structure `map` is based on a balanced binary tree and accessing elements takes $O(\log n)$ time, while the structure `unordered_map` uses a hash map and accessing elements takes $O(1)$ time on average.

The following code creates a map where the keys are strings and the values are integers:

```
map<string,int> m;
m["monkey"] = 4;
m["banana"] = 3;
m["harpsichord"] = 9;
cout << m["banana"] << "\n"; // 3
```

If the value of a key is requested but the map does not contain it, the key is automatically added to the map with a default value. For example, in the following code, the key "aybabtu" with value 0 is added to the map.

```
map<string,int> m;
cout << m["aybabtu"] << "\n"; // 0
```

The function `count` checks if a key exists in a map:

```
if (m.count("aybabtu")) {
    cout << "key exists in the map";
}
```

The following code prints all keys and values in a map:

```
for (auto x : m) {
    cout << x.first << " " << x.second << "\n";
}
```

4.4 Iterators and ranges

Many functions in the C++ standard library operate with iterators. An **iterator** is a variable that points to an element in a data structure.

Often used iterators are `begin` and `end` that define a range that contains all elements in a data structure. The iterator `begin` points to the first element in the data structure, and the iterator `end` points to the position *after* the last element. The situation looks as follows:

```
{  3,  4,  6,  8,  12,  13,  14,  17  }
↑                                    ↑
s.begin()                            s.end()
```

Note the asymmetry in the iterators: `s.begin()` points to an element in the data structure, while `s.end()` points outside the data structure. Thus, the range defined by the iterators is *half-open*.

Working with ranges

Iterators are used in C++ standard library functions that are given a range of elements in a data structure. Usually, we want to process all elements in a data structure, so the iterators `begin` and `end` are given for the function.

For example, the following code sorts a vector using the function `sort`, then reverses the order of the elements using the function `reverse`, and finally shuffles the order of the elements using the function `random_shuffle`.

```
sort(v.begin(), v.end());
reverse(v.begin(), v.end());
random_shuffle(v.begin(), v.end());
```

These functions can also be used with an ordinary array. In this case, the functions are given pointers to the array instead of iterators:

```
sort(t, t+n);
reverse(t, t+n);
random_shuffle(t, t+n);
```

Set iterators

Iterators are often used to access elements of a set. The following code creates an
iterator `it` that points to the first element in the set:

```
set<int>::iterator it = s.begin();
```

A shorter way to write the code is as follows:

```
auto it = s.begin();
```

The element to which an iterator points can be accessed through the * symbol.
For example, the following code prints the first element in the set:

```
auto it = s.begin();
cout << *it << "\n";
```

Iterators can be moved using the operators ++ (forward) and -- (backward),
meaning that the iterator moves to the next or previous element in the set.
    The following code prints all elements in the set:

```
for (auto it = s.begin(); it != s.end(); it++) {
    cout << *it << "\n";
}
```

The following code prints the last element in the set:

```
auto it = s.end();
it--;
cout << *it << "\n";
```

The function $\texttt{find}(x)$ returns an iterator that points to an element whose
value is x. However, if the set does not contain x, the iterator will be end.

```
auto it = s.find(x);
if (it == s.end()) cout << "x is missing";
```

The function `lower_bound`(x) returns an iterator to the smallest element
whose value is *at least x*, and the function `upper_bound`(x) returns an iterator
to the smallest element whose value is *larger than x*. If such elements do not
exist, the return value of the functions will be end. These functions are not
supported by the `unordered_set` structure that does not maintain the order of
the elements.
    For example, the following code finds the element nearest to x:

```
auto a = s.lower_bound(x);
if (a == s.begin() && a == s.end()) {
    cout << "the set is empty\n";
} else if (a == s.begin()) {
    cout << *a << "\n";
} else if (a == s.end()) {
    a--;
    cout << *a << "\n";
} else {
    auto b = a; b--;
    if (x-*b < *a-x) cout << *b << "\n";
    else cout << *a << "\n";
}
```

The code goes through all possible cases using the iterator a. First, the iterator points to the smallest element whose value is at least x. If a is both begin and end at the same time, the set is empty. If a equals begin, the corresponding element is nearest to x. If a equals end, the last element in the set is nearest to x. If none of the previous cases holds, the element nearest to x is either the element that corresponds to a or the previous element.

4.5 Other structures

Bitsets

A bitset is an array where each element is either 0 or 1. For example, the following code creates a bitset that contains 10 elements:

```
bitset<10> s;
s[2] = 1;
s[5] = 1;
s[6] = 1;
s[8] = 1;
cout << s[4] << "\n"; // 0
cout << s[5] << "\n"; // 1
```

The benefit in using bitsets is that they require less memory than ordinary arrays, because each element in a bitset only uses one bit of memory. For example, if n bits are stored in an int array, $32n$ bits of memory will be used, but a corresponding bitset only requires n bits of memory. In addition, the values of a bitset can be efficiently manipulated using bit operators, which makes it possible to optimize algorithms using bit sets.

The following code shows another way to create a bitset:

```
bitset<10> s(string("0010011010"));
cout << s[4] << "\n"; // 0
cout << s[5] << "\n"; // 1
```

The function `count` returns the number of ones in the bitset:

```
bitset<10> s(string("0010011010"));
cout << s.count() << "\n"; // 4
```

The following code shows examples of using bit operations:

```
bitset<10> a(string("0010110110"));
bitset<10> b(string("1011011000"));
cout << (a&b) << "\n"; // 0010010000
cout << (a|b) << "\n"; // 1011111110
cout << (a^b) << "\n"; // 1001101110
```

Deques

A `deque` is a dynamic array whose size can be changed at both ends of the array. Like a vector, a deque contains the functions `push_back` and `pop_back`, but it also contains the functions `push_front` and `pop_front` that are not available in a vector.

A deque can be used as follows:

```
deque<int> d;
d.push_back(5); // [5]
d.push_back(2); // [5,2]
d.push_front(3); // [3,5,2]
d.pop_back(); // [3,5]
d.pop_front(); // [5]
```

The internal implementation of a deque is more complex than that of a vector. For this reason, a deque is slower than a vector. Still, the time complexity of adding and removing elements is $O(1)$ on average at both ends.

Stacks

A `stack` is a data structure that provides two $O(1)$ time operations: adding an element to the top, and removing an element from the top. It is only possible to access the top element of a stack.

The following code shows how a stack can be used:

```
stack<int> s;
s.push(3);
s.push(2);
s.push(5);
cout << s.top(); // 5
s.pop();
cout << s.top(); // 2
```

Queues

A `queue` also provides two $O(1)$ time operations: adding a element to the end of the queue, and removing the first element in the queue. It is only possible to access the first and last element of a queue.

The following code shows how a queue can be used:

```
queue<int> s;
s.push(3);
s.push(2);
s.push(5);
cout << s.front(); // 3
s.pop();
cout << s.front(); // 2
```

Priority queues

A `priority_queue` maintains a set of elements. The supported operations are insertion and, depending on the type of the queue, retrieval and removal of either the minimum or maximum element. The time complexity is $O(\log n)$ for insertion and removal and $O(1)$ for retrieval.

While a set structure efficiently supports all the operations of a priority queue, the benefit in using a priority queue is that it has smaller constant factors. A priority queue is usually implemented using a heap structure that is much simpler than a balanced binary tree needed for an ordered set.

As default, the elements in the C++ priority queue are sorted in decreasing order, and it is possible to find and remove the largest element in the queue. The following code illustrates this:

```
priority_queue<int> q;
q.push(3);
q.push(5);
q.push(7);
q.push(2);
cout << q.top() << "\n"; // 7
q.pop();
cout << q.top() << "\n"; // 5
q.pop();
q.push(6);
cout << q.top() << "\n"; // 6
q.pop();
```

Using the following declaration, we can create a priority queue that allows us to find and remove the minimum element:

```
priority_queue<int,vector<int>,greater<int>> q;
```

4.6 Comparison to sorting

Often it is possible to solve a problem using either data structures or sorting. Sometimes there are remarkable differences in the actual efficiency of these approaches, which may be hidden in their time complexities.

Let us consider a problem where we are given two lists A and B that both contain n integers. Our task is to calculate the number of integers that belong to both of the lists. For example, for the lists

$$A = [5,2,8,9,4] \quad \text{and} \quad B = [3,2,9,5],$$

the answer is 3 because the numbers 2, 5 and 9 belong to both of the lists.

A straightforward solution to the problem is to go through all pairs of numbers in $O(n^2)$ time, but next we will concentrate on more efficient algorithms.

Algorithm 1

We construct a set of the numbers that appear in A, and after this, we iterate through the numbers in B and check for each number if it also belongs to A. This is efficient because the numbers of A are in a set. Using the set structure, the time complexity of the algorithm is $O(n \log n)$.

Algorithm 2

It is not needed to maintain an ordered set, so instead of the set structure we can also use the unordered_set structure. This is an easy way to make the algorithm more efficient, because we only have to change the underlying data structure. The time complexity of the new algorithm is $O(n)$.

Algorithm 3

Instead of data structures, we can use sorting. First, we sort both lists A and B. After this, we iterate through both the lists at the same time and find the common elements. The time complexity of sorting is $O(n \log n)$, and the rest of the algorithm works in $O(n)$ time, so the total time complexity is $O(n \log n)$.

Efficiency comparison

The following table shows how efficient the above algorithms are when n varies and the elements of the lists are random integers between $1 \dots 10^9$:

n	algorithm 1	algorithm 2	algorithm 3
10^6	1,5 s	0,3 s	0,2 s
$2 \cdot 10^6$	3,7 s	0,8 s	0,3 s
$3 \cdot 10^6$	5,7 s	1,3 s	0,5 s
$4 \cdot 10^6$	7,7 s	1,7 s	0,7 s
$5 \cdot 10^6$	10,0 s	2,3 s	0,9 s

Algorithm 1 and 2 are equal except that they use different set structures. In this problem, this choice has an important effect on the running time, because algorithm 2 is 4–5 times faster than algorithm 1.

However, the most efficient algorithm is algorithm 3 that uses sorting. It only uses half of the time compared to algorithm 2. Interestingly, the time complexity of both algorithm 1 and algorithm 3 is $O(n \log n)$, but despite this, algorithm 3 is ten times faster. This can be explained by the fact that sorting is a simple procedure and it is done only once at the beginning of algorithm 3, and the rest of the algorithm works in linear time. On the other hand, algorithm 3 maintains a complex balanced binary tree during the whole algorithm.

Chapter 5

Complete search

Complete search is a general method that can be used to solve almost any algorithm problem. The idea is to generate all possible solutions to the problem using brute force, and then select the best solution or count the number of solutions, depending on the problem.

Complete search is a good technique if there is enough time to go through all the solutions, because the search is usually easy to implement and it always gives the correct answer. If complete search is too slow, other techniques, such as greedy algorithms or dynamic programming, may be needed.

5.1 Generating subsets

We first consider the problem of generating all subsets of a set of n elements. There are two common methods for this: we can either implement a recursive search or use bit operations of integers.

Method 1

An elegant way to go through all subsets of a set is to use recursion. The following function generates the subsets of the set $\{1, 2, \ldots, n\}$. The function maintains a vector that will contain the elements of each subset. The search begins when the function is called with parameter 1.

```
void gen(int k) {
    if (k == n+1) {
        // process subset v
    } else {
        gen(k+1);
        v.push_back(k);
        gen(k+1);
        v.pop_back();
    }
}
```

The parameter k is the next candidate to be included in the subset. The function considers two cases that both generate a recursive call: either k is included or not included in the subset. Finally, when $k = n + 1$, all elements have been processed and one subset has been generated.

The following tree illustrates how the function is called when $n = 3$. We can always choose either the left branch (k is not included in the subset) or the right branch (k is included in the subset).
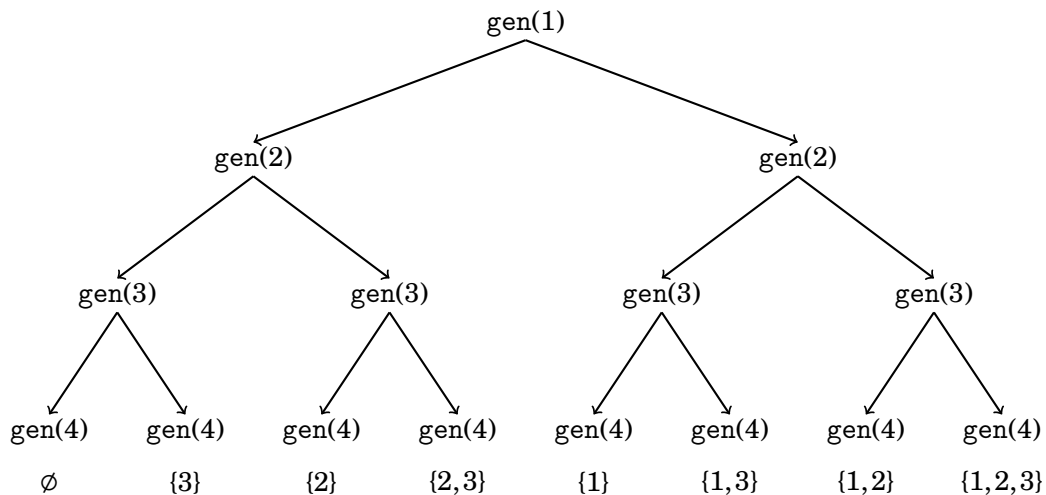
Method 2

Another way to generate subsets is to exploit the bit representation of integers. Each subset of a set of n elements can be represented as a sequence of n bits, which corresponds to an integer between $0 \ldots 2^n - 1$. The ones in the bit sequence indicate which elements are included in the subset.

The usual convention is that the kth element is included in the subset exactly when the kth last bit in the sequence is one. For example, the bit representation of 25 is 11001, that corresponds to the subset $\{1, 4, 5\}$.

The following code goes through all subsets of a set of n elements

```
for (int b = 0; b < (1<<n); b++) {
    // process subset b
}
```

The following code shows how we can find the elements of a subset that corresponds to a bit sequence. When processing each subset, the code builds a vector that contains the elements in the subset.

```
for (int b = 0; b < (1<<n); b++) {
    vector<int> v;
    for (int i = 0; i < n; i++) {
        if (b&(1<<i)) v.push_back(i+1);
    }
}
```

5.2   Generating permutations

Next we will consider the problem of generating all permutations of a set of n elements. Again, there are two approaches: we can either use recursion or go through the permutations iteratively.

Method 1

Like subsets, permutations can be generated using recursion. The following function goes through the permutations of the set $\{1,2,\ldots,n\}$. The function builds a vector that contains the elements in the permutation, and the search begins when the function is called without parameters.

```
void gen() {
    if (v.size() == n) {
        // process permutation v
    } else {
        for (int i = 1; i <= n; i++) {
            if (p[i]) continue;
            p[i] = 1;
            v.push_back(i);
            gen();
            p[i] = 0;
            v.pop_back();
        }
    }
}
```

Each function call adds a new element to the vector v. The array p indicates which elements are already included in the permutation: if $p[k]=0$, element k is not included, and if $p[k]=1$, element k is included. If the size of v equals the size of the set, a permutation has been generated.

Method 2

Another method for generating permutations is to begin with the permutation $\{1,2,\ldots,n\}$ and repeatedly use a function that constructs the next permutation in increasing order. The C++ standard library contains the function next_permutation that can be used for this:

```
vector<int> v;
for (int i = 1; i <= n; i++) {
    v.push_back(i);
}
do {
    // process permutation v
} while (next_permutation(v.begin(),v.end()));
```

5.3 Backtracking

A **backtracking** algorithm begins with an empty solution and extends the solution step by step. The search recursively goes through all different ways how a solution can be constructed.

As an example, consider the **queen problem** where the task is to calculate the number of ways we can place n queens to an $n \times n$ chessboard so that no two queens attack each other. For example, when $n = 4$, there are two possible solutions to the problem:
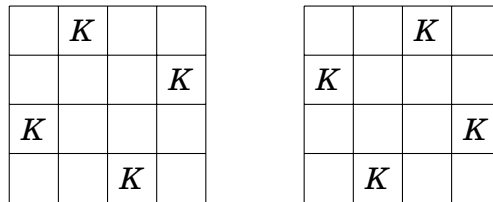
The problem can be solved using backtracking by placing queens to the board row by row. More precisely, exactly one queen will be placed to each row so that no queen attacks any of the queens placed before. A solution has been found when all n queens have been placed to the board.

For example, when $n = 4$, some partial solutions generated by the backtracking algorithm are as follows:
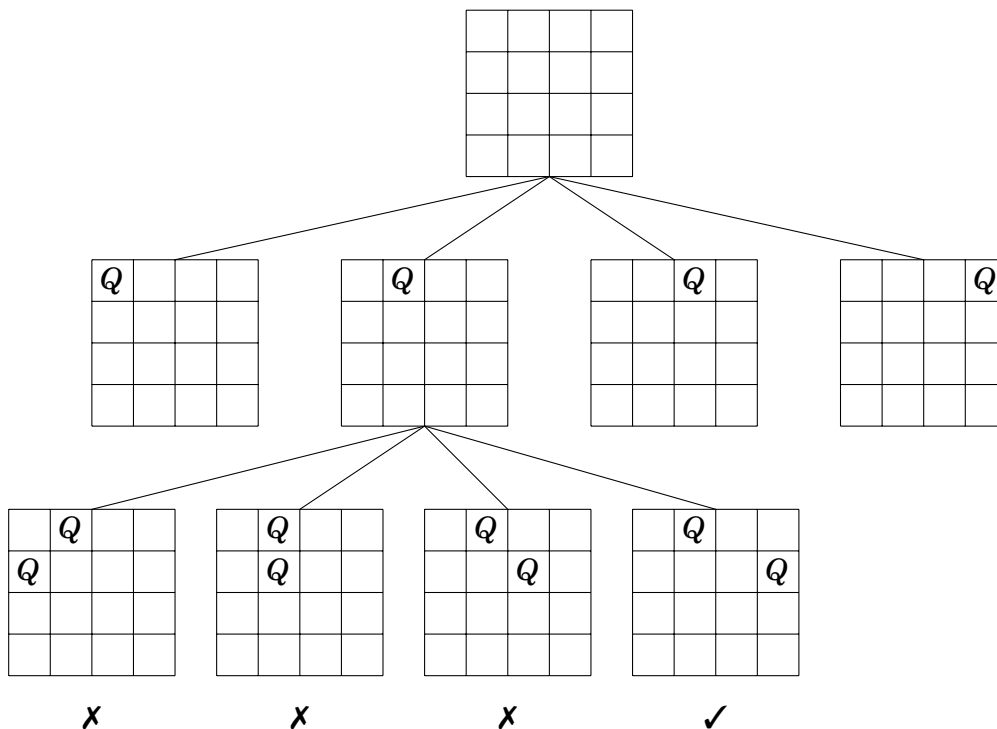
At the bottom level, the three first boards are not valid, because the queens attack each other. However, the fourth board is valid and it can be extended to a complete solution by placing two more queens to the board.

The following code implements the search:

```
void search(int y) {
    if (y == n) {
        c++;
        return;
    }
    for (int x = 0; x < n; x++) {
        if (r1[x] || r2[x+y] || r3[x-y+n-1]) continue;
        r1[x] = r2[x+y] = r3[x-y+n-1] = 1;
        search(y+1);
        r1[x] = r2[x+y] = r3[x-y+n-1] = 0;
    }
}
```

The search begins by calling search(0). The size of the board is in the variable n, and the code calculates the number of solutions to the variable c.

The code assumes that the rows and columns of the board are numbered from 0. The function places a queen to row y where $0 \le y < n$. Finally, if $y = n$, a solution has been found and the variable c is increased by one.

The array r1 keeps track of the columns that already contain a queen, and the arrays r2 and r3 keep track of the diagonals. It is not allowed to add another queen to a column or diagonal that already contains a queen. For example, the rows and the diagonals of the 4×4 board are numbered as follows:

0	1	2	3
0	1	2	3
0	1	2	3
0	1	2	3

0	1	2	3
1	2	3	4
2	3	4	5
3	4	5	6

3	4	5	6
2	3	4	5
1	2	3	4
0	1	2	3

r1   r2   r3

The above backtracking algorithm tells us that there are 92 ways to place 8 queens to the 8×8 chessboard. When n increases, the search quickly becomes slow, because the number of the solutions increases exponentially. For example, calculating the ways to place 16 queens to the 16×16 chessboard already takes about a minute on a modern computer (there are 14772512 solutions).

5.4 Pruning the search

We can often optimize backtracking by pruning the search tree. The idea is to add "intelligence" to the algorithm so that it will notice as soon as possible if a partial solution cannot be extended to a complete solution. Such optimizations can have a tremendous effect on the efficiency of the search.

Let us consider a problem of calculating the number of paths in an $n \times n$ grid from the upper-left corner to the lower-right corner so that each square will be visited exactly once. For example, in a 7×7 grid, there are 111712 such paths. One of the paths is as follows:
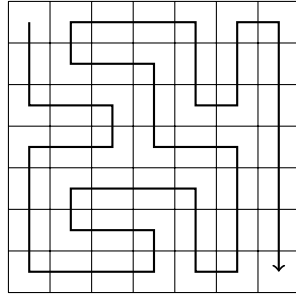
We will concentrate on the 7×7 case, because its level of difficulty is appropriate to our needs. We begin with a straightforward backtracking algorithm, and then optimize it step by step using observations how the search can be pruned. After each optimization, we measure the running time of the algorithm and the number of recursive calls, so that we will clearly see the effect of each optimization on the efficiency of the search.

Basic algorithm

The first version of the algorithm does not contain any optimizations. We simply use backtracking to generate all possible paths from the upper-left corner to the lower-right corner and count the number of such paths.

- running time: 483 seconds

- recursive calls: 76 billions

Optimization 1

In any solution, we first move one step down or right. There are always two paths that are symmetric about the diagonal of the grid after the first step. For example, the following paths are symmetric:
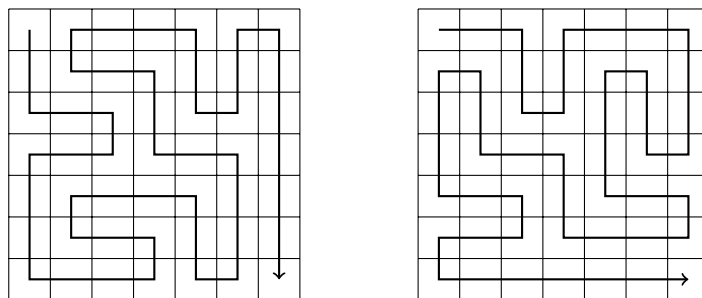
Hence, we can decide that we always first move one step down, and finally multiply the number of the solutions by two.

- running time: 244 seconds

- recursive calls: 38 billions

Optimization 2

If the path reaches the lower-right square before it has visited all other squares of the grid, it is clear that it will not be possible to complete the solution. An example of this is the following path:
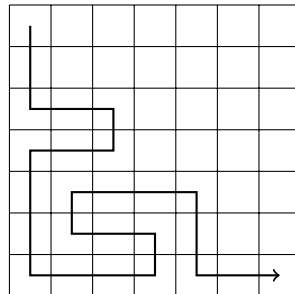
Using this observation, we can terminate the search immediately if we reach the lower-right square too early.

- running time: 119 seconds

- recursive calls: 20 billions

Optimization 3

If the path touches the wall so that there is an unvisited square on both sides, the grid splits into two parts. For example, in the following path both the left and right squares are unvisited:
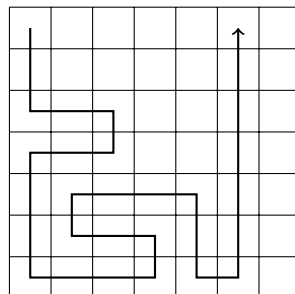
Now it will not be possible to visit every square, so we can terminate the search. It turns out that this optimization is very useful:

- running time: 1.8 seconds

- recursive calls: 221 millions

Optimization 4

The idea of the previous optimization can be generalized: the grid splits into two parts if the top and bottom neighbors of the current square are unvisited and the left and right neighbors are wall or visited (or vice versa).

For example, in the following path the top and bottom neighbors are unvisited, so the path cannot visit all squares in the grid anymore:
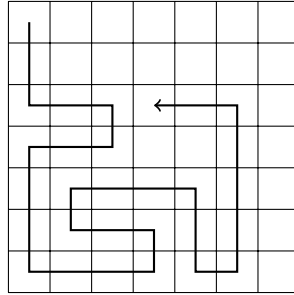
Thus, we can terminate the search in all such cases. After this optimization, the search will be very efficient:

- running time: 0.6 seconds

- recursive calls: 69 millions

Now it is a good moment to stop optimizing the algorithm and see what we have achieved. The running time of the original algorithm was 483 seconds, and now after the optimizations, the running time is only 0.6 seconds. Thus, the algorithm became nearly 1000 times faster thanks to the optimizations.

This is a usual phenomenon in backtracking, because the search tree is usually large and even simple observations can effectively prune the search. Especially useful are optimizations that occur during the first steps of the algorithm, i.e., at the top of the search tree.

5.5 Meet in the middle

Meet in the middle is a technique where the search space is divided into two parts of about equal size. A separate search is performed for both of the parts, and finally the results of the searches are combined.

The technique can be used if there is an efficient way to combine the results of the searches. In such a situation, the two searches may require less time than one large search. Typically, we can turn a factor of 2^n into a factor of $2^{n/2}$ using the meet in the middle technique.

As an example, consider a problem where we are given a list of n numbers and a number x. Our task is to find out if it is possible to choose some numbers from the list so that their sum is x. For example, given the list $[2,4,5,9]$ and $x = 15$, we can choose the numbers $[2,4,9]$ to get $2 + 4 + 9 = 15$. However, if $x = 10$, it is not possible to form the sum.

A standard solution for the problem is to go through all subsets of the elements and check if the sum of any of the subsets is x. The running time of such a solution is $O(2^n)$, because there are 2^n subsets. However, using the meet in the middle technique, we can achieve a more efficient $O(2^{n/2})$ time solution. Note that $O(2^n)$ and $O(2^{n/2})$ are different complexities because $2^{n/2}$ equals $\sqrt{2^n}$.

The idea is to divide the list into two lists A and B such that both lists contain about half of the numbers. The first search generates all subsets of the numbers

in A and stores their sums to a list S_A. Correspondingly, the second search creates a list S_B from B. After this, it suffices to check if it is possible to choose one element from S_A and another element from S_B so that their sum is x. This is possible exactly when there is a way to form the sum x using the numbers in the original list.

For example, suppose that the list is $[2,4,5,9]$ and $x = 15$. First, we divide the list into $A = [2,4]$ and $B = [5,9]$. After this, we create lists $S_A = [0,2,4,6]$ and $S_B = [0,5,9,14]$. In this case, the sum $x = 15$ is possible to form, because we can choose the number 6 from S_A and the number 9 from S_B, which corresponds to the solution $[2,4,9]$.

The time complexity of the algorithm is $O(2^{n/2})$, because both lists A and B contain $n/2$ numbers and it takes $O(2^{n/2})$ time to calculate the sums of their subsets to lists S_A and S_B. After this, it is possible to check in $O(2^{n/2})$ time if the sum x can be formed using the numbers in S_A and S_B.

Chapter 6

Greedy algorithms

A **greedy algorithm** constructs a solution to the problem by always making a choice that looks the best at the moment. A greedy algorithm never takes back its choices, but directly constructs the final solution. For this reason, greedy algorithms are usually very efficient.

The difficulty in designing greedy algorithms is to find a greedy strategy that always produces an optimal solution to the problem. The locally optimal choices in a greedy algorithm should also be globally optimal. It is often difficult to argue that a greedy algorithm works.

6.1 Coin problem

As a first example, we consider a problem where we are given a set of coin values and our task is to form a sum of money using the coins. The values of the coins are $\{c_1, c_2, \ldots, c_k\}$, and each coin can be used as many times we want. What is the minimum number of coins needed?

For example, if the coins are the euro coins (in cents)

$$\{1, 2, 5, 10, 20, 50, 100, 200\}$$

and the sum of money is 520, we need at least four coins. The optimal solution is to select coins $200 + 200 + 100 + 20$ whose sum is 520.

Greedy algorithm

A simple greedy algorithm to the problem is to always select the largest possible coin, until we have constructed the required sum of money. This algorithm works in the example case, because we first select two 200 cent coins, then one 100 cent coin and finally one 20 cent coin. But does this algorithm always work?

It turns out that, for the set of euro coins, the greedy algorithm *always* works, i.e., it always produces a solution with the fewest possible number of coins. The correctness of the algorithm can be shown as follows:

Each coin 1, 5, 10, 50 and 100 appears at most once in an optimal solution. The reason for this is that if the solution would contain two such coins, we could

replace them by one coin and obtain a better solution. For example, if the solution would contain coins $5+5$, we could replace them by coin 10.

In the same way, coins 2 and 20 appear at most twice in an optimal solution, because we could replace coins $2+2+2$ by coins $5+1$ and coins $20+20+20$ by coins $50+10$. Moreover, an optimal solution cannot contain coins $2+2+1$ or $20+20+10$, because we could replace them by coins 5 and 50.

Using these observations, we can show for each coin x that it is not possible to optimally construct a sum x or any larger sum by only using coins that are smaller than x. For example, if $x=100$, the largest optimal sum using the smaller coins is $50+20+20+5+2+2=99$. Thus, the greedy algorithm that always selects the largest coin produces the optimal solution.

This example shows that it can be difficult to argue that a greedy algorithm works, even if the algorithm itself is simple.

General case

In the general case, the coin set can contain any coins and the greedy algorithm *does not* necessarily produce an optimal solution.

We can prove that a greedy algorithm does not work by showing a counterexample where the algorithm gives a wrong answer. In this problem we can easily find a counterexample: if the coins are $\{1,3,4\}$ and the target sum is 6, the greedy algorithm produces the solution $4+1+1$ while the optimal solution is $3+3$.

It is not known if the general coin problem can be solved using any greedy algorithm. However, as we will see in Chapter 7, the general problem can be efficiently solved using a dynamic programming algorithm that always gives the correct answer.

6.2 Scheduling

Many scheduling problems can be solved using greedy algorithms. A classic problem is as follows: Given n events with their starting and ending times, our goal is to plan a schedule that includes as many events as possible. It is not possible to select an event partially. For example, consider the following events:

event	starting time	ending time
A	1	3
B	2	5
C	3	9
D	6	8

In this case the maximum number of events is two. For example, we can select events B and D as follows:
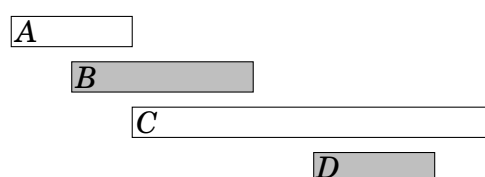
It is possible to invent several greedy algorithms for the problem, but which of them works in every case?

Algorithm 1

The first idea is to select as *short* events as possible. In the example case this algorithm selects the following events:
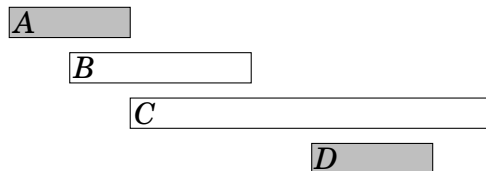
However, selecting short events is not always a correct strategy, but the algorithm fails, for example, in the following case:
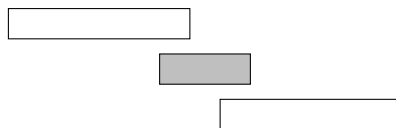
If we select the short event, we can only select one event. However, it would be possible to select both the long events.

Algorithm 2

Another idea is to always select the next possible event that *begins* as *early* as possible. This algorithm selects the following events:
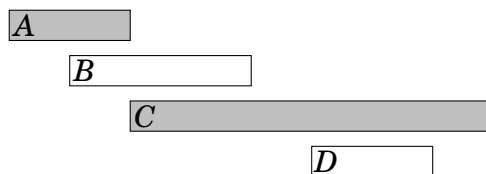
However, we can find a counterexample also for this algorithm. For example, in the following case, the algorithm only selects one event:
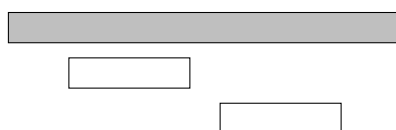
If we select the first event, it is not possible to select any other events. However, it would be possible to select the other two events.

Algorithm 3

The third idea is to always select the next possible event that *ends* as *early* as possible. This algorithm selects the following events:
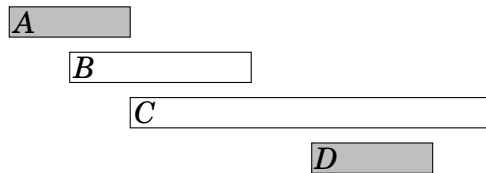
It turns out that this algorithm *always* produces an optimal solution. First, it is always an optimal choice to first select an event that ends as early as possible. After this, it is an optimal choice to select the next event using the same strategy, etc., until we cannot select any more events.

One way to argue that the algorithm works is to consider what happens if we first select an event that ends later than the event that ends as early as possible. Now, we will have at most an equal number of choices how we can select the next event. Hence, selecting an event that ends later can never yield a better solution, and the greedy algorithm is correct.

6.3   Tasks and deadlines

Let us now consider a problem where we are given n tasks with durations and deadlines and our task is to choose an order to perform the tasks. For each task, we earn $d - x$ points where d is the task's deadline and x is the moment when we finished the task. What is the largest possible total score we can obtain?

For example, suppose that the tasks are as follows:

task	duration	deadline
A	4	2
B	3	5
C	2	7
D	4	5

In this case, an optimal schedule for the tasks is as follows:
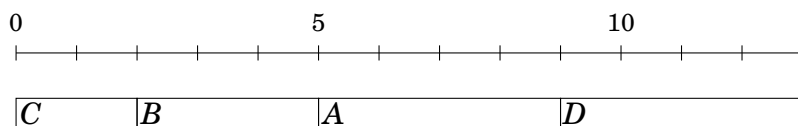
In this solution, C yields 5 points, B yields 0 points, A yields -7 points and D yields -8 points, so the total score is -10.

Surprisingly, the optimal solution to the problem does not depend on the deadlines at all, but a correct greedy strategy is to simply perform the tasks *sorted by their durations* in increasing order. The reason for this is that if we ever perform two tasks one after another such that the first task takes longer than the second task, we can obtain a better solution if we swap the tasks. For example, consider the following schedule:
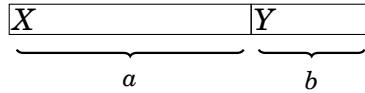
Here $a > b$, so we should swap the tasks:
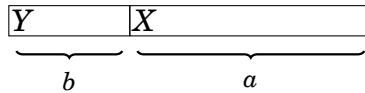
Now X gives b points fewer and Y gives a points more, so the total score increases by $a - b > 0$. In an optimal solution, for any two consecutive tasks, it must hold that the shorter task comes before the longer task. Thus, the tasks must be performed sorted by their durations.

6.4 Minimizing sums

We will next consider a problem where we are given n numbers $a_1, a_2, \ldots, a_n$ and our task is to find a value x that minimizes the sum

$$|a_1 - x|^c + |a_2 - x|^c + \cdots + |a_n - x|^c.$$

We will focus on the cases $c = 1$ and $c = 2$.

Case $c = 1$

In this case, we should minimize the sum

$$|a_1 - x| + |a_2 - x| + \cdots + |a_n - x|.$$

For example, if the numbers are $[1, 2, 9, 2, 6]$, the best solution is to select $x = 2$ which produces the sum

$$|1 - 2| + |2 - 2| + |9 - 2| + |2 - 2| + |6 - 2| = 12.$$

In the general case, the best choice for x is the *median* of the numbers, i.e., the middle number after sorting. For example, the list $[1, 2, 9, 2, 6]$ becomes $[1, 2, 2, 6, 9]$ after sorting, so the median is 2.

The median is an optimal choice, because if x is smaller than the median, the sum becomes smaller by increasing x, and if x is larger then the median, the sum becomes smaller by decreasing x. Hence, the optimal solution is that x is the median. If n is even and there are two medians, both medians and all values between them are optimal solutions.

Case $c = 2$

In this case, we should minimize the sum

$$(a_1 - x)^2 + (a_2 - x)^2 + \cdots + (a_n - x)^2.$$

For example, if the numbers are $[1,2,9,2,6]$, the best solution is to select $x = 4$ which produces the sum

$$(1-4)^2 + (2-4)^2 + (9-4)^2 + (2-4)^2 + (6-4)^2 = 46.$$

In the general case, the best choice for x is the *average* of the numbers. In the example the average is $(1+2+9+2+6)/5 = 4$. This result can be derived by presenting the sum as follows:

$$nx^2 - 2x(a_1 + a_2 + \cdots + a_n) + (a_1^2 + a_2^2 + \cdots + a_n^2)$$

The last part does not depend on x, so we can ignore it. The remaining parts form a function $nx^2 - 2xs$ where $s = a_1 + a_2 + \cdots + a_n$. This is a parabola opening upwards with roots $x = 0$ and $x = 2s/n$, and the minimum value is the average of the roots $x = s/n$, i.e., the average of the numbers $a_1, a_2, \ldots, a_n$.

6.5 Data compression

A **binary code** assigns for each character of a given string a **codeword** that consists of bits. We can *compress* the string using the binary code by replacing each character by the corresponding codeword. For example, the following binary code assigns codewords for characters A–D:

character	codeword
A	00
B	01
C	10
D	11

This is a **constant-length** code which means that the length of each codeword is the same. For example, we can compress the string AABACDACA as follows:

$$00\,00\,01\,00\,10\,11\,00\,10\,00$$

Using this code, the length of the compressed string is 18 bits. However, we can compress the string better if we use a **variable-length** code where codewords may have different lengths. Then we can give short codewords for characters that appear often and long codewords for characters that appear rarely. It turns out that an **optimal** code for the above string is as follows:

character	codeword
A	0
B	110
C	10
D	111

An optimal code produces a compressed string that is as short as possible. In this case, the compressed string using the optimal code is

$$0\,0\,110\,0\,10\,111\,0\,10\,0,$$

so only 15 bits are needed instead of 18 bits. Thus, thanks to a better code it was possible to save 3 bits in the compressed string.

We require that no codeword is a prefix of another codeword. For example, it is not allowed that a code would contain both codewords 10 and 1011. The reason for this is that we want to be able to generate the original string from the compressed string. If a codeword could be a prefix of another codeword, this would not always be possible. For example, the following code is *not* valid:

character	codeword
A	10
B	11
C	1011
D	111

Using this code, it would not be possible to know if the compressed string 1011 corresponds to the string AB or the string C.

Huffman coding

Huffman coding [23] is a greedy algorithm that constructs an optimal code for compressing a given string. The algorithm builds a binary tree based on the frequencies of the characters in the string, and each character's codeword can be read by following a path from the root to the corresponding node. A move to the left corresponds to bit 0, and a move to the right corresponds to bit 1.

Initially, each character of the string is represented by a node whose weight is the number of times the character occurs in the string. Then at each step two nodes with minimum weights are combined by creating a new node whose weight is the sum of the weights of the original nodes. The process continues until all nodes have been combined.

Next we will see how Huffman coding creates the optimal code for the string AABACDACA. Initially, there are four nodes that correspond to the characters in the string:

The node that represents character A has weight 5 because character A appears 5 times in the string. The other weights have been calculated in the same way.

The first step is to combine the nodes that correspond to characters B and D, both with weight 1. The result is:
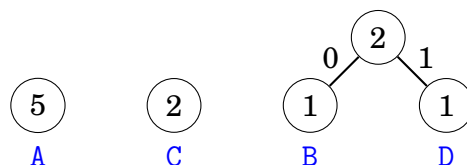
After this, the nodes with weight 2 are combined:
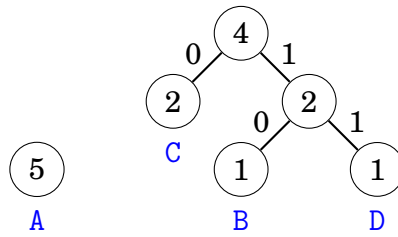
Finally, the two remaining nodes are combined:
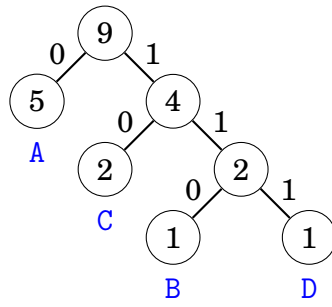
Now all nodes are in the tree, so the code is ready. The following codewords can be read from the tree:

character	codeword
A	0
B	110
C	10
D	111

Chapter 7

Dynamic programming

Dynamic programming is a technique that combines the correctness of complete search and the efficiency of greedy algorithms. Dynamic programming can be applied if the problem can be divided into overlapping subproblems that can be solved independently.

There are two uses for dynamic programming:

- **Finding an optimal solution**: We want to find a solution that is as large as possible or as small as possible.

- **Counting the number of solutions**: We want to calculate the total number of possible solutions.

We will first see how dynamic programming can be used to find an optimal solution, and then we will use the same idea for counting the solutions.

Understanding dynamic programming is a milestone in every competitive programmer's career. While the basic idea of the technique is simple, the challenge is how to apply it to different problems. This chapter introduces a set of classic problems that are a good starting point.

7.1 Coin problem

We first discuss a problem that we have already seen in Chapter 6: Given a set of coin values $\{c_1, c_2, \ldots, c_k\}$ and a sum of money x, our task is to form the sum x using as few coins as possible.

In Chapter 6, we solved the problem using a greedy algorithm that always selects the largest possible coin. The greedy algorithm works, for example, when the coins are the euro coins, but in the general case the greedy algorithm does not necessarily produce an optimal solution.

Now it is time to solve the problem efficiently using dynamic programming, so that the algorithm works for any coin set. The dynamic programming algorithm is based on a recursive function that goes through all possibilities how to form the sum, like a brute force algorithm. However, the dynamic programming algorithm is efficient because it uses *memoization* and calculates the answer to each subproblem only once.

Recursive formulation

The idea in dynamic programming is to formulate the problem recursively so that the answer to the problem can be calculated from answers to smaller subproblems. In the coin problem, a natural recursive problem is as follows: what is the smallest number of coins required for constructing sum x?

Let $f(x)$ be a function that gives the answer to the problem, i.e., $f(x)$ is the smallest number of coins required for constructing sum x. The values of the function depend on the values of the coins. For example, if the coin values are $\{1, 3, 4\}$, the first values of the function are as follows:

$$
\begin{aligned}
f(0) &= 0 \\
f(1) &= 1 \\
f(2) &= 2 \\
f(3) &= 1 \\
f(4) &= 1 \\
f(5) &= 2 \\
f(6) &= 2 \\
f(7) &= 2 \\
f(8) &= 2 \\
f(9) &= 3 \\
f(10) &= 3
\end{aligned}
$$

First, $f(0) = 0$ because no coins are needed for the sum 0. Moreover, $f(3) = 1$ because the sum 3 can be formed using coin 3, and $f(5) = 2$ because the sum 5 can be formed using coins 1 and 4.

The essential property in the function is that each value of $f(x)$ can be calculated recursively from smaller values of the function. For example, if the coin set is $\{1, 3, 4\}$, there are three ways to select the first coin in a solution: we can choose coin 1, 3 or 4. If coin 1 is chosen, the remaining task is to form the sum $x - 1$. Similarly, if coin 3 or 4 is chosen, we should form the sum $x - 3$ or $x - 4$.

Thus, the recursive formula is

$$f(x) = \min(f(x-1), f(x-3), f(x-4)) + 1$$

where the function min gives the smallest of its parameters. In the general case, for the coin set $\{c_1, c_2, \ldots, c_k\}$, the recursive formula is

$$f(x) = \min(f(x-c_1), f(x-c_2), \ldots, f(x-c_k)) + 1.$$

The base case for the function is

$$f(0) = 0,$$

because no coins are needed for constructing the sum 0. In addition, it is convenient to define

$$f(x) = \infty \quad \text{if } x < 0.$$

This means that an infinite number of coins is needed for forming a negative sum of money. This prevents the function from constructing a solution where the initial sum of money is negative.

Once a recursive function that solves the problem has been found, we can directly implement a solution in C++:

```cpp
int f(int x) {
    if (x == 0) return 0;
    if (x < 0) return 1e9;
    int u = 1e9;
    for (int i = 1; i <= k; i++) {
        u = min(u, f(x-c[i])+1);
    }
    return u;
}
```

The code assumes that the available coins are $c[1], c[2], \ldots, c[k]$, and the value 10^9 denotes infinity. This function works but it is not efficient yet, because it goes through a large number of ways to construct the sum. However, the function can be made efficient by using memoization.

Memoization

Dynamic programming allows us to calculate the value of a recursive function efficiently using **memoization**. This means that an auxiliary array is used for recording the values of the function for different parameters. For each parameter, the value of the function is calculated recursively only once, and after this, the value can be directly retrieved from the array.

In this problem, we can use an array

```cpp
int d[N];
```

where $d[x]$ will contain the value of $f(x)$. The constant N has to be chosen so that all required values of the function fit in the array.

After this, the function can be efficiently implemented as follows:

```cpp
int f(int x) {
    if (x == 0) return 0;
    if (x < 0) return 1e9;
    if (d[x]) return d[x];
    int u = 1e9;
    for (int i = 1; i <= k; i++) {
        u = min(u, f(x-c[i])+1);
    }
    d[x] = u;
    return d[x];
}
```

The function handles the base cases $x = 0$ and $x < 0$ as previously. Then the function checks if $f(x)$ has already been calculated in $d[x]$. If the value of $f(x)$ is found in the array, the function directly returns it. Otherwise the function calculates the value recursively and stores it in $d[x]$.

Using memoization the function works efficiently, because the answer for each parameter x is calculated recursively only once. After a value of $f(x)$ has been stored in the array, it can be efficiently retrieved whenever the function will be called again with the parameter x.

The time complexity of the resulting algorithm is $O(xk)$ where the sum is x and the number of coins is k. In practice, the algorithm can be used if x is so small that it is possible to allocate an array for all possible function parameters.

Note that the array can also be constructed using a loop that calculates all the values instead of a recursive function:

```
d[0] = 0;
for (int i = 1; i <= x; i++) {
    int u = 1e9;
    for (int j = 1; j <= k; j++) {
        if (i-c[j] < 0) continue;
        u = min(u, d[i-c[j]]+1);
    }
    d[i] = u;
}
```

This implementation is shorter and somewhat more efficient than recursion, and experienced competitive programmers often prefer dynamic programming solutions that are implemented using loops. Still, the underlying idea is the same as in the recursive function.

Constructing a solution

Sometimes we are asked both to find the value of an optimal solution and also to give an example how such a solution can be constructed. In the coin problem, this means that the algorithm should show how to select the coins that produce the sum x using as few coins as possible.

We can construct the solution by adding another array to the code. The array indicates for each sum of money the first coin that should be chosen in an optimal solution. In the following code, the array e is used for this:

```
d[0] = 0;
for (int i = 1; i <= x; i++) {
    d[i] = 1e9;
    for (int j = 1; j <= k; j++) {
        if (i-c[j] < 0) continue;
        int u = d[i-c[j]]+1;
        if (u < d[i]) {
            d[i] = u;
            e[i] = c[j];
        }
    }
}
```

After this, we can print the coins needed for the sum x as follows:

```
while (x > 0) {
    cout << e[x] << "\n";
    x -= e[x];
}
```

Counting the number of solutions

Let us now consider a variant of the problem that is otherwise like the original problem, but we should count the total number of solutions instead of finding the optimal solution. For example, if the coins are $\{1,3,4\}$ and the target sum is 5, there are a total of 6 solutions:

- $1+1+1+1+1$
- $1+1+3$
- $1+3+1$

- $3+1+1$
- $1+4$
- $4+1$

The number of the solutions can be calculated using the same idea as finding the optimal solution. The difference is that when finding the optimal solution, we maximize or minimize something in the recursion, but now we will calculate sums of numbers of solutions.

To solve the problem, we can define a function $f(x)$ that gives the number of ways to construct the sum x using the coins. For example, $f(5) = 6$ when the coins are $\{1,3,4\}$. The value of $f(x)$ can be calculated recursively using the formula

$$f(x) = f(x - c_1) + f(x - c_2) + \cdots + f(x - c_k),$$

because to form the sum x, we have to first choose some coin c_i and then form the sum $x - c_i$. The base cases are $f(0) = 1$, because there is exactly one way to form the sum 0 using an empty set of coins, and $f(x) = 0$, when $x < 0$, because it is not possible to form a negative sum of money.

If the coin set is $\{1,3,4\}$, the function is

$$f(x) = f(x - 1) + f(x - 3) + f(x - 4)$$

and the first values of the function are:

$$
\begin{aligned}
f(0) &= 1 \\
f(1) &= 1 \\
f(2) &= 1 \\
f(3) &= 2 \\
f(4) &= 4 \\
f(5) &= 6 \\
f(6) &= 9 \\
f(7) &= 15 \\
f(8) &= 25 \\
f(9) &= 40
\end{aligned}
$$

The following code calculates the value of $f(x)$ using dynamic programming by filling the array d for parameters $0\ldots x$:

```
d[0] = 1;
for (int i = 1; i <= x; i++) {
    for (int j = 1; j <= k; j++) {
        if (i-c[j] < 0) continue;
        d[i] += d[i-c[j]];
    }
}
```

Often the number of solutions is so large that it is not required to calculate the exact number but it is enough to give the answer modulo m where, for example, $m = 10^9 + 7$. This can be done by changing the code so that all calculations are done modulo m. In the above code, it suffices to add the line

```
        d[i] %= m;
```

after the line

```
        d[i] += d[i-c[j]];
```

Now we have discussed all basic techniques related to dynamic programming. Since dynamic programming can be used in many different situations, we will now go through a set of problems that show further examples about possibilities of dynamic programming.

7.2   Longest increasing subsequence

Given an array that contains n numbers $x_1, x_2, \ldots, x_n$, our task is to find the **longest increasing subsequence** of the array. This is a sequence of array elements that goes from left to right, and each element in the sequence is larger than the previous element. For example, in the array

1	2	3	4	5	6	7	8
6	2	5	1	7	4	8	3

the longest increasing subsequence contains 4 elements:

1	2	3	4	5	6	7	8
6	2	5	1	7	4	8	3

Let $f(k)$ be the length of the longest increasing subsequence that ends at position k. Using this function, the answer to the problem is the largest of

the values $f(1), f(2), \ldots, f(n)$. For example, in the above array the values of the function are as follows:

$$
\begin{aligned}
f(1) &= 1 \\
f(2) &= 1 \\
f(3) &= 2 \\
f(4) &= 1 \\
f(5) &= 3 \\
f(6) &= 2 \\
f(7) &= 4 \\
f(8) &= 2
\end{aligned}
$$

When calculating the value of $f(k)$, there are two possibilities how the subsequence that ends at position k is constructed:

1. The subsequence only contains the element x_k. In this case $f(k) = 1$.

2. The subsequence is constructed by adding the element x_k to a subsequence that ends at position i where $i < k$ and $x_i < x_k$. In this case $f(k) = f(i) + 1$.

For example, in the above example $f(7) = 4$, because the subsequence $[2, 5, 7]$ of length 3 ends at position 5, and by adding the element at position 7 to this subsequence, we get the optimal subsequence $[2, 5, 7, 8]$ of length 4.

An easy way to calculate the value of $f(k)$ is to go through all previous values $f(1), f(2), \ldots, f(k-1)$ and select the best solution. The time complexity of such an algorithm is $O(n^2)$. Surprisingly, it is also possible to solve the problem in $O(n \log n)$ time. Can you see how?

7.3 Paths in a grid

Our next problem is to find a path in an $n \times n$ grid from the upper-left corner to the lower-right corner such that we only move down and right. Each square contains a number, and the path should be constructed so that the sum of numbers along the path is as large as possible.

The following picture shows an optimal path in a grid:

3	7	9	2	7
9	8	3	5	5
1	7	9	8	5
3	8	6	4	10
6	3	9	7	8

The sum of numbers on the path is 67, and this is the largest possible sum on a path from the upper-left corner to the lower-right corner.

We can approach the problem by calculating for each square (y, x) the maximum sum on a path from the upper-left corner to square (y, x). Let $f(y, x)$ denote this sum, so $f(n, n)$ is the maximum sum on a path from the upper-left corner to the lower-right corner.

The recursive formula is based on the observation that a path that ends at square (y,x) can come either from square $(y,x-1)$ or square $(y-1,x)$:
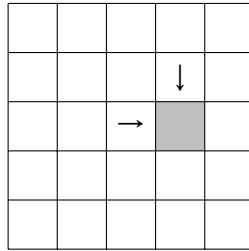
Let $r(y,x)$ denote the number in square (y,x). The base cases for the recursive function are as follows:

$$
\begin{aligned}
f(1,1) &= r(1,1) \\
f(1,x) &= f(1,x-1)+r(1,x) \\
f(y,1) &= f(y-1,1)+r(y,1)
\end{aligned}
$$

In the general case there are two possible paths, and we should select the path that produces the larger sum:

$$f(y,x) = \max(f(y,x-1),f(y-1,x))+r(y,x)$$

The time complexity of the solution is $O(n^2)$, because each value $f(y,x)$ can be calculated in constant time using the values of the adjacent squares.

7.4 Knapsack

Knapsack is a classic problem where we are given n objects with weights $p_1, p_2, \ldots, p_n$ and values $a_1, a_2, \ldots, a_n$. Our task is to choose a subset of the objects such that the sum of the weights is at most x and the sum of the values is as large as possible.

For example, if the objects are

object	weight	value
A	5	1
B	6	3
C	8	5
D	5	3

and the maximum allowed total weight is 12, an optimal solution is to select objects B and D. Their total weight $6+5 = 11$ does not exceed 12, and their total value $3+3 = 6$ is the largest possible.

This task can be solved in two different ways using dynamic programming. We can either regard the problem as maximizing the total value of the objects or minimizing the total weight of the objects.

Solution 1

Maximization: Let $f(k,u)$ denote the largest possible total value when a subset of objects $1 \ldots k$ is selected such that the total weight is u. The solution to the problem is the largest value $f(n,u)$ where $0 \le u \le x$. A recursive formula for calculating the function is

$$f(k,u) = \max(f(k-1,u), f(k-1,u-p_k) + a_k),$$

because we can either include or not include object k in the solution. The base cases are $f(0,0) = 0$ and $f(0,u) = -\infty$ when $u \ne 0$. The time compexity of the solution is $O(nx)$.

In the example case, the optimal solution is $f(4,11) = 6$ that can be constructed using the following sequence:

$$f(4,11) = f(3,6) + 3 = f(2,6) + 3 = f(1,0) + 3 + 3 = f(0,0) + 3 + 3 = 6.$$

Solution 2

Minimization: Let $f(k,u)$ denote the smallest possible total weight when a subset of objects $1 \ldots k$ is selected such that the total weight is u. The solution to the problem is the largest value u for which $0 \le u \le s$ and $f(n,u) \le x$ where $s = \sum_{i=1}^{n} a_i$. A recursive formula for calculating the function is

$$f(k,u) = \min(f(k-1,u), f(k-1,u-a_k) + p_k)$$

as in solution 1. The base cases are $f(0,0) = 0$ and $f(0,u) = \infty$ when $u \ne 0$. The time complexity of the solution is $O(ns)$.

In the example case, the optimal solution is $f(4,6) = 11$ that can be constructed using the following sequence:

$$f(4,6) = f(3,3) + 5 = f(2,3) + 5 = f(1,0) + 6 + 5 = f(0,0) + 6 + 5 = 11.$$

It is interesting to note how the parameters of the input affect the efficiency of the solutions. The efficiency of solution 1 depends on the weights of the objects, while the efficiency of solution 2 depends on the values of the objects.

7.5 Edit distance

The **edit distance** or **Levenshtein distance** is the minimum number of editing operations needed to transform a string into another string. The allowed editing operations are as follows:

- insert a character (e.g. ABC → ABCA)

- remove a character (e.g. ABC → AC)

- change a character (e.g. ABC → ADC)

For example, the edit distance between LOVE and MOVIE is 2, because we can first perform the operation LOVE → MOVE (change) and then the operation MOVE → MOVIE (insertion). This is the smallest possible number of operations, because it is clear that it is not possible to use only one operation.

Suppose we are given strings x and y that contain n and m characters, respectively, and we wish to calculate the edit distance between them. This can be done using dynamic programming in $O(nm)$ time. Let $f(a,b)$ denote the edit distance between the first a characters of x and the first b characters of y. Using this function, the edit distance between x and y equals $f(n,m)$.

The base cases for the function are

$$\begin{aligned} f(0,b) &= b \\ f(a,0) &= a \end{aligned}$$

and in the general case the formula is

$$f(a,b) = \min(f(a,b-1)+1, f(a-1,b)+1, f(a-1,b-1)+c),$$

where $c = 0$ if the ath character of x equals the bth character of y, and otherwise $c = 1$. The formula considers all possible ways to shorten the strings:

- $f(a,b-1)$ means that a character is inserted to x

- $f(a-1,b)$ means that a character is removed from x

- $f(a-1,b-1)$ means that x and y contain the same character ($c = 0$), or a character in x is transformed into a character in y ($c = 1$)

The following table shows the values of f in the example case:

	M	O	V	I	E
	0	1	2	3	4	5
L	1	1	2	3	4	5
O	2	2	1	2	3	4
V	3	3	2	1	2	3
E	4	4	3	2	2	2

The lower-right corner of the table tells us that the edit distance between LOVE and MOVIE is 2. The table also shows how to construct the shortest sequence of editing operations. In this case the path is as follows:

	M	O	V	I	E
	0	1	2	3	4	5
L	1	1	2	3	4	5
O	2	2	1	2	3	4
V	3	3	2	1	2	3
E	4	4	3	2	2	2

The last characters of `LOVE` and `MOVIE` are equal, so the edit distance between them equals the edit distance between `LOV` and `MOVI`. We can use one editing operation to remove the character `I` from `MOVI`. Thus, the edit distance is one larger than the edit distance between `LOV` and `MOV`, etc.

7.6 Counting tilings

Sometimes the states of a dynamic programming solution are more complex than fixed combinations of numbers. As an example, we consider the problem of calculating the number of distinct ways to fill an $n \times m$ grid using 1×2 and 2×1 size tiles. For example, one valid solution for the 4×7 grid is
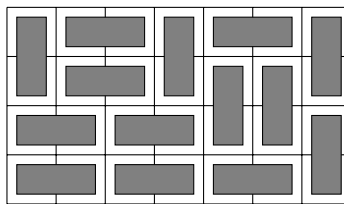
and the total number of solutions is 781.

The problem can be solved using dynamic programming by going through the grid row by row. Each row in a solution can be represented as a string that contains m characters from the set {⊓,⊔,⊏,⊐}. For example, the above solution consists of four rows that correspond to the following strings:

- ⊓⊏⊐⊓⊏⊐⊓

- ⊔⊏⊐⊔⊓⊓⊔

- ⊏⊐⊏⊐⊔⊔⊓

- ⊏⊐⊏⊐⊏⊐⊔

Let $f(k,x)$ denote the number of ways to construct a solution for rows $1 \ldots k$ in the grid so that string x corresponds to row k. It is possible to use dynamic programing here, because the state of a row is constrained only by the state of the previous row.

A solution is valid if row 1 does not contain the character ⊔, row n does not contain the character ⊓, and all consecutive rows are *compatible*. For example, the rows ⊔⊏⊐⊔⊓⊓⊔ and ⊏⊐⊏⊐⊔⊔⊓ are compatible, while the rows ⊓⊏⊐⊓⊏⊐⊓ and ⊏⊐⊏⊐⊏⊐⊔ are not compatible.

Since a row consists of m characters and there are four choices for each character, the number of distinct rows is at most 4^m. Thus, the time complexity of the solution is $O(n4^{2m})$ because we can go through the $O(4^m)$ possible states for each row, and for each state, there are $O(4^m)$ possible states for the previous row. In practice, it is a good idea to rotate the grid so that the shorter side has length m, because the factor 4^{2m} dominates the time complexity.

It is possible to make the solution more efficient by using a better representation for the rows. It turns out that it is sufficient to know which columns of the

previous row contain the upper square of a vertical tile. Thus, we can represent a row using only characters ⊓ and □, where □ is a combination of characters ⊔, ⊏ and ⊐. Using this representation, there are only 2^m distinct rows and the time complexity is $O(n2^{2m})$.

As a final note, there is also a surprising direct formula for calculating the number of tilings[1]:

$$\prod_{a=1}^{\lceil n/2 \rceil} \prod_{b=1}^{\lceil m/2 \rceil} 4 \cdot (\cos^2 \frac{\pi a}{n+1} + \cos^2 \frac{\pi b}{m+1})$$

This formula is very efficient, because it calculates the number of tilings in $O(nm)$ time, but since the answer is a product of real numbers, a practical problem in using the formula is how to store the intermediate results accurately.

---

[1]Surprisingly, this formula was discovered independently by [29] and [38] in 1961.

Chapter 8

Amortized analysis

The time complexity of an algorithm is often easy to analyze just by examining the structure of the algorithm: what loops does the algorithm contain, and how many times the loops are performed. However, sometimes a straightforward analysis does not give a true picture of the efficiency of the algorithm.

Amortized analysis can be used for analyzing algorithms that contain operations whose time complexity varies. The idea is to estimate the total time used for all such operations during the execution of the algorithm, instead of focusing on individual operations.

8.1 Two pointers method

In the **two pointers method**, two pointers are used for iterating through the elements in an array. Both pointers can move during the algorithm, but each pointer can move to one direction only. This restriction ensures that the algorithm works efficiently.

We will next discuss two problems that can be solved using the two pointers method.

Subarray sum

As the first example, we consider a problem where we are given an array of n positive numbers and a target sum x, and we should find a subarray whose sum is x or report that there is no such subarray. For example, the array

	1	2	3	4	5	6	7	8
	1	3	2	5	1	1	2	3

contains a subarray whose sum is 8:

	1	2	3	4	5	6	7	8
	1	3	2	5	1	1	2	3

It turns out that the problem can be solved in $O(n)$ time by using the two pointers method. The idea is that the left and right pointer indicate the first and last element of an subarray. On each turn, the left pointer moves one step forward, and the right pointer moves forward as long as the subarray sum is at most x. If the sum becomes exactly x, a solution has been found.

As an example, consider the following array with target sum $x = 8$:

1	2	3	4	5	6	7	8
1	3	2	5	1	1	2	3

Initially, the subarray contains the elements 1, 3 and 2, and the sum of the subarray is 6. The subarray cannot be larger, because the next element 5 would make the sum larger than x.

1	2	3	4	5	6	7	8
1	3	2	5	1	1	2	3

Then, the left pointer moves one step forward. The right pointer does not move, because otherwise the sum would become too large.

1	2	3	4	5	6	7	8
1	3	2	5	1	1	2	3

Again, the left pointer moves one step forward, and this time the right pointer moves three steps forward. The sum is $2 + 5 + 1 = 8$, so we have found a subarray where the sum of the elements is x.

1	2	3	4	5	6	7	8
1	3	2	5	1	1	2	3

The time complexity of the algorithm depends on the number of steps the right pointer moves. There is no upper bound how many steps the pointer can move on a single turn. However, the pointer moves *a total of* $O(n)$ steps during the algorithm, because it only moves forward.

Since both the left and right pointer move $O(n)$ steps during the algorithm, the time complexity is $O(n)$.

2SUM problem

Another problem that can be solved using the two pointers method is the following problem, also known as the **2SUM problem**: We are given an array of n numbers and a target sum x, and our task is to find two numbers in the array such that their sum is x, or report that no such numbers exist.

To solve the problem, we first sort the numbers in the array in increasing order. After that, we iterate through the array using two pointers. The left pointer starts at the first element and moves one step forward on each turn. The right pointer begins at the last element and always moves backward until the sum of the elements is at most x. If the sum of the elements is exactly x, a solution has been found.

For example, consider the following array with target sum $x = 12$:

1	2	3	4	5	6	7	8
1	4	5	6	7	9	9	10

The initial positions of the pointers are as follows. The sum of the numbers is $1 + 10 = 11$ that is smaller than x.

1	2	3	4	5	6	7	8
1	4	5	6	7	9	9	10

Then the left pointer moves one step forward. The right pointer moves three steps backward, and the sum becomes $4 + 7 = 11$.

1	2	3	4	5	6	7	8
1	4	5	6	7	9	9	10

After this, the left pointer moves one step forward again. The right pointer does not move, and a solution $5 + 7 = 12$ has been found.

1	2	3	4	5	6	7	8
1	4	5	6	7	9	9	10

The algorithm consists of two phases: First, sorting the array takes $O(n \log n)$ time. After this, the left pointer moves $O(n)$ steps forward, and the right pointer moves $O(n)$ steps backward. Thus, the total time complexity of the algorithm is $O(n \log n)$.

Note that it is possible to solve the problem in another way in $O(n \log n)$ time using binary search. In such a solution, we iterate through the array and for each number, we try to find another number such that the sum is x. This can be done by performing n binary searches, and each search takes $O(\log n)$ time.

A more difficult problem is the **3SUM problem** that asks to find *three* numbers in the array such that their sum is x. Using the idea of the above algorithm, this problem can be solved in $O(n^2)$ time[1]. Can you see how?

---

[1] For a long time, it was thought that solving the 3SUM problem more efficiently than in $O(n^2)$ time would not be possible. However, in 2014, it turned out [21] that this is not the case.

8.2 Nearest smaller elements

Amortized analysis is often used for estimating the number of operations performed on a data structure. The operations may be distributed unevenly so that most operations occur during a certain phase of the algorithm, but the total number of the operations is limited.

As an example, consider the problem of finding for each element of an array the **nearest smaller element**, i.e., the first smaller element that precedes the element in the array. It is possible that no such element exists, in which case the algorithm should report this. It turns out that the problem can be solved in $O(n)$ time using an appropriate data structure.
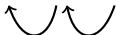
An efficient solution to the problem is to iterate through the array from left to right, and maintain a chain of elements where the first element is the current element and each following element is the nearest smaller element of the previous element. If the chain only contains one element, the current element does not have the nearest smaller element. At each step, elements are removed from the chain until the first element is smaller than the current element, or the chain is empty. After this, the current element is added to the chain.

As an example, consider the following array:

1	2	3	4	5	6	7	8
1	3	4	2	5	3	4	2

First, the numbers 1, 3 and 4 are added to the chain, because each number is larger than the previous number. Thus, the nearest smaller element of 4 is 3, and the nearest smaller element of 3 is 1.

1	2	3	4	5	6	7	8
1	3	4	2	5	3	4	2

The next number 2 is smaller than the two first numbers in the chain. Thus, the numbers 4 and 3 are removed from the chain, and then the number 2 is added to the chain. Its nearest smaller element is 1:

1	2	3	4	5	6	7	8
1	3	4	2	5	3	4	2

After this, the number 5 is larger than the number 2, so it will be added to the chain, and its nearest smaller element is 2:

1	2	3	4	5	6	7	8
1	3	4	2	5	3	4	2

The algorithm continues in the same way and finds the nearest smaller element for each number in the array. But how efficient is the algorithm?

The efficiency of the algorithm depends on the total time used for manipulating the chain. If an element is larger than the first element of the chain, it is directly added to the chain, which is efficient. However, sometimes the chain can contain several larger elements and it takes time to remove them. Still, each element is added exactly once to the chain and removed at most once from the chain. Thus, each element causes $O(1)$ chain operations, and the total time complexity of the algorithm is $O(n)$.

8.3   Sliding window minimum

A **sliding window** is a constant-size subarray that moves through the array. At each position of the window, we want to calculate some information about the elements inside the window. An interesting problem is to maintain the **sliding window minimum**, which means that at each position of the window, we should report the smallest element inside the window.
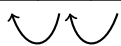
The sliding window minimum can be calculated using the same idea that we used for calculating the nearest smaller elements. We maintain a chain whose first element is always the last element in the window, and each element in the chain is smaller than the previous element. The last element in the chain is always the smallest element inside the window. When the sliding window moves forward and a new element appears, we remove from the chain all elements that are larger than the new element. After this, we add the new element to the chain. Finally, if the last element in the chain does not belong to the window anymore, it is removed from the chain.

As an example, consider the following array:

1	2	3	4	5	6	7	8
2	1	4	5	3	4	1	2

Suppose that the size of the sliding window is 4. At the first window position, the smallest element is 1:

1	2	3	4	5	6	7	8
2	1	4	5	3	4	1	2

Then the window moves one step forward. The new number 3 is smaller than the numbers 5 and 4 in the chain, so the numbers 5 and 4 are removed from the chain and the number 3 is added to the chain. The smallest element is still 1.

1	2	3	4	5	6	7	8
2	1	4	5	3	4	1	2

After this, the window moves again, and the smallest element 1 does not belong to the window anymore. Thus, it is removed from the chain and the smallest element is now 3. In addition, the new number 4 is added to the chain.

1	2	3	4	5	6	7	8
2	1	4	5	3	4	1	2
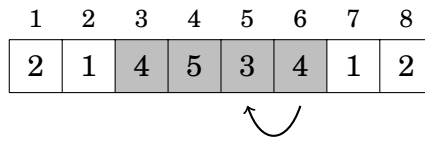
The next new element 1 is smaller than all elements in the chain. Thus, all elements are removed from the chain and it will only contain the element 1:

1	2	3	4	5	6	7	8
2	1	4	5	3	4	1	2
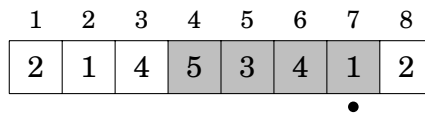
Finally the window reaches its last position. The number 2 is added to the chain, but the smallest element inside the window is still 1.

1	2	3	4	5	6	7	8
2	1	4	5	3	4	1	2
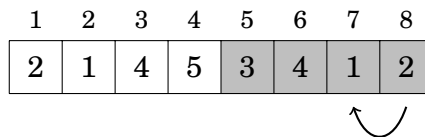
Also in this algorithm, each element in the array is added to the chain exactly once and removed from the chain at most once. Thus, the total time complexity of the algorithm is $O(n)$.

Chapter 9

Range queries

In this chapter, we discuss data structures that allow us to efficiently answer range queries. In a **range query**, we are given two indices to an array, and our task is to calculate some value based on the elements between the given indices. Typical range queries are:

- **sum query**: calculate the sum of elements

- **minimum query**: find the smallest element

- **maximum query**: find the largest element

For example, consider the range $[4,7]$ in the following array:

1	2	3	4	5	6	7	8
1	3	8	4	6	1	3	4

In this range, the sum of elements is $4+6+1+3=16$, the minimum element is 1 and the maximum element is 6.

A simple way to process range queries is to go through all elements in the range. For example, the following function sum calculates the sum of elements in a range $[a,b]$ of an array t:

```
int sum(int a, int b) {
    int s = 0;
    for (int i = a; i <= b; i++) {
        s += t[i];
    }
    return s;
}
```

The above function works in $O(n)$ time, where n is the number of elements in the array. Thus, we can process q queries in $O(nq)$ time using the function. However, if both n and q are large, this approach is slow, and it turns out that there are ways to process range queries much more efficiently.

9.1 Static array queries

We first focus on a situation where the array is **static**, i.e., the elements are never modified between the queries. In this case, it suffices to construct a static data structure that tells us the answer for any possible query.

Sum queries

It turns out that we can easily process sum queries on a static array, because we can construct a data structure called a **sum array**. Each element in a sum array corresponds to the sum of elements in the original array up to that position.

For example, consider the following array:

1	2	3	4	5	6	7	8
1	3	4	8	6	1	4	2

The corresponding sum array is as follows:

1	2	3	4	5	6	7	8
1	4	8	16	22	23	27	29

Let $\text{sum}(a,b)$ denote the sum of elements in the range $[a,b]$. Since the sum array contains all values of the form $\text{sum}(1,k)$, we can calculate any value of $\text{sum}(a,b)$ in $O(1)$ time, because

$$\text{sum}(a,b) = \text{sum}(1,b) - \text{sum}(1,a-1).$$

By defining $\text{sum}(1,0) = 0$, the above formula also holds when $a = 1$.

For example, consider the range $[4,7]$:

1	2	3	4	5	6	7	8
1	3	4	8	6	1	4	2

The sum in the range is $8+6+1+4 = 19$. This sum can be calculated using two values in the sum array:

1	2	3	4	5	6	7	8
1	4	8	16	22	23	27	29

Thus, the sum in the range $[4,7]$ is $27-8 = 19$.

It is also possible to generalize this idea to higher dimensions. For example, we can construct a two-dimensional sum array that can be used for calculating the sum of any rectangular subarray in $O(1)$ time. Each value in such an array is the sum of a subarray that begins at the upper-left corner of the array.

The following picture illustrates the idea:
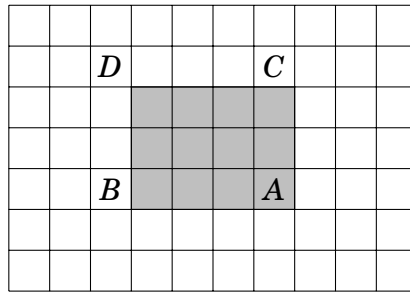
The sum of the gray subarray can be calculated using the formula

$$S(A) - S(B) - S(C) + S(D),$$

where $S(X)$ denotes the sum of a rectangular subarray from the upper-left corner to the position of X.

Minimum queries

Next we will see how we can process range minimum queries in $O(1)$ time after an $O(n \log n)$ time preprocessing[1]. Note that minimum and maximum queries can always be processed using similar techniques, so it suffices to focus on minimum queries.

Let $\text{rmq}(a, b)$ ("range minimum query") denote the minimum element in the range $[a, b]$. The idea is to precalculate all values of $\text{rmq}(a, b)$ where $b - a + 1$, the length of the range, is a power of two. For example, for the array

1	2	3	4	5	6	7	8
1	3	4	8	6	1	4	2

the following values will be calculated:

a	b	$\text{rmq}(a,b)$	a	b	$\text{rmq}(a,b)$	a	b	$\text{rmq}(a,b)$
1	1	1	1	2	1	1	4	1
2	2	3	2	3	3	2	5	3
3	3	4	3	4	4	3	6	1
4	4	8	4	5	6	4	7	1
5	5	6	5	6	1	5	8	1
6	6	1	6	7	1	1	8	1
7	7	4	7	8	2			
8	8	2						

The number of precalculated values is $O(n \log n)$, because there are $O(\log n)$ range lengths that are powers of two. In addition, the values can be calculated efficiently using the recursive formula

$$\text{rmq}(a, b) = \min(\text{rmq}(a, a + w - 1), \text{rmq}(a + w, b)),$$

_____

[1]There are also sophisticated techniques [14] where the preprocessing time is only $O(n)$, but such algorithms are not needed in competitive programming.

85

where $b-a+1$ is a power of two and $w = (b-a+1)/2$. Calculating all those values takes $O(n \log n)$ time.

After this, any value of $\text{rmq}(a,b)$ can be calculated in $O(1)$ time as a minimum of two precalculated values. Let k be the largest power of two that does not exceed $b-a+1$. We can calculate the value of $\text{rmq}(a,b)$ using the formula

$$\text{rmq}(a,b) = \min(\text{rmq}(a, a+k-1), \text{rmq}(b-k+1, b)).$$

In the above formula, the range $[a,b]$ is represented as the union of the ranges $[a, a+k-1]$ and $[b-k+1, b]$, both of length k.

As an example, consider the range $[2,7]$:

1	2	3	4	5	6	7	8
1	3	4	8	6	1	4	2

The length of the range is 6, and the largest power of two that does not exceed 6 is 4. Thus the range $[2,7]$ is the union of the ranges $[2,5]$ and $[4,7]$:

1	2	3	4	5	6	7	8
1	3	4	8	6	1	4	2

1	2	3	4	5	6	7	8
1	3	4	8	6	1	4	2

Since $\text{rmq}(2,5) = 3$ and $\text{rmq}(4,7) = 1$, we can conclude that $\text{rmq}(2,7) = 1$.

9.2 Binary indexed trees

A **binary indexed tree** or **Fenwick tree** [13] can be seen as a dynamic variant of a sum array. This data structure supports two $O(\log n)$ time operations: calculating the sum of elements in a range and modifying the value of an element.

The advantage of a binary indexed tree is that it allows us to efficiently update the array elements between the sum queries. This would not be possible using a sum array, because after each update, we should build the whole sum array again in $O(n)$ time.

Structure

A binary indexed tree can be represented as an array where the value at position x equals the sum of elements in the range $[x-k+1, x]$, where k is the largest power of two that divides x. For example, if $x = 6$, then $k = 2$, because 2 divides 6 but 4 does not divide 6.

For example, consider the following array:

1	2	3	4	5	6	7	8
1	3	4	8	6	1	4	2

The corresponding binary indexed tree is as follows:

1	2	3	4	5	6	7	8
1	4	4	16	6	7	4	29

For example, the value at position 6 in the binary indexed tree is 7, because the sum of elements in the range $[5,6]$ of the array is $6+1=7$.

The following picture shows more clearly how each value in the binary indexed tree corresponds to a range in the array:
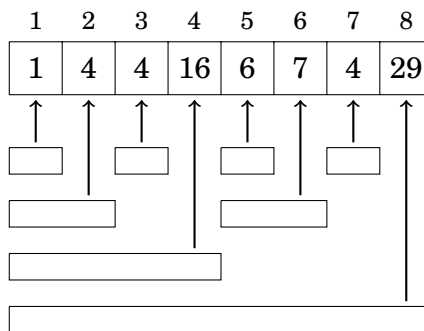
Sum queries

The values in the binary indexed tree can be used to efficiently calculate the sum of elements in any range $[1,k]$, because such a range can be divided into $O(\log n)$ ranges whose sums are available in the binary indexed tree.

For example, the range $[1,7]$ corresponds to the following values:
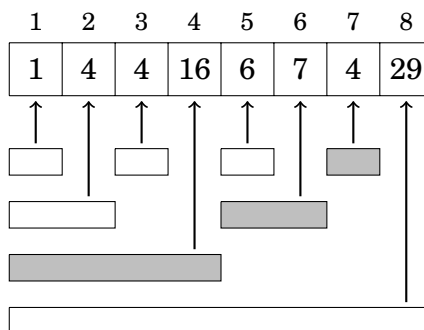
Hence, the sum of elements in the range $[1,7]$ is $16+7+4=27$.

To calculate the sum of elements in any range $[a,b]$, we can use the same trick that we used with sum arrays:

$$\text{sum}(a,b) = \text{sum}(1,b) - \text{sum}(1,a-1).$$

Also in this case, only $O(\log n)$ values are needed.

Array updates

When a value in the array changes, several values in the binary indexed tree should be updated. For example, if the element at position 3 changes, the sums of the following ranges change:
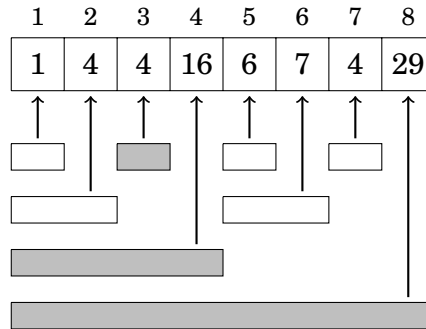
Since each array element belongs to $O(\log n)$ ranges in the binary indexed tree, it suffices to update $O(\log n)$ values.

Implementation

The operations of a binary indexed tree can be implemented in an elegant and efficient way using bit operations. The key fact needed is that $k \& -k$ isolates the last one bit of a number k. For example, $26 \& -26 = 2$ because the number 26 corresponds to 11010 and the number 2 corresponds to 10.

It turns out that when processing a sum query, the position k in the binary indexed tree needs to be decreased by $k \& -k$ at every step, and when updating the array, the position k needs to be increased by $k \& -k$ at every step.

Suppose that the binary indexed tree is stored in an array b. The following function calculates the sum of elements in a range $[1, k]$:

```
int sum(int k) {
    int s = 0;
    while (k >= 1) {
        s += b[k];
        k -= k&-k;
    }
    return s;
}
```

The following function increases the value of the element at position k by x (x can be positive or negative):

```
void add(int k, int x) {
    while (k <= n) {
        b[k] += x;
        k += k&-k;
    }
}
```

The time complexity of both the functions is $O(\log n)$, because the functions access $O(\log n)$ values in the binary indexed tree, and each move to the next position takes $O(1)$ time using bit operations.

9.3 Segment trees

A **segment tree** is a data structure that supports two operations: processing a range query and modifying an element in the array. Segment trees can support sum queries, minimum and maximum queries and many other queries so that both operations work in $O(\log n)$ time.

Compared to a binary indexed tree, the advantage of a segment tree is that it is a more general data structure. While binary indexed trees only support sum queries, segment trees also support other queries. On the other hand, a segment tree requires more memory and is a bit more difficult to implement.

Structure

A segment tree is a binary tree that contains $2n - 1$ nodes. The nodes on the bottom level of the tree correspond to the array elements, and the other nodes contain information needed for processing range queries.

Throughout the section, we assume that the size of the array is a power of two and zero-based indexing is used, because it is convenient to build a segment tree for such an array. If the size of the array is not a power of two, we can always append extra elements to it.

We will first discuss segment trees that support sum queries. As an example, consider the following array:

0	1	2	3	4	5	6	7
5	8	6	3	2	7	2	6

The corresponding segment tree is as follows:
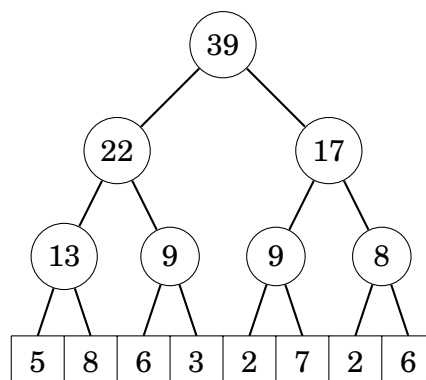
Each internal node in the segment tree contains information about a range of size 2^k in the original array. In the above tree, the value of each internal node is the sum of the corresponding array elements, and it can be calculated as the sum of the values of its left and right child node.

Range queries

The sum of elements in a given range can be calculated as a sum of values in the segment tree. For example, consider the following range:

5	8	6	3	2	7	2	6

The sum of elements in the range is $6+3+2+7+2+6 = 26$. The following two nodes in the tree correspond to the range:
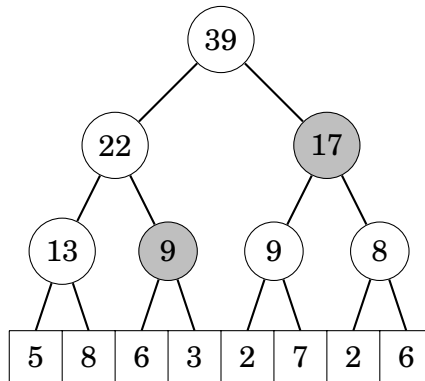
Thus, the sum of elements in the range is $9+17 = 26$.

When the sum is calculated using nodes that are located as high as possible in the tree, at most two nodes on each level of the tree are needed. Hence, the total number of nodes is only $O(\log n)$.

Array updates

When an element in the array changes, we should update all nodes in the tree whose value depends on the element. This can be done by traversing the path from the element to the top node and updating the nodes along the path.

The following picture shows which nodes in the segment tree change if the element 7 in the array changes.
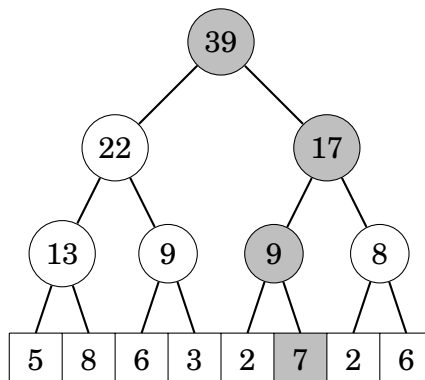
The path from bottom to top always consists of $O(\log n)$ nodes, so each update changes $O(\log n)$ nodes in the tree.

Storing the tree

A segment tree can be stored in an array of $2N$ elements where N is a power of two. Such a tree corresponds to an array indexed from 0 to $N-1$.

In the segment tree array, the element at position 1 corresponds to the top node of the tree, the elements at positions 2 and 3 correspond to the second level of the tree, and so on. Finally, the elements at positions $N \ldots 2N - 1$ correspond to the bottom level of the tree, i.e., the elements of the original array.

For example, the segment tree
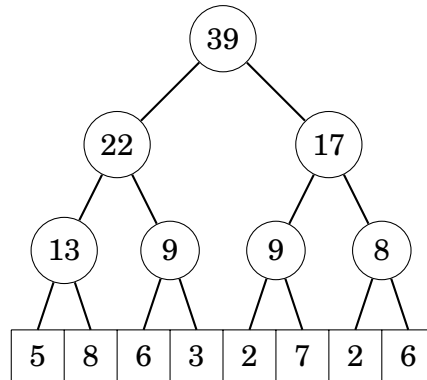
can be stored as follows ($N = 8$):

1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
39	22	17	13	9	9	8	5	8	6	3	2	7	2	6

Using this representation, for a node at position k,

- the parent node is at position $\lfloor k/2 \rfloor$,

- the left child node is at position $2k$, and

- the right child node is at position $2k + 1$.

Functions

Assume that the segment tree is stored in an array p. The following function calculates the sum of elements in a range $[a, b]$:

```
int sum(int a, int b) {
    a += N; b += N;
    int s = 0;
    while (a <= b) {
        if (a%2 == 1) s += p[a++];
        if (b%2 == 0) s += p[b--];
        a /= 2; b /= 2;
    }
    return s;
}
```

The function starts at the bottom of the tree and moves one level up at each step. Initially, the range $[a+N, b+N]$ corresponds to the range $[a, b]$ in the original array. At each step, the function adds the value of the left and right node to the sum if their parent nodes do not belong to the range. This process continues, until the sum of the range has been calculated.

The following function increases the value of the element at position k by x:

```
void add(int k, int x) {
    k += N;
    p[k] += x;
    for (k /= 2; k >= 1; k /= 2) {
        p[k] = p[2*k]+p[2*k+1];
    }
}
```

First the function updates the element at the bottom level of the tree. After this, the function updates the values of all internal nodes in the tree, until it reaches the top node of the tree.

Both above functions work in $O(\log n)$ time, because a segment tree of n elements consists of $O(\log n)$ levels, and the operations move one level forward in the tree at each step.

Other queries

Segment trees can support any queries as long as we can divide a range into two parts, calculate the answer for both parts and then efficiently combine the answers. Examples of such queries are minimum and maximum, greatest common divisor, and bit operations and, or and xor.

For example, the following segment tree supports minimum queries:
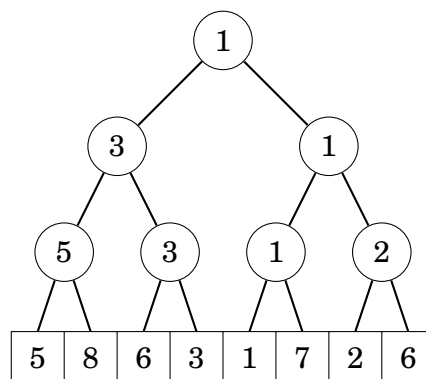
In this segment tree, every node in the tree contains the smallest element in the corresponding range of the array. The top node of the tree contains the smallest element in the whole array. The operations can be implemented like previously, but instead of sums, minima are calculated.

Binary search in a tree

The structure of the segment tree allows us to use binary search for finding elements in the array. For example, if the tree supports minimum queries, we can find the position of the smallest element in $O(\log n)$ time.

For example, in the following tree the smallest element 1 can be found by traversing a path downwards from the top node:
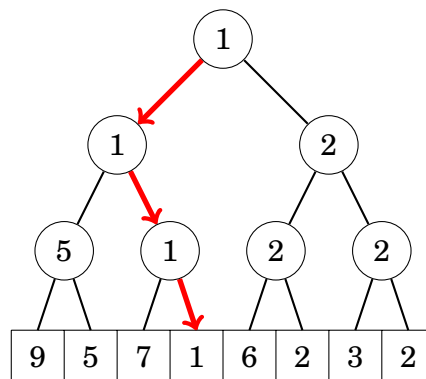
9.4 Additional techniques

Index compression

A limitation in data structures that are built upon an array is that the elements are indexed using consecutive integers. Difficulties arise when large indices are needed. For example, if we wish to use the index 10^9, the array should contain 10^9 elements which would require too much memory.

However, we can often bypass this limitation by using **index compression**, where the original indices are replaced with indices $1, 2, 3$, etc. This can be done if we know all the indices needed during the algorithm beforehand.

The idea is to replace each original index x with $p(x)$ where p is a function that compresses the indices. We require that the order of the indices does not change, so if $a < b$, then $p(a) < p(b)$. This allows us to conviently perform queries even if the indices are compressed.

For example, if the original indices are 555, 10^9 and 8, the new indices are:

$$
\begin{aligned}
p(8) &= 1 \\
p(555) &= 2 \\
p(10^9) &= 3
\end{aligned}
$$

Range updates

So far, we have implemented data structures that support range queries and updates of single values. Let us now consider an opposite situation, where we should update ranges and retrieve single values. We focus on an operation that increases all elements in a range $[a, b]$ by x.

Surprisingly, we can use the data structures presented in this chapter also in this situation. To do this, we build an **inverse sum array** for the array. The idea is that the original array is the sum array of the inverse sum array. For example, consider the following array:

1	2	3	4	5	6	7	8
3	3	1	1	1	5	2	2

The inverse sum array for the above array is as follows:

1	2	3	4	5	6	7	8
3	0	−2	0	0	4	−3	0

For example, the value 5 at position 6 in the original array corresponds to the sum $3 - 2 + 4 = 5$.

The advantage of the inverse sum array is that we can update a range in the original array by changing just two elements in the inverse sum array. For example, if we want to increase the elements in the range $2 \ldots 5$ by 5, it suffices to increase the value at position 2 by 5 and decrease the value at position 6 by 5. The result is as follows:

1	2	3	4	5	6	7	8
3	5	−2	0	0	−1	−3	0

More generally, to increase the elements in a range $[a, b]$ by x, we increase the value at position a by x and decrease the value at position $b + 1$ by x. Thus, it is only needed to update single values and process sum queries, so we can use a binary indexed tree or a segment tree.

A more difficult problem is to support both range queries and range updates. In Chapter 28 we will see that even this is possible.

Chapter 10

Bit manipulation

All data in computer programs is internally stored as bits, i.e., as numbers 0 and 1. In this chapter, we will learn how integers are represented as bits, and how bit operations can be used to manipulate them. It turns out that there are many uses for bit operations in algorithm programming.

10.1   Bit representation

Every nonnegative integer can be represented as a sum

$$c_k 2^k + \ldots + c_2 2^2 + c_1 2^1 + c_0 2^0,$$

where each coefficient c_i is either 0 or 1. The bit representation of such a number is $c_k \cdots c_2 c_1 c_0$. For example, the number 43 corresponds to the sum

$$1 \cdot 2^5 + 0 \cdot 2^4 + 1 \cdot 2^3 + 0 \cdot 2^2 + 1 \cdot 2^1 + 1 \cdot 2^0,$$

so the bit representation of the number is 101011.

In programming, the length of the bit representation depends on the data type of the number. For example, in C++ the type int is usually a 32-bit type and an int number consists of 32 bits. Thus, the bit representation of 43 as an int number is as follows:

$$00000000000000000000000000101011$$

The bit representation of a number is either **signed** or **unsigned**. Usually a signed representation is used, which means that both negative and positive numbers can be represented. A signed number of n bits can contain any integer between 2^{n-1} and $2^{n-1} - 1$. For example, the int type in C++ is a signed type, and it can contain any integer between 2^{31} and $2^{31} - 1$.

The first bit in a signed representation is the sign of the number (0 for nonnegative numbers and 1 for negative numbers), and the remaining $n - 1$ bits contain the magnitude of the number. **Two's complement** is used, which means that the opposite number of a number is calculated by first inverting all the bits in the number, and then increasing the number by one.

For example, the bit representation of -43 as an int number is as follows:

$$11111111111111111111111111010101$$

In a signed representation, only nonnegative numbers can be used, but the upper bound of the numbers is larger. A signed number of n bits can contain any integer between 0 and $2^n - 1$. For example, the unsigned int type in C++ can contain any integer between 0 and $2^{32} - 1$.

There is a connection between signed and unsigned representations: a number $-x$ in a signed representation equals the number $2^n - x$ in an unsigned representation. For example, the following code shows that the signed number $x = -43$ equals the unsigned number $y = 2^{32} - 43$:

```
int x = -43;
unsigned int y = x;
cout << x << "\n"; // -43
cout << y << "\n"; // 4294967253
```

If a number is larger than the upper bound of the bit representation, the number will overflow. In a signed representation, the next number after $2^{n-1} - 1$ is -2^{n-1}, and in an unsigned representation, the next number after 2^{n-1} is 0. For example, consider the following code:

```
int x = 2147483647
cout << x << "\n"; // 2147483647
x++;
cout << x << "\n"; // -2147483648
```

Initially, the value of x is $2^{31} - 1$. This is the largest number that can be stored in an int variable, so the next number after $2^{31} - 1$ is -2^{31}.

10.2   Bit operations

And operation

The **and** operation x & y produces a number that has one bits in positions where both x and y have one bits. For example, 22 & 26 = 18, because

$$
\begin{array}{r r l}
 & 10110 & (22) \\
\& & 11010 & (26) \\
\hline
= & 10010 & (18) \\
\end{array}
$$

Using the and operation, we can check if a number x is even because x & 1 = 0 if x is even, and x & 1 = 1 if x is odd. More generally, x is divisible by 2^k exactly when x & $(2^k - 1) = 0$.

Or operation

The **or** operation $x \mid y$ produces a number that has one bits in positions where at least one of x and y have one bits. For example, 22 | 26 = 30, because

$$
\begin{array}{r r l}
 & 10110 & (22) \\
\mid & 11010 & (26) \\
\hline
= & 11110 & (30)
\end{array}
$$

Xor operation

The **xor** operation $x \wedge y$ produces a number that has one bits in positions where exactly one of x and y have one bits. For example, 22 ^ 26 = 12, because

$$
\begin{array}{r r l}
 & 10110 & (22) \\
\wedge & 11010 & (26) \\
\hline
= & 01100 & (12)
\end{array}
$$

Not operation

The **not** operation $\sim x$ produces a number where all the bits of x have been inverted. The formula $\sim x = -x - 1$ holds, for example, $\sim 29 = -30$.

The result of the not operation at the bit level depends on the length of the bit representation, because the operation changes all bits. For example, if the numbers are 32-bit `int` numbers, the result is as follows:

$$
\begin{array}{r c r l}
x & = & 29 & 00000000000000000000000000011101 \\
\sim x & = & -30 & 11111111111111111111111111100010
\end{array}
$$

Bit shifts

The left bit shift $x << k$ appends k zero bits to the number, and the right bit shift $x >> k$ removes the k last bits from the number. For example, $14 << 2 = 56$, because 14 equals 1110 and 56 equals 111000. Similarly, $49 >> 3 = 6$, because 49 equals 110001 and 6 equals 110.

Note that $x << k$ corresponds to multiplying x by 2^k, and $x >> k$ corresponds to dividing x by 2^k rounded down to an integer.

Applications

A number of the form $1 << k$ has a one bit in position k and all other bits are zero, so we can use such numbers to access single bits of numbers. For example, the kth bit of a number is one exactly when $x \,\&\, (1 << k)$ is not zero. The following code prints the bit representation of an `int` number x:

```
for (int i = 31; i >= 0; i--) {
    if (x&(1<<i)) cout << "1";
    else cout << "0";
}
```

It is also possible to modify single bits of numbers using the above idea. For example, the expression $x \mid (1 << k)$ sets the kth bit of x to one, the expression x & ~$(1 << k)$ sets the kth bit of x to zero, and the expression x ^ $(1 << k)$ inverts the kth bit of x.

The formula x & $(x-1)$ sets the last one bit of x to zero, and the formula x & $-x$ sets all the one bits to zero, except for the last one bit. The formula $x \mid (x-1)$ inverts all the bits after the last one bit. Also note that a positive number x is of the form 2^k if x & $(x-1) = 0$.

Additional functions

The g++ compiler provides the following functions for counting bits:

- `__builtin_clz(x)`: the number of zeros at the beginning of the number

- `__builtin_ctz(x)`: the number of zeros at the end of the number

- `__builtin_popcount(x)`: the number of ones in the number

- `__builtin_parity(x)`: the parity (even or odd) of the number of ones

The functions can be used as follows:

```
int x = 5328; // 00000000000000000001010011010000
cout << __builtin_clz(x) << "\n"; // 19
cout << __builtin_ctz(x) << "\n"; // 4
cout << __builtin_popcount(x) << "\n"; // 5
cout << __builtin_parity(x) << "\n"; // 1
```

The above functions support `int` numbers, but there are also `long long` functions available with the suffix `ll`.

10.3 Representing sets

Each subset of a set $\{0, 1, 2, \ldots, n-1\}$ corresponds to an n bit number where the one bits indicate which elements are included in the subset. For example, the set $\{1, 3, 4, 8\}$ corresponds to the number $2^8 + 2^4 + 2^3 + 2^1 = 282$, whose bit representation is 100011010.

The benefit in using the bit representation is that the information whether an element belongs to the set requires only one bit of memory. In addition, set operations can be efficiently implemented as bit operations.

Set implementation

In the following code, x contains a subset of $\{0, 1, 2, \ldots, 31\}$. The code adds the elements 1, 3, 4 and 8 to the set and then prints the elements.

```
// x is an empty set
int x = 0;
// add elements 1, 3, 4 and 8 to the set
x |= (1<<1);
x |= (1<<3);
x |= (1<<4);
x |= (1<<8);
// print the elements in the set
for (int i = 0; i < 32; i++) {
    if (x&(1<<i)) cout << i << " ";
}
cout << "\n";
```

The output of the code is as follows:

```
1 3 4 8
```

Set operations

Set operations can be implemented as follows:

- a & b is the intersection $a \cap b$ of a and b

- a | b is the union $a \cup b$ of a and b

- ~a is the complement $\bar{a}$ of a

- a & (~b) is the difference $a \setminus b$ of a and b

For example, the following code constructs the union of $\{1, 3, 4, 8\}$ and $\{3, 6, 8, 9\}$:

```
// set {1,3,4,8}
int x = (1<<1)+(1<<3)+(1<<4)+(1<<8);
// set {3,6,8,9}
int y = (1<<3)+(1<<6)+(1<<8)+(1<<9);
// union of the sets
int z = x|y;
// print the elements in the union
for (int i = 0; i < 32; i++) {
    if (z&(1<<i)) cout << i << " ";
}
cout << "\n";
```

The output of the code is as follows:

```
1 3 4 6 8 9
```

Iterating through subsets

The following code goes through the subsets of $\{0, 1, \ldots, n-1\}$:

```
for (int b = 0; b < (1<<n); b++) {
    // process subset b
}
```

The following code goes through the subsets with exactly k elements:

```
for (int b = 0; b < (1<<n); b++) {
    if (__builtin_popcount(b) == k) {
        // process subset b
    }
}
```

The following code goes through the subsets of a set x:

```
int b = 0;
do {
    // process subset b
} while (b=(b-x)&x);
```

10.4 Dynamic programming

From permutations to subsets

Using dynamic programming, it is often possible to change an iteration over permutations into an iteration over subsets, so that the dynamic programming state contains a subset of a set and possibly some additional information.

The benefit in this is that $n!$, the number of permutations of an n element set, is much larger than 2^n, the number of subsets of the same set. For example, if $n = 20$, then $n! \approx 2.4 \cdot 10^{18}$ and $2^n \approx 10^6$. Hence, for certain values of n, we can efficiently go through subsets but not through permutations.

As an example, consider the problem of calculating the number of permutations of a set $\{0, 1, \ldots, n-1\}$, where the difference between any two consecutive elements is larger than one. For example, when $n = 4$, there are two such permutations: $(1, 3, 0, 2)$ and $(2, 0, 3, 1)$.

Let $f(x, k)$ denote the number of valid permutations of a subset x where the last element is k and the difference between any two consecutive elements is larger than one. For example, $f(\{0, 1, 3\}, 1) = 1$, because there is a permutation $(0, 3, 1)$, and $f(\{0, 1, 3\}, 3) = 0$, because 0 and 1 cannot be next to each other.

Using f, the answer to the problem equals

$$\sum_{i=0}^{n-1} f(\{0, 1, \ldots, n-1\}, i),$$

because the permutation has to contain all elements $\{0, 1, \ldots, n-1\}$ and the last element can be any element.

The dynamic programming values can be stored as follows:

```
int d[1<<n][n];
```

First, $f(\{k\}, k) = 1$ for all values of k:

```
for (int i = 0; i < n; i++) d[1<<i][i] = 1;
```

Then, the other values can be calculated as follows:

```
for (int b = 0; b < (1<<n); b++) {
    for (int i = 0; i < n; i++) {
        for (int j = 0; j < n; j++) {
            if (abs(i-j) > 1 && (b&(1<<i)) && (b&(1<<j))) {
                d[b][i] += d[b^(1<<i)][j];
            }
        }
    }
}
```

In the above code, the variable b goes through all subsets and each permutation is of the form $(\ldots, j, i)$, where the difference between i and j is larger than one and i and j belong to b.

Finally, the number of solutions can be calculated as follows:

```
int s = 0;
for (int i = 0; i < n; i++) {
    s += d[(1<<n)-1][i];
}
```

Counting subsets

Our last problem in this chapter is as follows: We are given a collection C that consists of m sets, and our task is to determine for each set the number of sets in C that are its subsets. For example, consider the following collection:

$$C = \{\{0\}, \{0, 2\}, \{1, 4\}, \{0, 1, 4\}, \{1, 4, 5\}\}$$

For any set x in C, let $f(x)$ denote the number of sets (including x) in C that are subsets of x. For example, $f(\{0, 1, 4\}) = 3$, because the sets $\{0\}$, $\{1, 4\}$ and $\{0, 1, 4\}$ are subsets of $\{0, 1, 4\}$. Using this notation, our task is to calculate the value of $f(x)$ for every set x in the collection.

We will assume that each set is a subset of $\{0, 1, \ldots, n-1\}$. Thus, the collection can contain at most 2^n sets. A straightforward way to solve the problem is to go through all pairs of sets in the collection. However, a more efficient solution is possible using dynamic programming.

Let $c(x, k)$ denote the number of sets in C that equal a set x if we are allowed to remove any subset of $\{0, 1, \ldots, k\}$ from x. For example, in the above collection, $c(\{0, 1, 4\}, 1) = 2$, where the corresponding sets are $\{1, 4\}$ and $\{0, 1, 4\}$.

It turns out that we can calculate all values of $c(x, k)$ in $O(2^n n)$ time. This solves our problem, because

$$f(x) = c(x, n - 1).$$

The base cases for the function are:

$$c(x, -1) = \begin{cases} 0 & \text{if } x \notin C \\ 1 & \text{if } x \in C \end{cases}$$

For larger values of k, the following recursion holds:

$$c(x, k) = \begin{cases} c(x, k - 1) & \text{if } k \notin x \\ c(x, k - 1) + c(x \setminus \{k\}, k - 1) & \text{if } k \in x \end{cases}$$

We can conveniently implement the algorithm by representing the sets using bits. Assume that there is an array

```
int d[1<<n];
```

that is initialized so that $d[x] = 1$ if x belongs to C and otherwise $d[x] = 0$. We can now implement the algorithm as follows:

```
for (int k = 0; k < n; k++) {
    for (int b = 0; b < (1<<n); b++) {
        if (b&(1<<k)) d[b] += d[b^(1<<k)];
    }
}
```

The above code is based on the recursive definition of c. As a special trick, the code only uses the array d to calculate all values of the function. Finally, for each set x in C, $f(x) = d[x]$.

Part II

Graph algorithms

Chapter 11

Basics of graphs

Many programming problems can be solved by modeling the problem as a graph problem and using an appropriate graph algorithm. A typical example of a graph is a network of roads and cities in a country. Sometimes, though, the graph is hidden in the problem and it may be difficult to detect it.

This part of the book discusses graph algorithms, especially focusing on topics that are important in competitive programming. In this chapter, we go through concepts related to graphs, and study different ways to represent graphs in algorithms.

11.1   Graph terminology

A **graph** consists of **nodes** and **edges** between them. In this book, the variable n denotes the number of nodes in a graph, and the variable m denotes the number of edges. The nodes are numbered using integers $1, 2, \ldots, n$.

For example, the following graph consists of 5 nodes and 7 edges:
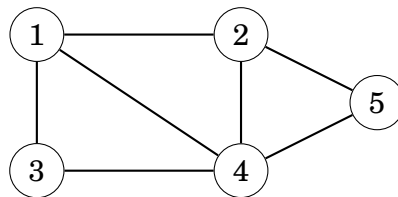
A **path** leads from node a to node b through edges of the graph. The **length** of a path is the number of edges in it. For example, the above graph contains the path $1 \rightarrow 3 \rightarrow 4 \rightarrow 5$ from node 1 to node 5:
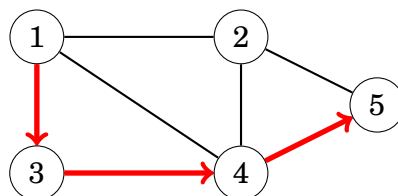
A path is a **cycle** if the first and last node is the same. For example, the above graph contains the cycle $1 \rightarrow 3 \rightarrow 4 \rightarrow 1$. A path is **simple** if each node appears at most once in the path.

Connectivity

A graph is **connected** if there is path between any two nodes. For example, the following graph is connected:
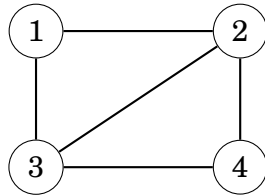
The following graph is not connected, because it is not possible to get from node 4 to any other node:
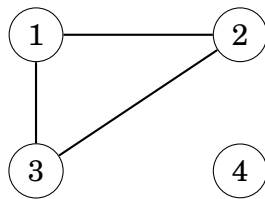
The connected parts of a graph are called its **components**. For example, the following graph contains three components: $\{1, 2, 3\}$, $\{4, 5, 6, 7\}$ and $\{8\}$.
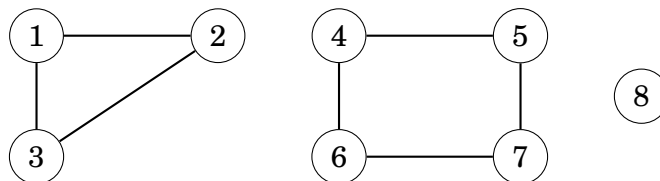
A **tree** is a connected graph that consists of n nodes and $n-1$ edges. There is a unique path between any two nodes in a tree. For example, the following graph is a tree:
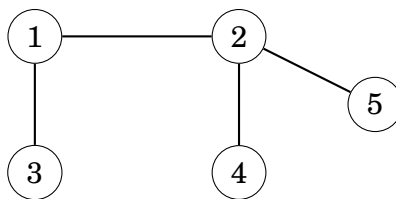
Edge directions

A graph is **directed** if the edges can be traversed in one direction only. For example, the following graph is directed:
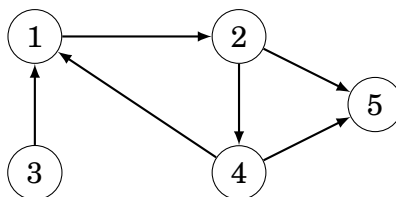
The above graph contains the path $3 \rightarrow 1 \rightarrow 2 \rightarrow 5$ from node 3 to node 5, but there is no path from node 5 to node 3.

Edge weights

In a **weighted** graph, each edge is assigned a **weight**. Often the weights are interpreted as edge lengths. For example, the following graph is weighted:
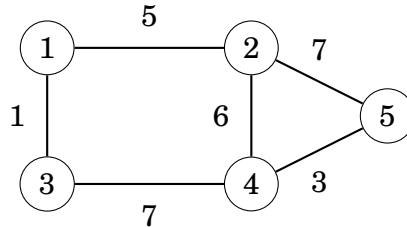
The length of a path in a weighted graph is the sum of edge weights on the path. For example, in the above graph, the length of the path $1 \rightarrow 2 \rightarrow 5$ is 12 and the length of the path $1 \rightarrow 3 \rightarrow 4 \rightarrow 5$ is 11. The latter path is the **shortest** path from node 1 to node 5.

Neighbors and degrees

Two nodes are **neighbors** or **adjacent** if there is an edge between them. The **degree** of a node is the number of its neighbors. For example, in the following graph, the neighbors of node 2 are 1, 4 and 5, so its degree is 3.
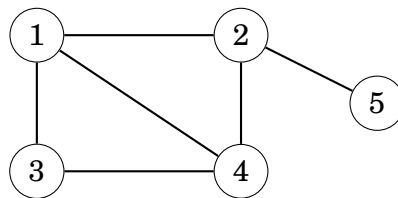
The sum of degrees in a graph is always $2m$, where m is the number of edges, because each edge increases the degree of exactly two nodes by one. For this reason, the sum of degrees is always even.

A graph is **regular** if the degree of every node is a constant d. A graph is **complete** if the degree of every node is $n-1$, i.e., the graph contains all possible edges between the nodes.

In a directed graph, the **indegree** of a node is the number of edges that end at the node, and the **outdegree** of a node is the number of edges that start at the node. For example, in the following graph, the indegree of node 2 is 2 and the outdegree of the node is 1.
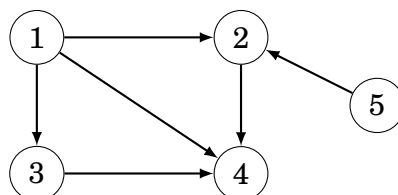
Colorings

In a **coloring** of a graph, each node is assigned a color so that no adjacent nodes have the same color.

A graph is **bipartite** if it is possible to color it using two colors. It turns out that a graph is bipartite exactly when it does not contain a cycle with an odd number of edges. For example, the graph
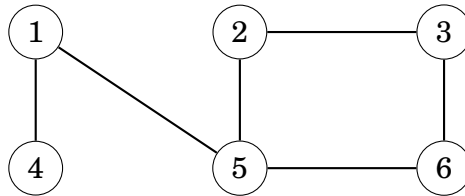
is bipartite, because it can be colored as follows:
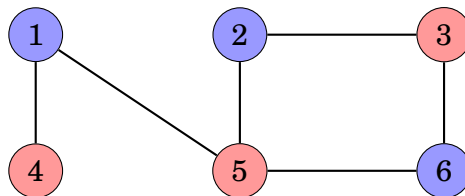
However, the graph
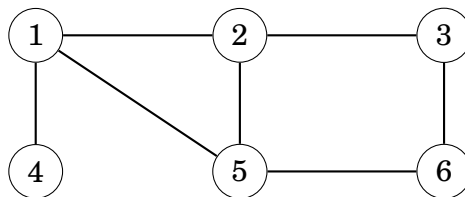
is not bipartite, because it is not possible to color the following cycle of three nodes using two colors:
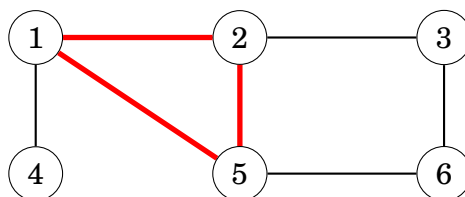
Simplicity

A graph is **simple** if no edge starts and ends at the same node, and there are no multiple edges between two nodes. Often we assume that graphs are simple. For example, the following graph is *not* simple:
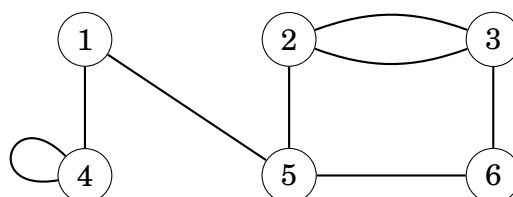
11.2 Graph representation

There are several ways to represent graphs in algorithms. The choice of a data structure depends on the size of the graph and the way the algorithm processes it. Next we will go through three common representations.

Adjacency list representation

In the adjacency list representation, each node x in the graph is assigned an **adjacency list** that consists of nodes to which there is an edge from x. Adjacency lists are the most popular way to represent graphs, and most algorithms can be efficiently implemented using them.

A convenient way to store the adjacency lists is to declare an array of vectors as follows:

```
vector<int> v[N];
```

The constant N is chosen so that all adjacency lists can be stored. For example, the graph
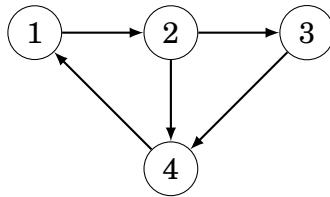
can be stored as follows:

```
v[1].push_back(2);
v[2].push_back(3);
v[2].push_back(4);
v[3].push_back(4);
v[4].push_back(1);
```

If the graph is undirected, it can be stored in a similar way, but each edge is added in both directions.

For a weighted graph, the structure can be extended as follows:

```
vector<pair<int,int>> v[N];
```

If there is an edge from node a to node b with weight w, the adjacency list of node a contains the pair (b,w). For example, the graph
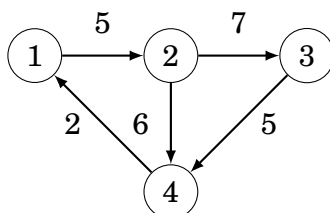
can be stored as follows:

```
v[1].push_back({2,5});
v[2].push_back({3,7});
v[2].push_back({4,6});
v[3].push_back({4,5});
v[4].push_back({1,2});
```

The benefit in using adjacency lists is that we can efficiently find the nodes to which we can move from a given node through an edge. For example, the following loop goes through all nodes to which we can move from node s:

```
for (auto u : v[s]) {
    // process node u
}
```

Adjacency matrix representation

An **adjacency matrix** is a two-dimensional array that indicates which edges the graph contains. We can efficiently check from an adjacency matrix if there is an edge between two nodes. The matrix can be stored as an array

```
int v[N][N];
```

where each value $v[a][b]$ indicates whether the graph contains an edge from node a to node b. If the edge is included in the graph, then $v[a][b] = 1$, and otherwise $v[a][b] = 0$. For example, the graph
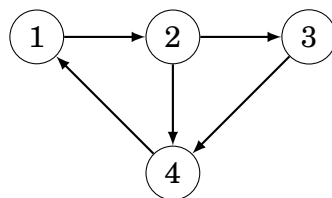
can be represented as follows:

	1	2	3	4
1	0	1	0	0
2	0	0	1	1
3	0	0	0	1
4	1	0	0	0

If the graph is directed, the adjacency matrix representation can be extended so that the matrix contains the weight of the edge if the edge exists. Using this representation, the graph
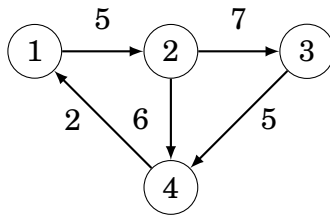
corresponds to the following matrix:

	1	2	3	4
1	0	5	0	0
2	0	0	7	6
3	0	0	0	5
4	2	0	0	0

The drawback in the adjacency matrix representation is that there are n^2 elements in the matrix and usually most of them are zero. For this reason, the representation cannot be used if the graph is large.

Edge list representation

An **edge list** contains all edges of a graph in some order. This is a convenient way to represent a graph if the algorithm processes all edges of the graph and it is not needed to find edges that start at a given node.

The edge list can be stored in a vector

```
vector<pair<int,int>> v;
```

where each pair (a, b) denotes that there is an edge from node a to node b. Thus, the graph
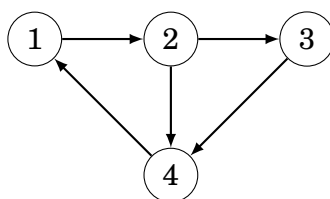
can be represented as follows:

```
v.push_back({1,2});
v.push_back({2,3});
v.push_back({2,4});
v.push_back({3,4});
v.push_back({4,1});
```

If the graph is weighted, the structure can be extended as follows:

```
vector<tuple<int,int,int>> v;
```

Each element in this list is of the form (a, b, w), which means that there is an edge from node a to node b with weight w. For example, the graph
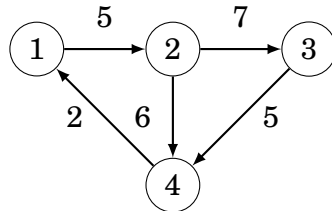
can be represented as follows:

```
v.push_back(make_tuple(1,2,5));
v.push_back(make_tuple(2,3,7));
v.push_back(make_tuple(2,4,6));
v.push_back(make_tuple(3,4,5));
v.push_back(make_tuple(4,1,2));
```

Chapter 12

Graph traversal

This chapter discusses two fundamental graph algorithms: depth-first search and breadth-first search. Both algorithms are given a starting node in the graph, and they visit all nodes that can be reached from the starting node. The difference in the algorithms is the order in which they visit the nodes.

12.1 Depth-first search

Depth-first search (DFS) is a straightforward graph traversal technique. The algorithm begins at a starting node, and proceeds to all other nodes that are reachable from the starting node using the edges in the graph.

Depth-first search always follows a single path in the graph as long as it finds new nodes. After this, it returns to previous nodes and begins to explore other parts of the graph. The algorithm keeps track of visited nodes, so that it processes each node only once.

Example

Let us consider how depth-first search processes the following graph:
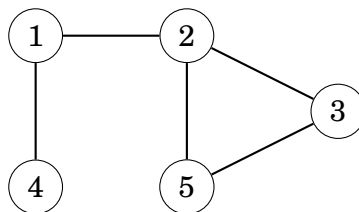
We may begin the search at any node in the graph, but we will now begin the search at node 1.

The search first proceeds to node 2:
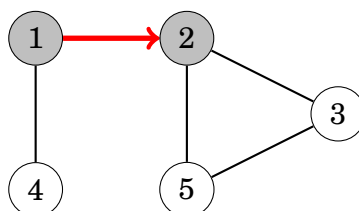
After this, nodes 3 and 5 will be visited:
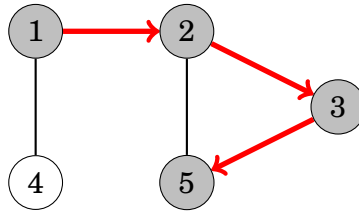
The neighbors of node 5 are 2 and 3, but the search has already visited both of them, so it is time to return to previous nodes. Also the neighbors of nodes 3 and 2 have been visited, so we next move from node 1 to node 4:
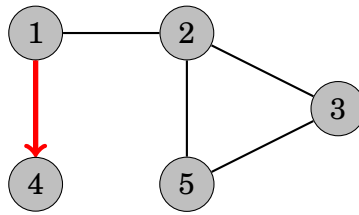
After this, the search terminates because it has visited all nodes.

The time complexity of depth-first search is $O(n + m)$ where n is the number of nodes and m is the number of edges, because the algorithm processes each node and edge once.

Implementation

Depth-first search can be conveniently implemented using recursion. The following function dfs begins a depth-first search at a given node. The function assumes that the graph is stored as adjacency lists in an array

```
vector<int> v[N];
```

and also maintains an array

```
int z[N];
```

that keeps track of the visited nodes. Initially, each array value is 0, and when the search arrives at node s, the value of $z[s]$ becomes 1. The function can be implemented as follows:

```
void dfs(int s) {
    if (z[s]) return;
    z[s] = 1;
    // process node s
    for (auto u: v[s]) {
        dfs(u);
    }
}
```

12.2 Breadth-first search

Breadth-first search (BFS) visits the nodes in increasing order of their distance from the starting node. Thus, we can calculate the distance from the starting node to all other nodes using breadth-first search. However, breadth-first search is more difficult to implement than depth-first search.

Breadth-first search goes through the nodes one level after another. First the search explores the nodes whose distance from the starting node is 1, then the nodes whose distance is 2, and so on. This process continues until all nodes have been visited.

Example

Let us consider how the algorithm processes the following graph:
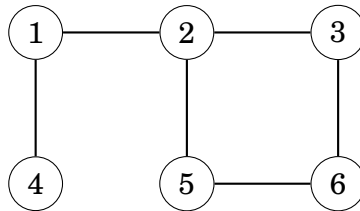
Suppose again that the search begins at node 1. First, we process all nodes that can be reached from node 1 using a single edge:
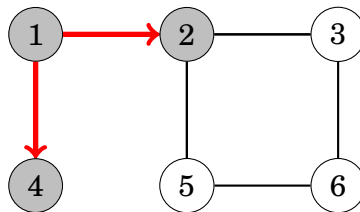
After this, we proceed to nodes 3 and 5:
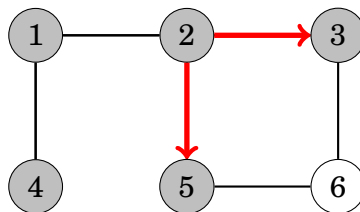
Finally, we visit node 6:
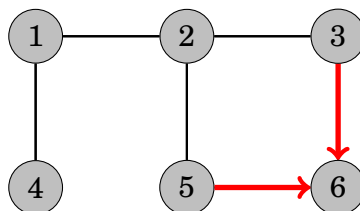
Now we have calculated the distances from the starting node to all nodes in the graph. The distances are as follows:

node	distance
1	0
2	1
3	2
4	1
5	2
6	3

Like in depth-first search, the time complexity of breadth-first search is $O(n+m)$ where n is the number of nodes and m is the number of edges.

Implementation

Breadth-first search is more difficult to implement than depth-first search, because the algorithm visits nodes in different parts of the graph. A typical implementation is based on a queue that contains nodes. At each step, the next node in the queue will be processed.

The following code begins a breadth-first search at node x. The code assumes that the graph is stored as adjacency lists and maintains a queue

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

that contains the nodes in increasing order of their distance. New nodes are always added to the end of the queue, and the node at the beginning of the queue is the next node to be processed.

In addition, the code uses arrays

```
int z[N], e[N];
```

so that the array z indicates which nodes the search has already visited and the array e will contain the distances to all nodes in the graph. The search can be implemented as follows:

```
z[s] = 1; e[x] = 0;
q.push(x);
while (!q.empty()) {
    int s = q.front(); q.pop();
    // process node s
    for (auto u : v[s]) {
        if (z[u]) continue;
        z[u] = 1; e[u] = e[s]+1;
        q.push(u);
    }
}
```

12.3 Applications

Using the graph traversal algorithms, we can check many properties of the graph. Usually, either depth-first search or bredth-first search can be used, but in practice, depth-first search is a better choice, because it is easier to implement. In the following applications we will assume that the graph is undirected.

Connectivity check

A graph is connected if there is a path between any two nodes in the graph. Thus, we can check if a graph is connected by choosing an arbitrary node and finding out if we can reach all other nodes.

For example, in the graph
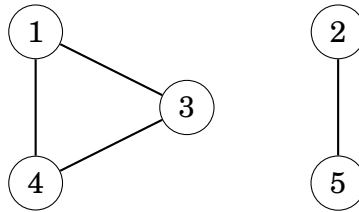
a depth-first search from node 1 visits the following nodes:
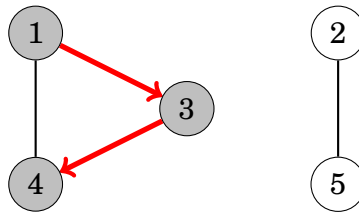
Since the search did not visit all the nodes, we can conclude that the graph is not connected. In a similar way, we can also find all connected components of a graph by iterating through the nodes and always starting a new depth-first search if the current node does not belong to any component yet.

Finding cycles

A graph contains a cycle if during a graph traversal, we find a node whose neighbor (other than the previous node in the current path) has already been visited. For example, the graph
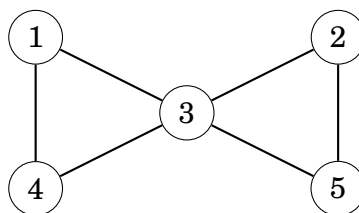
contains two cycles and we can find one of them as follows:
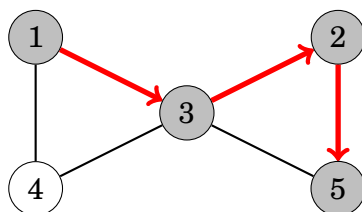
When we move from node 2 to node 5 it turns out that the neighbor 3 has already been visited. Thus, the graph contains a cycle that goes through node 3, for example, $3 \rightarrow 2 \rightarrow 5 \rightarrow 3$.

Another way to find out whether a graph contains a cycle is to simply calculate the number of nodes and edges in every component. If a component contains c nodes and no cycle, it must contain exactly $c-1$ edges (so it has to be a tree). If there are c or more edges, the component surely contains a cycle.

Bipartiteness check

A graph is bipartite if its nodes can be colored using two colors so that there are no adjacent nodes with the same color. It is surprisingly easy to check if a graph is bipartite using graph traversal algorithms.

The idea is to color the starting node blue, all its neighbors red, all their neighbors blue, and so on. If at some point of the search we notice that two adjacent nodes have the same color, this means that the graph is not bipartite. Otherwise the graph is bipartite and one coloring has been found.

For example, the graph
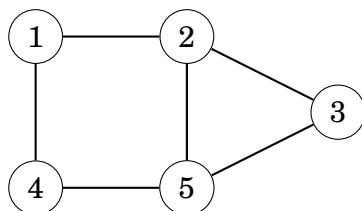
is not bipartite, because a search from node 1 proceeds as follows:
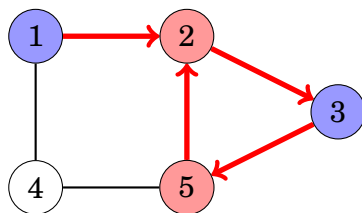
We notice that the color or both nodes 2 and 5 is red, while they are adjacent nodes in the graph. Thus, the graph is not bipartite.

This algorithm always works, because when there are only two colors available, the color of the starting node in a component determines the colors of all other nodes in the component. It does not make any difference whether the starting node is red or blue.

Note that in the general case, it is difficult to find out if the nodes in a graph can be colored using k colors so that no adjacent nodes have the same color. Even when $k = 3$, no efficient algorithm is known but the problem is NP-hard [19].

Chapter 13

Shortest paths

Finding the shortest path between two nodes of a graph is an important problem that has many applications in practice. For example, a natural problem in a road network is to calculate the length of the shortest route between two cities, given the lengths of the roads.

In an unweighted graph, the length of a path equals the number of edges in the path and we can simply use breadth-first search to find the shortest path. However, in this chapter we concentrate on weighted graphs where more sophisticated algorithms are needed for finding shortest paths.

13.1   Bellman–Ford algorithm

The **Bellman–Ford algorithm** [5] finds the shortest paths from a starting node to all other nodes in the graph. The algorithm can process all kinds of graphs, provided that the graph does not contain a cycle with negative length. If the graph contains a negative cycle, the algorithm can detect this.

The algorithm keeps track of distances from the starting node to other nodes. Initially, the distance to the starting node is 0 and the distance to all other nodes in infinite. The algorithm reduces the distances by finding edges that shorten the paths until it is not possible to reduce any distance.

Example

Let us consider how the Bellman–Ford algorithm works in the following graph:
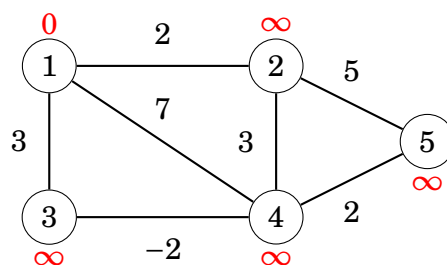
Each node in the graph is assigned a distance. Initially, the distance to the starting node is 0, and the distance to all other nodes is infinite.

The algorithm searches for edges that reduce distances. First, all edges from node 1 reduce distances:
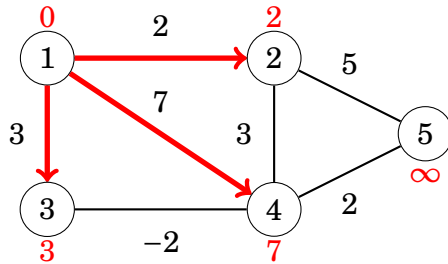
After this, edges $2 \rightarrow 5$ and $3 \rightarrow 4$ reduce distances:
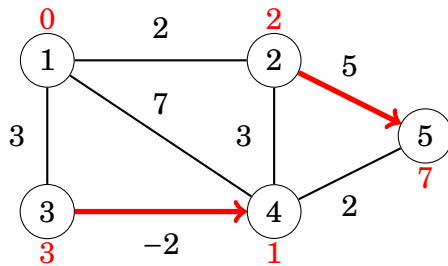
Finally, there is one more change:
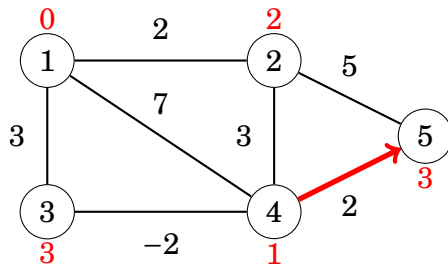
After this, no edge can reduce any distance. This means that the distances are final and we have successfully calculated the shortest distance from the starting node to all other nodes.

For example, the shortest distance 3 from node 1 to node 5 corresponds to the following path:
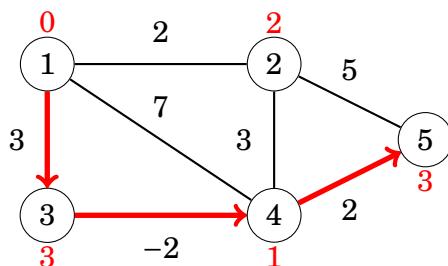
Implementation

The following implementation of the Bellman–Ford algorithm finds the shortest distances from a node x to all other nodes in the graph. The code assumes that the graph is stored as adjacency lists in an array

```
vector<pair<int,int>> v[N];
```

as pairs of the form (x,w): there is an edge to node x with weight w.

The algorithm consists of $n-1$ rounds, and on each round the algorithm goes through all edges of the graph and tries to reduce the distances. The algorithm constructs an array e that will contain the distance from x to all nodes in the graph. The initial value 10^9 means infinity.

```
for (int i = 1; i <= n; i++) e[i] = 1e9;
e[x] = 0;
for (int i = 1; i <= n-1; i++) {
    for (int a = 1; a <= n; a++) {
        for (auto b : v[a]) {
            e[b.first] = min(e[b.first],e[a]+b.second);
        }
    }
}
```

The time complexity of the algorithm is $O(nm)$, because the algorithm consists of $n-1$ rounds and iterates through all m edges during a round. If there are no negative cycles in the graph, all distances are final after $n-1$ rounds, because each shortest path can contain at most $n-1$ edges.

In practice, the final distances can usually be found faster than in $n-1$ rounds. Thus, a possible way to make the algorithm more efficient is to stop the algorithm if no distance can be reduced during a round.

Negative cycle

The Bellman–Ford algorithm can be also used to check if the graph contains a cycle with negative length. For example, the graph
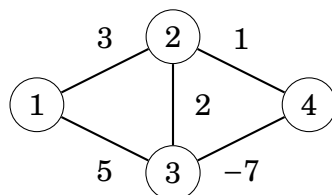
contains a negative cycle $2 \rightarrow 3 \rightarrow 4 \rightarrow 2$ with length -4.

If the graph contains a negative cycle, we can shorten a path that contains the cycle infinitely many times by repeating the cycle again and again. Thus, the concept of a shortest path is not meaningful in this situation.

A negative cycle can be detected using the Bellman–Ford algorithm by running the algorithm for n rounds. If the last round reduces any distance, the graph

contains a negative cycle. Note that this algorithm can be used to search for a negative cycle in the whole graph regardless of the starting node.

SPFA algorithm

The **SPFA algorithm** ("Shortest Path Faster Algorithm") [12] is a variant of the Bellman–Ford algorithm, that is often more efficient than the original algorithm. The SPFA algorithm does not go through all the edges on each round, but instead, it chooses the edges to be examined in a more intelligent way.

The algorithm maintains a queue of nodes that might be used for reducing the distances. First, the algorithm adds the starting node x to the queue. Then, the algorithm always processes the first node in the queue, and when an edge $a \rightarrow b$ reduces a distance, node b is added to the queue.

The following implementation uses a queue q. In addition, an array z indicates if a node is already in the queue, in which case the algorithm does not add the node to the queue again.

```
for (int i = 1; i <= n; i++) e[i] = 1e9;
e[x] = 0;
q.push(x);
while (!q.empty()) {
    int a = q.front(); q.pop();
    z[a] = 0;
    for (auto b : v[a]) {
        if (e[a]+b.second < e[b.first]) {
            e[b.first] = e[a]+b.second;
            if (!z[b]) {q.push(b); z[b] = 1;}
        }
    }
}
```

The efficiency of the SPFA algorithm depends on the structure of the graph: the algorithm is often efficient, but its worst case time complexity is still $O(nm)$ and it is possible to create inputs that make the algorithm as slow as the original Bellman–Ford algorithm.

13.2   Dijkstra's algorithm

Dijkstra's algorithm [9] finds the shortest paths from the starting node to all other nodes, like the Bellman–Ford algorithm. The benefit in Dijsktra's algorithm is that it is more efficient and can be used for processing large graphs. However, the algorithm requires that there are no negative weight edges in the graph.

Like the Bellman–Ford algorithm, Dijkstra's algorithm maintains distances to the nodes and reduces them during the search. Dijkstra's algorithm is efficient, because it only processes each edge in the graph once, using the fact that there are no negative edges.

Example

Let us consider how Dijkstra's algorithm works in the following graph when the
starting node is node 1:
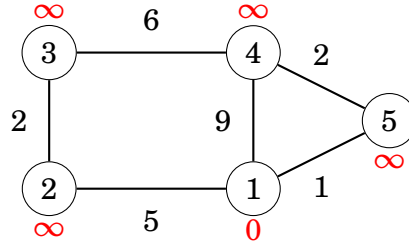
Like in the Bellman–Ford algorithm, initially the distance to the starting node is
0 and the distance to all other nodes is infinite.

At each step, Dijkstra's algorithm selects a node that has not been processed
yet and whose distance is as small as possible. The first such node is node 1 with
distance 0.

When a node is selected, the algorithm goes through all edges that start at
the node and reduces the distances using them:
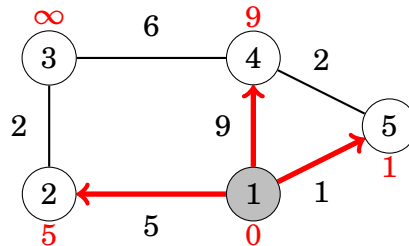
The edges from node 1 reduced distances to nodes 2, 4 and 5, whose distances are
now 5, 9 and 1.

The next node to be processed is node 5 with distance 1:
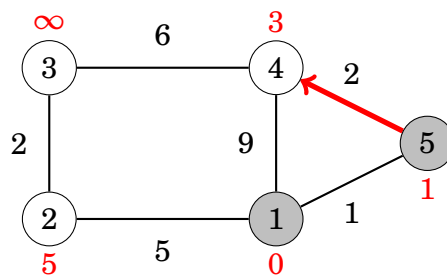
After this, the next node is node 4:

123
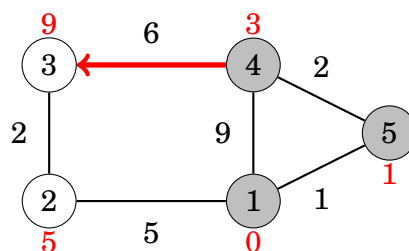
A remarkable property in Dijkstra's algorithm is that whenever a node is selected, its distance is final. For example, at this point of the algorithm, the distances 0, 1 and 3 are the final distances to nodes 1, 5 and 4.

After this, the algorithm processes the two remaining nodes, and the final distances are as follows:
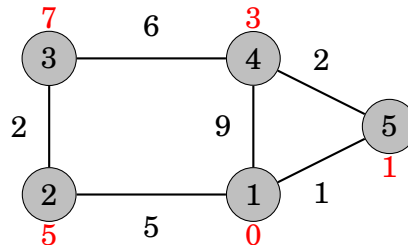
Negative edges

The efficiency of Dijkstra's algorithm is based on the fact that the graph does not contain negative edges. If there is a negative edge, the algorithm may give incorrect results. As an example, consider the following graph:
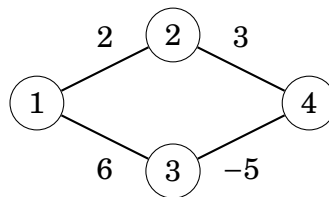
The shortest path from node 1 to node 4 is $1 \rightarrow 3 \rightarrow 4$ and its length is 1. However, Dijkstra's algorithm finds the path $1 \rightarrow 2 \rightarrow 4$ by following the minimum weight edges. The algorithm does not take into account that on the other path, the weight -5 compensates the previous large weight 6.

Implementation

The following implementation of Dijkstra's algorithm calculates the minimum distances from a node x to all other nodes. The graph is stored in an array v as adjacency lists like in the Bellman–Ford algorithm.

An efficient implementation of Dijkstra's algorithm requires that it is possible to efficiently find the minimum distance node that has not been processed. An appropriate data structure for this is a priority queue that contains the nodes ordered by their distances. Using a priority queue, the next node to be processed can be retrieved in logarithmic time.

In the following implementation, the priority queue contains pairs whose first element is the current distance to the node and second element is the identifier of the node.

```
priority_queue<pair<int,int>> q;
```

A small difficulty is that in Dijkstra's algorithm, we should find the node with the *minimum* distance, while the C++ priority queue finds the *maximum* element as default. An easy trick is to use *negative* distances, which allows us to directly use the C++ priority queue.

The code keeps track of processed nodes in an array z, and maintains the distances in an array e. Initially, the distance to the starting node is 0, and the distance to all other nodes is 10^9 (infinite).

```
for (int i = 1; i <= n; i++) e[i] = 1e9;
e[x] = 0;
q.push({0,x});
while (!q.empty()) {
    int a = q.top().second; q.pop();
    if (z[a]) continue;
    z[a] = 1;
    for (auto b : v[a]) {
        if (e[a]+b.second < e[b]) {
            e[b] = e[a]+b.second;
            q.push({-e[b],b});
        }
    }
}
```

The time complexity of the above implementation is $O(n + m \log m)$ because the algorithm goes through all nodes in the graph and adds for each edge at most one distance to the priority queue.

13.3   Floyd–Warshall algorithm

The **Floyd–Warshall algorithm** [16] is an alternative way to approach the problem of finding shortest paths. Unlike the other algorihms in this chapter, it finds all shortest paths between the nodes in a single run.

The algorithm maintains a two-dimensional array that contains distances between the nodes. First, the distances are calculated only using direct edges between the nodes. After this the algorithm reduces the distances by using intermediate nodes in the paths.

Example

Let us consider how the Floyd–Warshall algorithm works in the following graph:
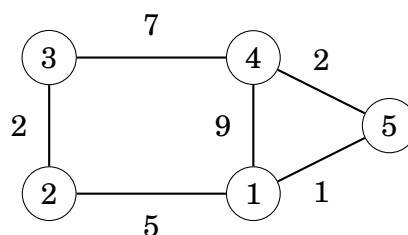
Initially, the distance from each node to itself is 0, and the distance between nodes a and b is x if there is an edge between nodes a and b with weight x. All other distances are infinite.

In this graph, the initial array is as follows:

	1	2	3	4	5
1	0	5	∞	9	1
2	5	0	2	∞	∞
3	∞	2	0	7	∞
4	9	∞	7	0	2
5	1	∞	∞	2	0

The algorithm consists of consecutive rounds. On each round, the algorithm selects a new node that can act as an intermediate node in paths from now on, and the algorithm reduces the distances in the array using this node.

On the first round, node 1 is the new intermediate node. There is a new path between nodes 2 and 4 with length 14, because node 1 connects them. There is also a new path between nodes 2 and 5 with length 6.

	1	2	3	4	5
1	0	5	∞	9	1
2	5	0	2	**14**	**6**
3	∞	2	0	7	∞
4	9	**14**	7	0	2
5	1	**6**	∞	2	0

On the second round, node 2 is the new intermediate node. This creates new paths between nodes 1 and 3 and between nodes 3 and 5:

	1	2	3	4	5
1	0	5	**7**	9	1
2	5	0	2	14	6
3	**7**	2	0	7	**8**
4	9	14	7	0	2
5	1	6	**8**	2	0

On the third round, node 3 is the new intermediate round. There is a new path between nodes 2 and 4:

	1	2	3	4	5
1	0	5	7	9	1
2	5	0	2	**9**	6
3	7	2	0	7	8
4	9	**9**	7	0	2
5	1	6	8	2	0

The algorithm continues like this, until all nodes have been appointed inter-mediate nodes. After the algorithm has finished, the array contains the minimum distances between any two nodes:

	1	2	3	4	5
1	0	5	7	3	1
2	5	0	2	9	6
3	7	2	0	7	8
4	3	9	7	0	2
5	1	6	8	2	0

For example, the array tells us that the shortest distance between nodes 2 and 4 is 8. This corresponds to the following path:
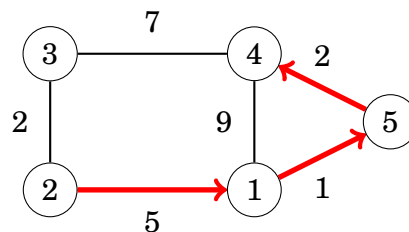
Implementation

The advantage of the Floyd–Warshall algorithm that it is easy to implement. The following code constructs a distance matrix d where d[a][b] is the shortest distance between nodes a and b. First, the algorithm initializes d using the adjacency matrix v of the graph (10^9 means infinity):

```
for (int i = 1; i <= n; i++) {
    for (int j = 1; j <= n; j++) {
        if (i == j) d[i][j] = 0;
        else if (v[i][j]) d[i][j] = v[i][j];
        else d[i][j] = 1e9;
    }
}
```

After this, the shortest distances can be found as follows:

```
for (int k = 1; k <= n; k++) {
    for (int i = 1; i <= n; i++) {
        for (int j = 1; j <= n; j++) {
            d[i][j] = min(d[i][j], d[i][k]+d[k][j]);
        }
    }
}
```

The time complexity of the algorithm is $O(n^3)$, because it contains three nested loops that go through the nodes in the graph.

Since the implementation of the Floyd–Warshall algorithm is simple, the algorithm can be a good choice even if it is only needed to find a single shortest path in the graph. However, the algorithm can only be used when the graph is so small that a cubic time complexity is fast enough.

Chapter 14

Tree algorithms

A **tree** is a connected, acyclic graph that consists of n nodes and $n-1$ edges. Removing any edge from a tree divides it into two components, and adding any edge to a tree creates a cycle. Moreover, there is always a unique path between any two nodes of a tree.

For example, the following tree consists of 7 nodes and 6 edges:
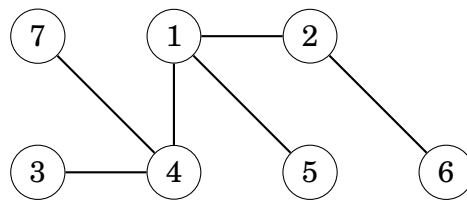
The **leaves** of a tree are the nodes with degree 1, i.e., with only one neighbor. For example, the leaves of the above tree are nodes 3, 5, 6 and 7.

In a **rooted** tree, one of the nodes is appointed the **root** of the tree, and all other nodes are placed underneath the root. For example, in the following tree, node 1 is the root of the tree.
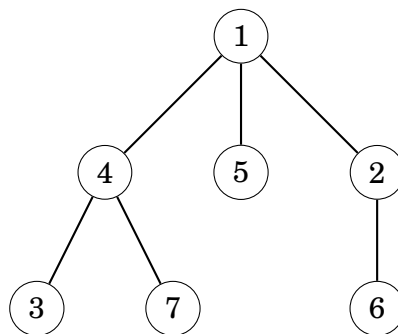
In a rooted tree, the **children** of a node are its lower neighbors, and the **parent** of a node is its upper neighbor. Each node has exactly one parent, except for the root that does not have a parent. For example, in the above tree, the children of node 4 are nodes 3 and 7, and the parent is node 1.

The structure of a rooted tree is *recursive*: each node in the tree is the root of a **subtree** that contains the node itself and all other nodes that can be reached by traversing down the tree. For example, in the above tree, the subtree of node 4 consists of nodes 4, 3 and 7.

14.1 Tree traversal

Depth-first search and breadth-first search can be used for traversing the nodes in a tree. The traversal of a tree is easier to implement than that of a general graph, because there are no cycles in the tree and it is not possible to reach a node from multiple directions.

The typical way to traverse a tree is to start a depth-first search at an arbitrary node. The following recursive function can be used:

```
void dfs(int s, int e) {
    // process node s
    for (auto u : v[s]) {
        if (u != e) dfs(u, s);
    }
}
```

The function parameters are the current node s and the previous node e. The purpose of the parameter e is to make sure that the search only moves to nodes that have not been visited yet.

The following function call starts the search at node x:

```
dfs(x, 0);
```

In the first call $e = 0$, because there is no previous node, and it is allowed to proceed to any direction in the tree.

Dynamic programming

Dynamic programming can be used to calculate some information during a tree traversal. Using dynamic programming, we can, for example, calculate in $O(n)$ time for each node in a rooted tree the number of nodes in its subtree or the length of the longest path from the node to a leaf.

As an example, let us calculate for each node s a value $c[s]$: the number of nodes in its subtree. The subtree contains the node itself and all nodes in the subtrees of its children. Thus, we can calculate the number of nodes recursively using the following code:

```
void dfs(int s, int e) {
    c[s] = 1;
    for (auto u : v[s]) {
        if (u == e) continue;
        dfs(u, s);
        c[s] += c[u];
    }
}
```

14.2 Diameter

The **diameter** of a tree is the length of the longest path between two nodes in the tree. For example, in the tree
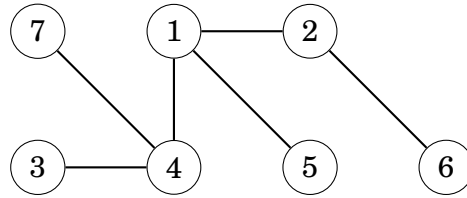
the diameter is 4, and it corresponds to two paths: the path between nodes 3 and 6, and the path between nodes 7 and 6.

Next we will learn two efficient algorithms for calculating the diameter of a tree. Both algorithms calculate the diameter in $O(n)$ time. The first algorithm is based on dynamic programming, and the second algorithm uses two depth-first searches to calculate the diameter.

Algorithm 1

First, we root the tree arbitrarily. After this, we use dynamic programming to calculate for each node x the length of the longest path that begins at some leaf, ascends to x and then descends to another leaf. The length of the longest such path equals the diameter of the tree.

In the example graph, the longest path begins at node 7, ascends to node 1, and then descends to node 6:
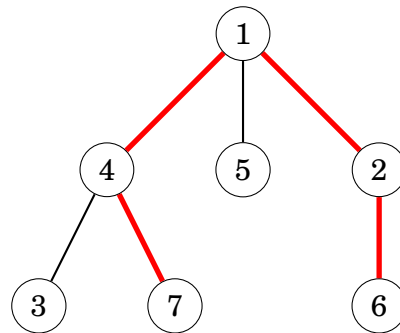
The algorithm first calculates for each node x the length of the longest path from x to a leaf. For example, in the above tree, the longest path from node 1 to a leaf has length 2 (the path can be $1 \rightarrow 4 \rightarrow 3$, $1 \rightarrow 4 \rightarrow 7$ or $1 \rightarrow 2 \rightarrow 6$). After this, the algorithm calculates for each node x the length of the longest path where x is the highest point of the path. The longest such path can be found by choosing two children with longest paths to leaves. For example, in the above graph, nodes 2 and 4 yield the longest path for node 1.

Algorithm 2

Another efficient way to calculate the diameter of a tree is based on two depth-first searches. First, we choose an arbitrary node a in the tree and find the

farthest node b from a. Then, we find the farthest node c from b. The diameter of the tree is the distance between b and c.

In the example graph, a, b and c could be:
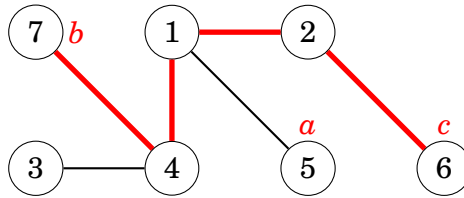
This is an elegant method, but why does it work?

It helps to draw the tree differently so that the path that corresponds to the diameter is horizontal, and all other nodes hang from it:
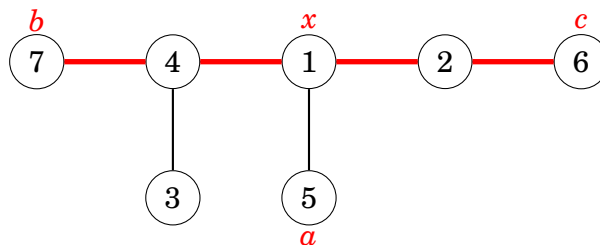
Node x indicates the place where the path from node a joins the path that corresponds to the diameter. The farthest node from a is node b, node c or some other node that is at least as far from node x. Thus, this node is always a valid choice for a starting node of a path that corresponds to the diameter.

14.3 Distances between nodes

A more difficult problem is to calculate for each node in the tree and for each direction, the maximum distance to a node in that direction. It turns out that this can be calculated in $O(n)$ time using dynamic programming.

In the example graph, the distances are as follows:
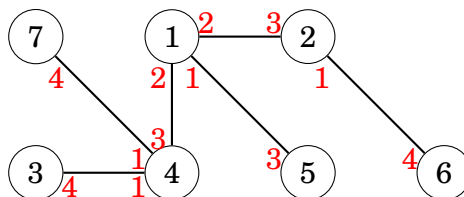
For example, the farthest node from node 4 in the direction of node 1 is node 6, and the distance to that node is 3 using the path $4 \rightarrow 1 \rightarrow 2 \rightarrow 6$.

Also in this problem, a good starting point is to root the tree. After this, all distances to leaves can be calculated using dynamic programming:
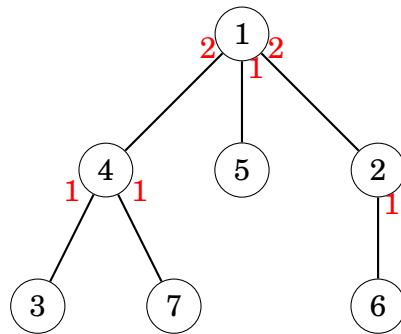
The remaining task is to calculate the distances through parents. This can be done by traversing the tree once again and keeping track of the largest distance from the parent of the current node to some other node in another direction.

For example, the distance from node 2 upwards is one larger than the distance from node 1 downwards in some other direction than node 2:
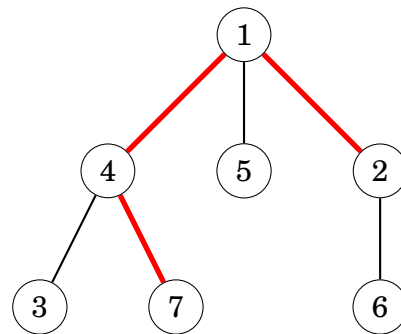
Finally, we can calculate the distances for all nodes and all directions:
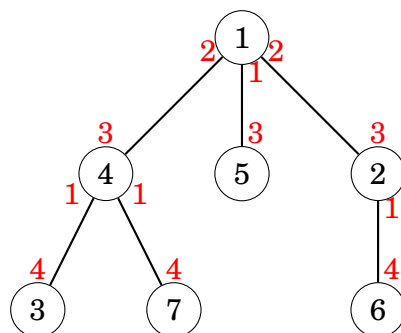
14.4 Binary trees

A **binary tree** is a rooted tree where each node has a left and right subtree. It is possible that a subtree of a node is empty. Thus, every node in a binary tree has zero, one or two children.

For example, the following tree is a binary tree:
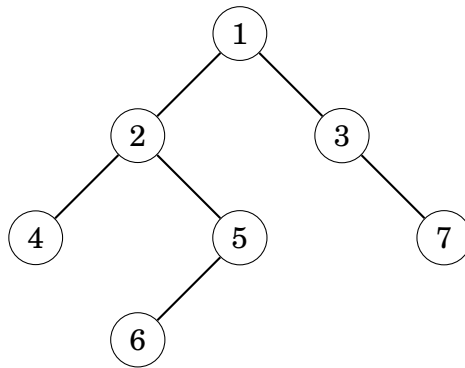
The nodes in a binary tree have three natural orderings that correspond to different ways to recursively traverse the tree:

- **pre-order**: first process the root, then traverse the left subtree, then traverse the right subtree

- **in-order**: first traverse the left subtree, then process the root, then traverse the right subtree

- **post-order**: first traverse the left subtree, then traverse the right subtree, then process the root

For the above tree, the nodes in pre-order are $[1,2,4,5,6,3,7]$, in in-order $[4,2,6,5,1,3,7]$ and in post-order $[4,6,5,2,7,3,1]$.

If we know the pre-order and in-order of a tree, we can reconstruct the exact structure of the tree. For example, the above tree is the only possible tree with pre-order $[1,2,4,5,6,3,7]$ and in-order $[4,2,6,5,1,3,7]$. In a similar way, the post-order and in-order also determine the structure of a tree.

However, the situation is different if we only know the pre-order and post-order of a tree. In this case, there may be more than one tree that match the orderings. For example, in both of the trees

the pre-order is $[1,2]$ and the post-order is $[2,1]$, but the structures of the trees are different.

Chapter 15

Spanning trees

A **spanning tree** of a graph consists of the nodes of the graph and some of the edges of the graph so that there is a path between any two nodes. Like trees in general, spanning trees are connected and acyclic. Usually there are several ways to construct a spanning tree.

For example, consider the following graph:
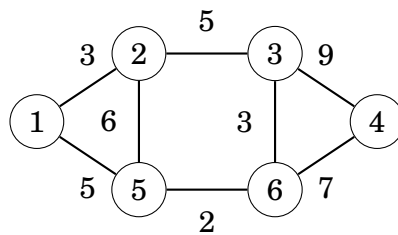
A possible spanning tree for the graph is as follows:
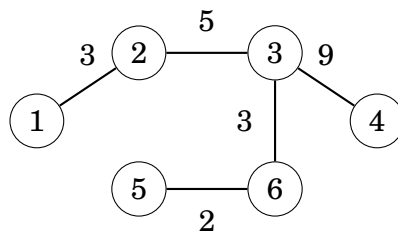
The weight of a spanning tree is the sum of the edge weights. For example, the weight of the above spanning tree is $3 + 5 + 9 + 3 + 2 = 22$.

A **minimum spanning tree** is a spanning tree whose weight is as small as possible. The weight of a minimum spanning tree for the example graph is 20, and such a tree can be constructed as follows:
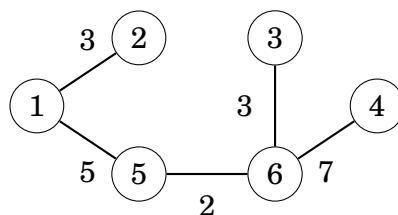
In a similar way, a **maximum spanning tree** is a spanning tree whose weight is as large as possible. The weight of a maximum spanning tree for the example graph is 32:
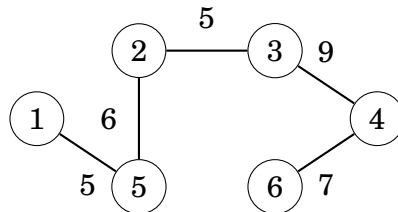
Note that there may be several minimum and maximum spanning trees for a graph, so the trees are not unique.

This chapter discusses algorithms for constructing spanning trees. It turns out that it is easy to find minimum and maximum spanning trees, because many greedy methods produce optimals solutions. We will learn two algorithms that both process the edges of the graph ordered by their weights. We will focus on finding minimum spanning trees, but similar algorithms can be used for finding maximum spanning trees by processing the edges in reverse order.

15.1   Kruskal's algorithm

In **Kruskal's algorithm** [31], the initial spanning tree only contains the nodes of the graph and does not contain any edges. Then the algorithm goes through the edges ordered by their weights, and always adds an edge to the tree if it does not create a cycle.

The algorithm maintains the components of the tree. Initially, each node of the graph belongs to a separate component. Always when an edge is added to the tree, two components are joined. Finally, all nodes belong to the same component, and a minimum spanning tree has been found.

Example

Let us consider how Kruskal's algorithm processes the following graph:
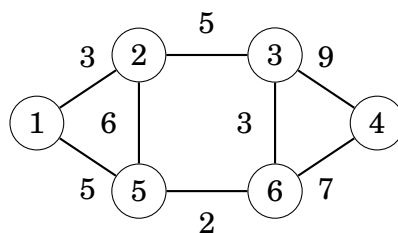
The first step in the algorithm is to sort the edges in increasing order of their weights. The result is the following list:

edge	weight
5–6	2
1–2	3
3–6	3
1–5	5
2–3	5
2–5	6
4–6	7
3–4	9

After this, the algorithm goes through the list and adds each edge to the tree if it joins two separate components.

Initially, each node is in its own component:
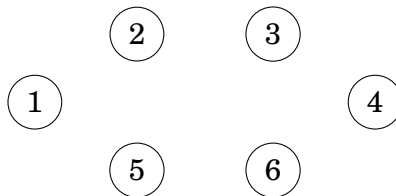
The first edge to be added to the tree is the edge 5–6 that creates the component {5,6} by joining the components {5} and {6}:
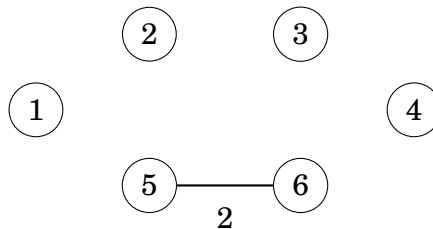
After this, the edges 1–2, 3–6 and 1–5 are added in a similar way:
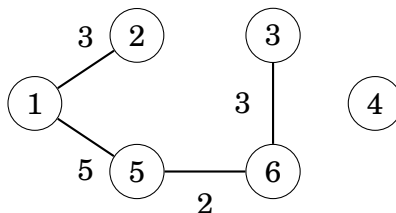
After those steps, most components have been joined and there are two components in the tree: {1,2,3,5,6} and {4}.

The next edge in the list is the edge 2–3, but it will not be included in the tree, because nodes 2 and 3 are already in the same component. For the same reason, the edge 2–5 will not be included in the tree.

Finally, the edge 4–6 will be included in the tree:
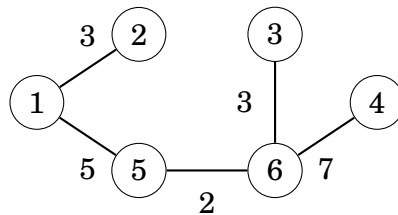
After this, the algorithm will not add any new edges, because the graph is connected and there is a path between any two nodes. The resulting graph is a minimum spanning tree with weight $2+3+3+5+7=20$.

Why does this work?

It is a good question why Kruskal's algorithm works. Why does the greedy strategy guarantee that we will find a minimum spanning tree?

Let us see what happens if the minimum weight edge of the graph is not included in the spanning tree. For example, suppose that a spanning tree for the previous graph would not contain the minimum weight edge 5–6. We do not know the exact structure of such a spanning tree, but in any case it has to contain some edges. Assume that the tree would be as follows:
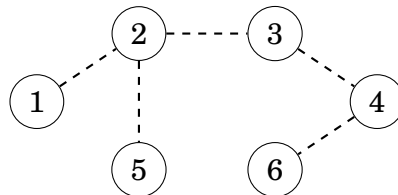
However, it is not possible that the above tree would be a minimum spanning tree for the graph. The reason for this is that we can remove an edge from the tree and replace it with the minimum weight edge 5–6. This produces a spanning tree whose weight is *smaller*:
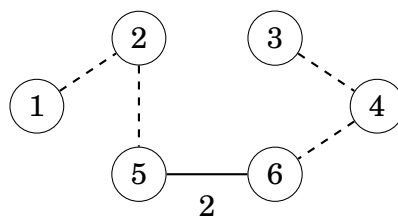
For this reason, it is always optimal to include the minimum weight edge in the tree to produce a minimum spanning tree. Using a similar argument, we can show that it is also optimal to add the next edge in weight order to the tree, and so on. Hence, Kruskal's algorithm works correctly and always produces a minimum spanning tree.

Implementation

When implementing Kruskal's algorithm, the edge list representation of the graph is convenient. The first phase of the algorithm sorts the edges in the list in $O(m \log m)$ time. After this, the second phase of the algorithm builds the minimum spanning tree as follows:

```
for (...) {
  if (!same(a,b)) union(a,b);
}
```

The loop goes through the edges in the list and always processes an edge a–b where a and b are two nodes. Two functions are needed: the function `same` determines if the nodes are in the same component, and the function `union` joins the components that contain nodes a and b.

The problem is how to efficiently implement the functions `same` and `union`. One possibility is to implement the function `same` as a graph traversal and check if we can get from node a to node b. However, the time complexity of such a function would be $O(n + m)$ and the resulting algorithm would be slow, because the function `same` will be called for each edge in the graph.

We will solve the problem using a union-find structure that implements both functions in $O(\log n)$ time. Thus, the time complexity of Kruskal's algorithm will be $O(m \log n)$ after sorting the edge list.

15.2 Union-find structure

A **union-find structure** maintains a collection of sets. The sets are disjoint, so no element belongs to more than one set. Two $O(\log n)$ time operations are supported: the `union` operation joins two sets, and the `find` operation finds the representative of the set that contains a given element.

Structure

In a union-find structure, one element in each set is the representative of the set, and there is a chain from any other element of the set to the representative. For example, assume that the sets are $\{1, 4, 7\}$, $\{5\}$ and $\{2, 3, 6, 8\}$:
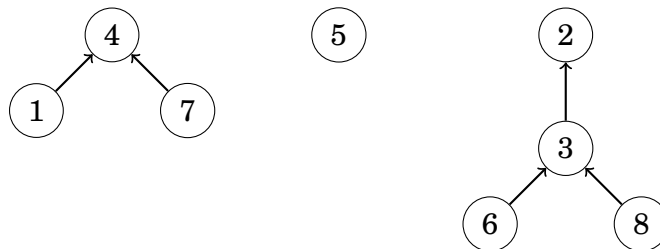
In this case the representatives of the sets are 4, 5 and 2. For each element, we can find its representative by following the chain that begins at the element. For example, the element 2 is the representative for the element 6, because we follow

the chain $6 \to 3 \to 2$. Two elements belong to the same set exactly when their representatives are the same.

Two sets can be joined by connecting the representative of one set to the representative of another set. For example, the sets $\{1, 4, 7\}$ and $\{2, 3, 6, 8\}$ can be joined as follows:
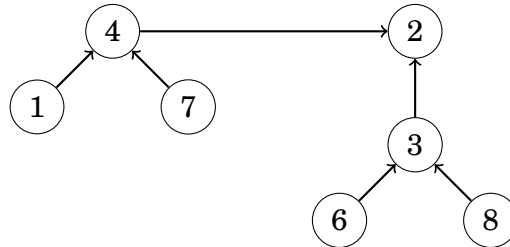
The resulting set contains the elements $\{1, 2, 3, 4, 6, 7, 8\}$. From this on, the element 2 is the representative for the entire set and the old representative 4 points to the element 2.

The efficiency of the union-find structure depends on how the sets are joined. It turns out that we can follow a simple strategy: always connect the representative of the smaller set to the representative of the larger set (or if the sets are of equal size, we can make an arbitrary choice). Using this strategy, the length of any chain will be $O(\log n)$, so we can find the representative of any element efficiently by following the corresponding chain.

Implementation

The union-find structure can be implemented using arrays. In the following implementation, the array k contains for each element the next element in the chain or the element itself if it is a representative, and the array s indicates for each representative the size of the corresponding set.

Initially, each element belongs to a separate set:

```
for (int i = 1; i <= n; i++) k[i] = i;
for (int i = 1; i <= n; i++) s[i] = 1;
```

The function find returns the representative for an element x. The representative can be found by following the chain that begins at x.

```
int find(int x) {
    while (x != k[x]) x = k[x];
    return x;
}
```

The function same checks whether elements a and b belong to the same set. This can easily be done by using the function find:

```
bool same(int a, int b) {
    return find(a) == find(b);
}
```

The function `union` joins the sets that contain elements a and b (the elements has to be in different sets). The function first finds the representatives of the sets and then connects the smaller set to the larger set.

```
void union(int a, int b) {
    a = find(a);
    b = find(b);
    if (s[a] < s[b]) swap(a,b);
    s[a] += s[b];
    k[b] = a;
}
```

The time complexity of the function `find` is $O(\log n)$ assuming that the length of each chain is $O(\log n)$. In this case, the functions `same` and `union` also work in $O(\log n)$ time. The function `union` makes sure that the length of each chain is $O(\log n)$ by connecting the smaller set to the larger set.

15.3 Prim's algorithm

Prim's algorithm [35] is an alternative method for finding a minimum spanning tree. The algorithm first adds an arbitrary node to the tree. After this, the algorithm always chooses a minimum-weight edge that adds a new node to the tree. Finally, all nodes have been added to the tree and a minimum spanning tree has been found.

Prim's algorithm resembles Dijkstra's algorithm. The difference is that Dijkstra's algorithm always selects an edge whose distance from the starting node is minimum, but Prim's algorithm simply selects the minimum weight edge that adds a new node to the tree.

Example

Let us consider how Prim's algorithm works in the following graph:
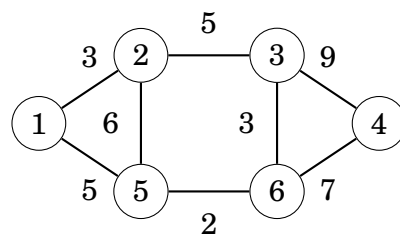
Initially, there are no edges between the nodes:
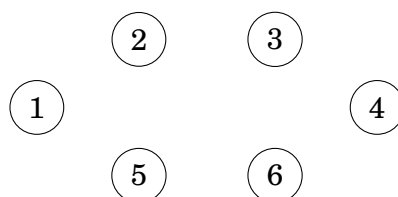
An arbitrary node can be the starting node, so let us choose node 1. First, we add node 2 that is connected by an edge of weight 3:
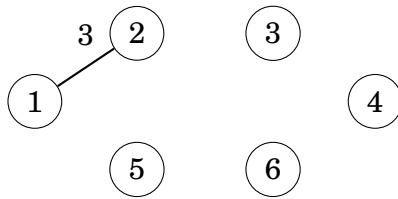
After this, there are two edges with weight 5, so we can add either node 3 or node 5 to the tree. Let us add node 3 first:
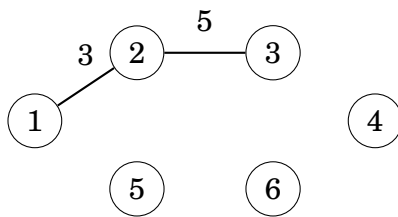
The process continues until all nodes have been included in the tree:
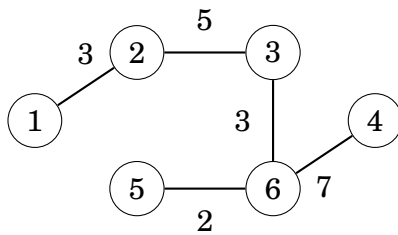
Implementation

Like Dijkstra's algorithm, Prim's algorithm can be efficiently implemented using a priority queue. The priority queue should contain all nodes that can be connected to the current component using a single edge, in increasing order of the weights of the corresponding edges.

The time complexity of Prim's algorithm is $O(n + m \log m)$ that equals the time complexity of Dijkstra's algorithm. In practice, Prim's and Kruskal's algorithms are both efficient, and the choice of the algorithm is a matter of taste. Still, most competitive programmers use Kruskal's algorithm.

Chapter 16

Directed graphs

In this chapter, we focus on two classes of directed graphs:

- **Acyclic graphs**: There are no cycles in the graph, so there is no path from any node to itself.

- **Successor graphs**: The outdegree of each node is 1, so each node has a unique successor.

It turns out that in both cases, we can design efficient algorithms that are based on the special properties of the graphs.

16.1 Topological sorting

A **topological sort** is a ordering of the nodes of a directed graph such that if there is a path from node a to node b, then node a appears before node b in the ordering. For example, for the graph
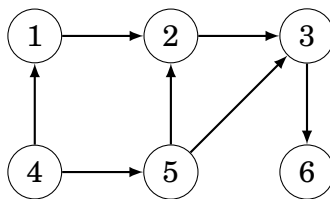
a possible topological sort is $[4,1,5,2,3,6]$:
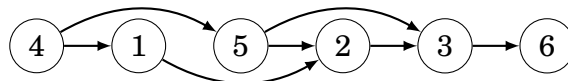
An acyclic graph always has a topological sort. However, if the graph contains a cycle, it is not possible to form a topological sort, because no node in the cycle can appear before the other nodes in the cycle. It turns out that depth-first search can be used to both check if a directed graph contains a cycle and, if it does not contain a cycle, to construct a topological sort.

Algorithm

The idea is to go through the nodes of the graph and always begin a depth-first search at the current node if it has not been processed yet. During the searches, the nodes have three possible states:

- state 0: the node has not been processed (white)

- state 1: the node is under processing (light gray)

- state 2: the node has been processed (dark gray)

Initially, the state of each node is 0. When a search reaches a node for the first time, its state becomes 1. Finally, after all successors of the node have been processed, its state becomes 2.

If the graph contains a cycle, we will find out this during the search, because sooner or later we will arrive at a node whose state is 1. In this case, it is not possible to construct a topological sort.

If the graph does not contain a cycle, we can construct a topological sort by adding each node to a list when the state of the node becomes 2. This list in reverse order is a topological sort.

Example 1

In the example graph, the search first proceeds from node 1 to node 6:
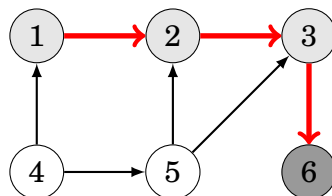
Now node 6 has been processed, so it is added to the list. After this, also nodes 3, 2 and 1 are added to the list:
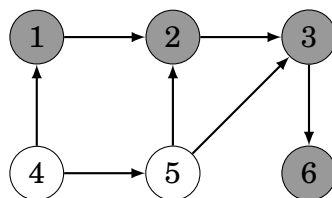
At this point, the list is $[6, 3, 2, 1]$. The next search begins at node 4:
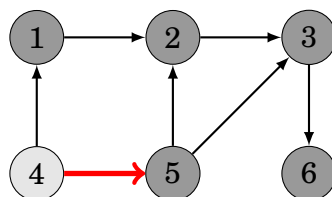
Thus, the final list is $[6,3,2,1,5,4]$. We have processed all nodes, so a topological sort has been found. The topological sort is the reverse list $[4,5,1,2,3,6]$:
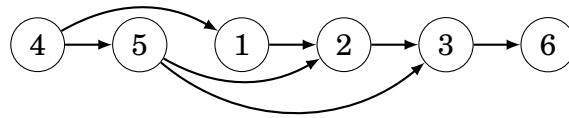
Note that a topological sort is not unique, but there can be several topological sorts for a graph.

Example 2

Let us now consider a graph for which we cannot construct a topological sort, because there is a cycle in the graph:
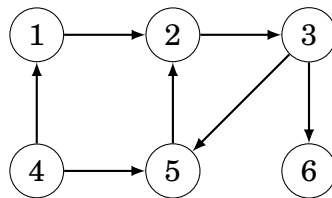
The search proceeds as follows:
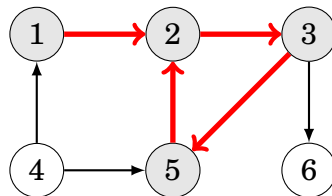
The search reaches node 2 whose state is 1, which means the graph contains a cycle. In this example, the cycle is $2 \rightarrow 3 \rightarrow 5 \rightarrow 2$.

16.2   Dynamic programming

If a directed graph is acyclic, dynamic programming can be applied to it. For example, we can efficiently solve the following problems concerning paths from a starting node to an ending node:

- how many different paths are there?

- what is the shortest/longest path?

- what is the minimum/maximum number of edges in a path?

- which nodes certainly appear in any path?

Counting the number of paths

As an example, let us calculate the number of paths from node 4 to node 6 in the following graph:
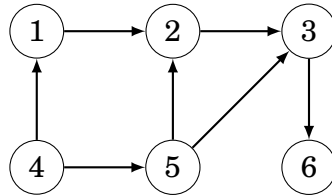
There are a total of three such paths:

- $4 \rightarrow 1 \rightarrow 2 \rightarrow 3 \rightarrow 6$

- $4 \rightarrow 5 \rightarrow 2 \rightarrow 3 \rightarrow 6$

- $4 \rightarrow 5 \rightarrow 3 \rightarrow 6$

To count the paths, we go through the nodes in a topological sort, and calculate for each node x the number of paths from node 4 to node x. A topological sort for the above graph is as follows:
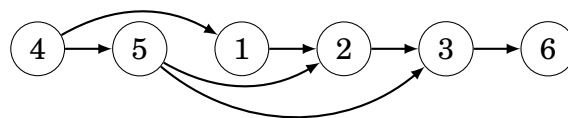
Hence, the numbers of paths are as follows:
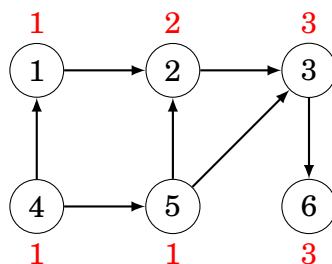
For example, since there are two paths from node 4 to node 2 and there is one path from node 4 to node 5, we can conclude that there are three paths from node 4 to node 3.

Extending Dijkstra's algorithm

A by-product of Dijkstra's algorithm is a directed, acyclic graph that indicates for each node in the original graph the possible ways to reach the node using a shortest path from the starting node. Dynamic programming can be applied to that graph. For example, in the graph
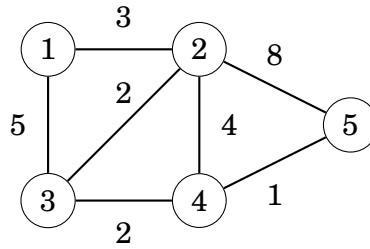
the shortest paths from node 1 may use the following edges:
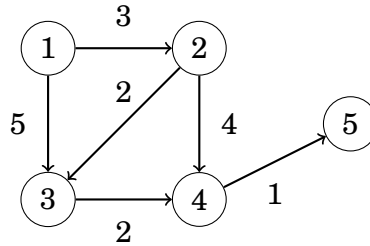
Now we can, for example, calculate the number of shortest paths from node 1 to node 5 using dynamic programming:
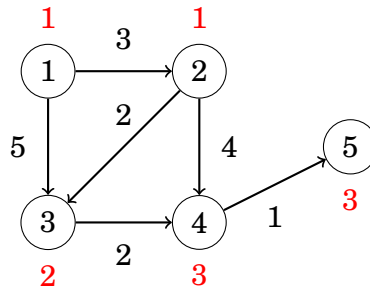
Representing problems as graphs

Actually, any dynamic programming problem can be represented as a directed, acyclic graph. In such a graph, each node corresponds to a dynamic programming state and the edges indicate how the states depend on each other.

As an example, consider the problem of forming a sum of money x using coins $\{c_1, c_2, \ldots, c_k\}$. In this problem, we can construct a graph where each node corresponds to a sum of money, and the edges show how the coins can be chosen. For example, for coins $\{1, 3, 4\}$ and $x = 6$, the graph is as follows:
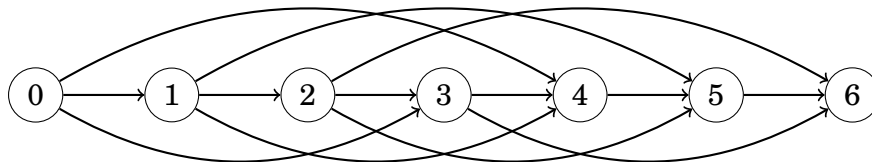
Using this representation, the shortest path from node 0 to node x corresponds to a solution with minimum number of coins, and the total number of paths from node 0 to node x equals the total number of solutions.

16.3 Successor paths

For the rest of the chapter, we will concentrate on **successor graphs** where the outdegree of each node is 1, i.e., exactly one edge starts at each node. A successor graph consists of one or more components, each of which contains one cycle and some paths that lead to it.

Successor graphs are sometimes called **functional graphs**. The reason for this is that any successor graph corresponds to a function f that defines the edges in the graph. The parameter for the function is a node in the graph, and the function gives the successor of the node.

For example, the function

x	1	2	3	4	5	6	7	8	9
$f(x)$	3	5	7	6	2	2	1	6	3

defines the following graph:
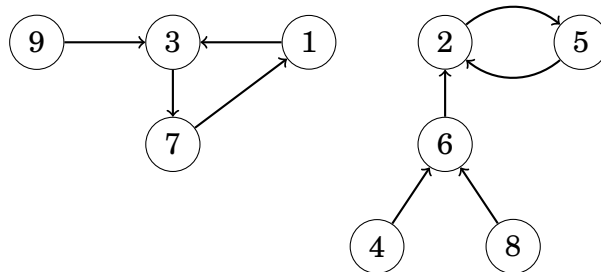
Since each node in a successor graph has a unique successor, we can define a function $f(x,k)$ that returns the node that we will reach if we begin at node x and walk k steps forward. For example, in the above graph $f(4,6) = 2$, because we will reach node 2 by walking 6 steps from node 4:
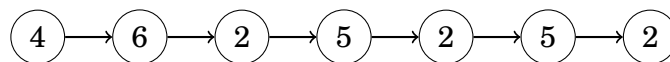
A straightforward way to calculate a value of $f(x,k)$ is to start at node x and walk k steps forward, which takes $O(k)$ time. However, using preprocessing, any value of $f(x,k)$ can be calculated in only $O(\log k)$ time.

The idea is to precalculate all values $f(x,k)$ where k is a power of two and at most u, where u is the maximum number of steps we will ever walk. This can be efficiently done, because we can use the following recursion:

$$f(x,k) = \begin{cases} f(x) & k = 1 \\ f(f(x,k/2),k/2) & k > 1 \end{cases}$$

Precalculating values $f(x,k)$ takes $O(n \log u)$ time, because $O(\log u)$ values are calculated for each node. In the above graph, the first values are as follows:

x	1	2	3	4	5	6	7	8	9
$f(x,1)$	3	5	7	6	2	2	1	6	3
$f(x,2)$	7	2	1	2	5	5	3	2	7
$f(x,4)$	3	2	7	2	5	5	1	2	3
$f(x,8)$	7	2	1	2	5	5	3	2	7
$\cdots$									

After this, any value of $f(x,k)$ can be calculated by presenting the number of steps k as a sum of powers of two. For example, if we want to calculate the value of $f(x,11)$, we first form the representation $11 = 8+2+1$. Using that,

$$f(x,11) = f(f(f(x,8),2),1).$$

For example, in the previous graph

$$f(4,11) = f(f(f(4,8),2),1) = 5.$$

Such a representation always consists of $O(\log k)$ parts, so calculating a value of $f(x,k)$ takes $O(\log k)$ time.

16.4 Cycle detection

Consider a successor graph that only contains a path that ends in a cycle. There are two interesting questions: if we begin our walk at the starting node, what is the first node in the cycle and how many nodes does the cycle contain?

For example, in the graph
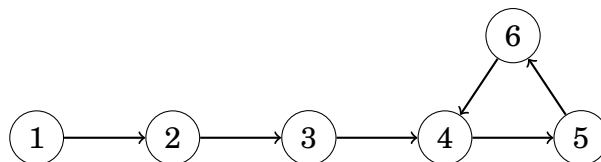
we begin our walk at node 1, the first node that belongs to the cycle is node 4, and the cycle consists of three nodes (4, 5 and 6).

An easy way to detect the cycle is to walk in the graph and keep track of all nodes that have been visited. Once a node is visited for the second time, we can conclude that the node is the first node in the cycle. This method works in $O(n)$ time and also uses $O(n)$ memory.

However, there are better algorithms for cycle detection. The time complexity of such algorithms is still $O(n)$, but they only use $O(1)$ memory. This is an important improvement if n is large. Next we will discuss Floyd's algorithm that achieves these properties.

Floyd's algorithm

Floyd's algorithm walks forward in the graph using two pointers a and b. Both pointers begin at a node x that is the starting node of the graph. Then, on each turn, the pointer a walks one step forward and the pointer b walks two steps forward. The process continues until the pointers meet each other:

```
a = f(x);
b = f(f(x));
while (a != b) {
    a = f(a);
    b = f(f(b));
}
```

At this point, the pointer a has walked k steps and the pointer b has walked $2k$ steps, so the length of the cycle divides k. Thus, the first node that belongs to the cycle can be found by moving the pointer a to node x and advancing the pointers step by step until they meet again:

```
a = x;
while (a != b) {
    a = f(a);
    b = f(b);
}
```

Now a and b point to the first node in the cycle that can be reached from node x. Finally, the length c of the cycle can be calculated as follows:

```
b = f(a);
c = 1;
while (a != b) {
    b = f(b);
    c++;
}
```

Chapter 17

Strongly connectivity

In a directed graph, the edges can be traversed in one direction only, so even if the graph is connected, this does not guarantee that there would be a path from a node to another node. For this reason, it is meaningful to define a new concept that requires more than connectivity.

A graph is **strongly connected** if there is a path from any node to all other nodes in the graph. For example, in the following picture, the left graph is strongly connected while the right graph is not.
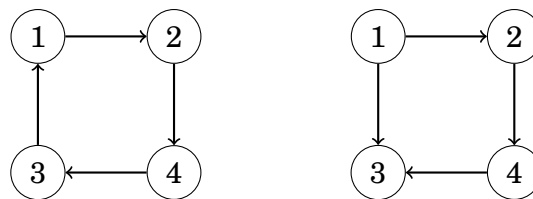
The right graph is not strongly connected because, for example, there is no path from node 2 to node 1.

The **strongly connected components** of a graph divide the graph into strongly connected parts that are as large as possible. The strongly connected components form an acyclic **component graph** that represents the deep structure of the original graph.

For example, for the graph
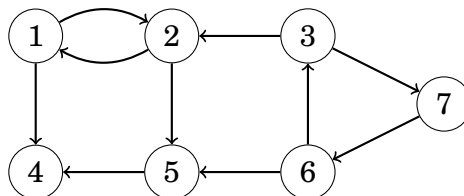
the strongly connected components are as follows:
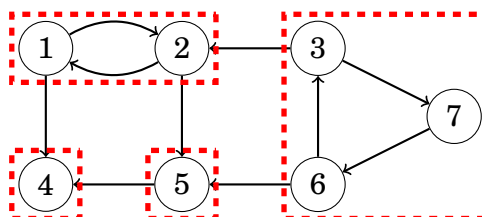
The corresponding component graph is as follows:
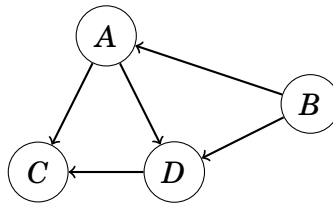
The components are $A = \{1,2\}$, $B = \{3,6,7\}$, $C = \{4\}$ and $D = \{5\}$.

A component graph is an acyclic, directed graph, so it is easier to process than the original graph. Since the graph does not contain cycles, we can always construct a topological sort and use dynamic programming techniques like those presented in Chapter 16.

17.1 Kosaraju's algorithm

Kosaraju's algorithm[1] is an efficient method for finding the strongly connected components of a directed graph. The algorithm performs two depth-first searches: the first search constructs a list of nodes according to the structure of the graph, and the second search forms the strongly connected components.

Search 1

The first phase of Kosaraju's algorithm constructs a list of nodes in the order in which a depth-first search processes them. The algorithm goes through the nodes, and begins a depth-first search at each unprocessed node. Each node will be added to the list after it has been processed.

In the example graph, the nodes are processed in the following order:
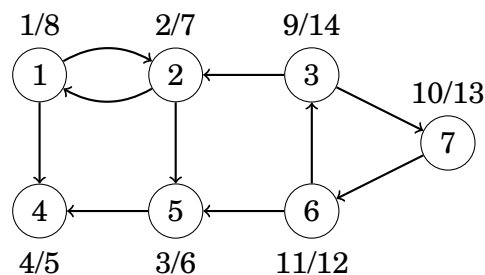
The notation x/y means that processing the node started at time x and finished at time y. Thus, the corresponding list is as follows:

_____

[1]According to [1], S. R. Kosaraju invented this algorithm in 1978 but did not publish it. In 1981, the same algorithm was rediscovered and published by M. Sharir [36].

node	processing time
4	5
5	6
2	7
1	8
6	12
7	13
3	14

Search 2

The second phase of the algorithm forms the strongly connected components of the graph. First, the algorithm reverses every edge in the graph. This guarantees that during the second search, we will always find strongly connected components that do not have extra nodes.

After reversing the edges, the example graph is as follows:
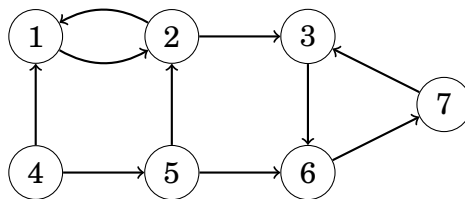
After this, the algorithm goes through the list of nodes created by the first search in *reverse* order. If a node does not belong to a component, the algorithm creates a new component and starts a depth-first search that adds all new nodes found during the search to the new component.

In the example graph, the first component begins at node 3:
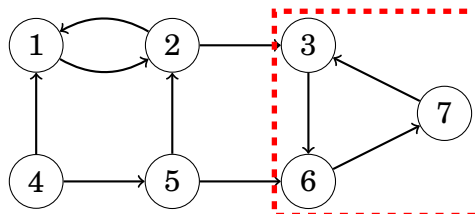
Note that since all edges are reversed, the component does not "leak" to other parts in the graph.

The next nodes in the list are nodes 7 and 6, but they already belong to a component. The next new component begins at node 1:
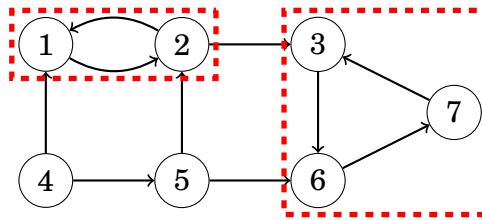
Finally, the algorithm processes nodes 5 and 4 that create the remaining strongy connected components:
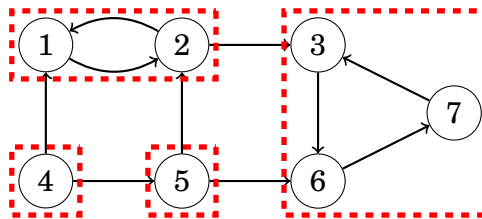
The time complexity of the algorithm is $O(n + m)$, because the algorithm performs two depth-first searches.

17.2   2SAT problem

Strongly connectivity is also linked with the **2SAT problem** [4]. In this problem, we are given a logical formula

$$(a_1 \vee b_1) \wedge (a_2 \vee b_2) \wedge \cdots \wedge (a_m \vee b_m),$$

where each a_i and b_i is either a logical variable $(x_1, x_2, \ldots, x_n)$ or a negation of a logical variable $(\neg x_1, \neg x_2, \ldots, \neg x_n)$. The symbols "$\wedge$" and "$\vee$" denote logical operators "and" and "or". Our task is to assign each variable a value so that the formula is true, or state that this is not possible.

For example, the formula

$$L_1 = (x_2 \vee \neg x_1) \wedge (\neg x_1 \vee \neg x_2) \wedge (x_1 \vee x_3) \wedge (\neg x_2 \vee \neg x_3) \wedge (x_1 \vee x_4)$$

is true when the variables are assigned as follows:

$$\begin{cases} x_1 = \text{false} \\ x_2 = \text{false} \\ x_3 = \text{true} \\ x_4 = \text{true} \end{cases}$$

However, the formula

$$L_2 = (x_1 \vee x_2) \wedge (x_1 \vee \neg x_2) \wedge (\neg x_1 \vee x_3) \wedge (\neg x_1 \vee \neg x_3)$$

is always false, regardless of how we assign the values. The reason for this is that we cannot choose a value for x_1 without creating a contradiction. If x_1 is false, both x_2 and $\neg x_2$ should be true which is impossible, and if x_1 is true, both x_3 and $\neg x_3$ should be true which is also impossible.

The 2SAT problem can be represented as a graph whose nodes correspond to variables x_i and negations $\neg x_i$, and edges determine the connections between the variables. Each pair $(a_i \vee b_i)$ generates two edges: $\neg a_i \to b_i$ and $\neg b_i \to a_i$. This means that if a_i does not hold, b_i must hold, and vice versa.

The graph for the formula L_1 is:
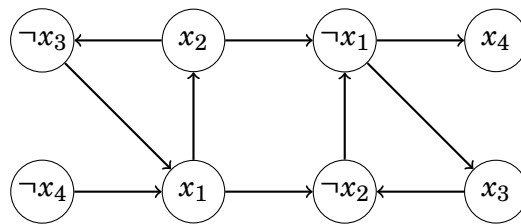
And the graph for the formula L_2 is:
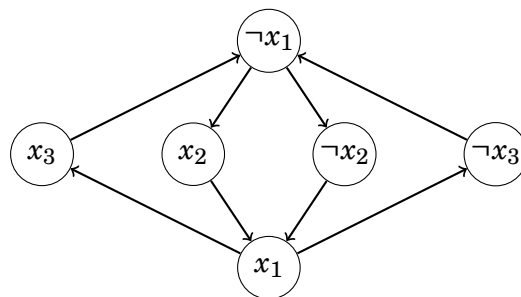
The structure of the graph tells us whether it is possible to assign the values of the variables so that the formula is true. It turns out that this can be done exactly when there are no nodes x_i and $\neg x_i$ such that both nodes belong to the same strongly connected component. If there are such nodes, the graph contains a path from x_i to $\neg x_i$ and also a path from $\neg x_i$ to x_i, so both x_i and $\neg x_i$ should be true which is not possible.

In the graph of the formula L_1 there are no nodes x_i and $\neg x_i$ such that both nodes belong to the same strongly connected component, so there is a solution. In the graph of the formula L_2 all nodes belong to the same strongly connected component, so there are no solutions.

If a solution exists, the values for the variables can be found by going through the nodes of the component graph in a reverse topological sort order. At each step, we process a component that does not contain edges that lead to an unprocessed component. If the variables in the component have not been assigned values, their values will be determined according to the values in the component, and if they already have values, they remain unchanged. The process continues until all variables have been assigned values.

The component graph for the formula L_1 is as follows:
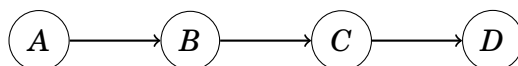
The components are $A = \{\neg x_4\}$, $B = \{x_1, x_2, \neg x_3\}$, $C = \{\neg x_1, \neg x_2, x_3\}$ and $D = \{x_4\}$. When constructing the solution, we first process the component D where x_4 becomes true. After this, we process the component C where x_1 and x_2 become false and x_3 becomes true. All variables have been assigned a value, so the remaining components A and B do not change the variables.

Note that this method works, because the graph has a special structure. If there are paths from node x_i to node x_j and from node x_j to node $\neg x_j$, then node x_i never becomes true. The reason for this is that there is also a path from node $\neg x_j$ to node $\neg x_i$, and both x_i and x_j become false.

A more difficult problem is the **3SAT problem** where each part of the formula is of the form $(a_i \lor b_i \lor c_i)$. This problem is NP-hard [19], so no efficient algorithm for solving the problem is known.

Chapter 18

Tree queries

This chapter discusses techniques for processing queries related to subtrees and paths of a rooted tree. For example, such queries are:

- what is the kth ancestor of a node?

- what is the sum of values in the subtree of a node?

- what is the sum of values in a path between two nodes?

- what is the lowest common ancestor of two nodes?

18.1 Finding ancestors

The kth **ancestor** of a node x in a rooted tree is the node that we will reach if we move k levels up from x. Let $f(x,k)$ denote the kth ancestor of x. For example, in the following tree, $f(2,1) = 1$ and $f(8,2) = 4$.
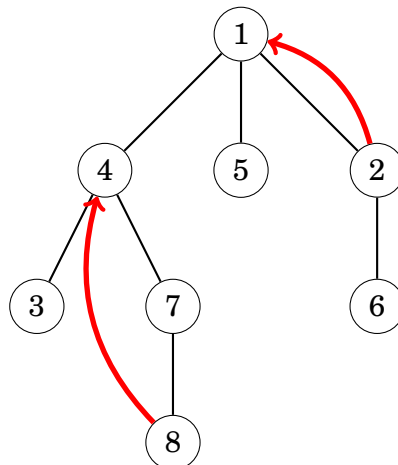
An easy way to calculate the value of $f(x,k)$ is to perform a sequence of k moves in the tree. However, the time complexity of this method is $O(n)$, because the tree may contain a chain of $O(n)$ nodes.

Fortunately, it turns out that using a technique similar to that used in Chapter 16.3, any value of $f(x,k)$ can be efficiently calculated in $O(\log k)$ time after

preprocessing. The idea is to precalculate all values $f(x,k)$ where k is a power of two. For example, the values for the above tree are as follows:

x	1	2	3	4	5	6	7	8
$f(x,1)$	0	1	4	1	1	2	4	7
$f(x,2)$	0	0	1	0	0	1	1	4
$f(x,4)$	0	0	0	0	0	0	0	0
...								

The value 0 means that the kth ancestor of a node does not exist.

The preprocessing takes $O(n \log n)$ time, because each node can have at most n ancestors. After this, any value of $f(x,k)$ can be calculated in $O(\log k)$ time by representing k as a sum where each term is a power of two.

18.2 Subtrees and paths

A **tree traversal array** contains the nodes of a rooted tree in the order in which a depth-first search from the root node visits them. For example, in the tree
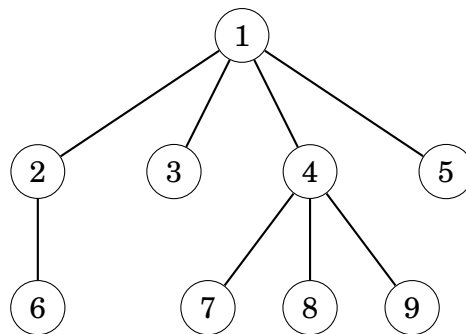
a depth-first search proceeds as follows:
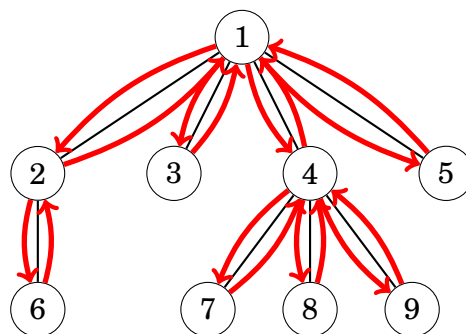
Hence, the corresponding tree traversal array is as follows:

1	2	3	4	5	6	7	8	9
1	2	6	3	4	7	8	9	5

Subtree queries

Each subtree of a tree corresponds to a subarray in the tree traversal array such that the first element in the subarray is the root node. For example, the following subarray contains the nodes in the subtree of node 4:

1	2	3	4	5	6	7	8	9
1	2	6	3	4	7	8	9	5

Using this fact, we can efficiently process queries that are related to subtrees of a tree. As an example, consider a problem where each node is assigned a value, and our task is to support the following queries:

- update the value of a node

- calculate the sum of values in the subtree of a node

Consider the following tree where the blue numbers are the values of the nodes. For example, the sum of the subtree of node 4 is $3 + 4 + 3 + 1 = 11$.
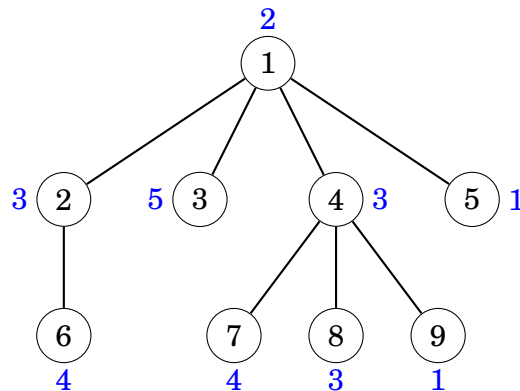
The idea is to construct a tree traversal array that contains three values for each node: (1) the identifier of the node, (2) the size of the subtree, and (3) the value of the node. For example, the array for the above tree is as follows:

1	2	3	4	5	6	7	8	9
1	2	6	3	4	7	8	9	5
9	2	1	1	4	1	1	1	1
2	3	4	5	3	4	3	1	1

Using this array, we can calculate the sum of values in any subtree by first finding out the size of the subtree and then the values of the corresponding nodes. For example, the values in the subtree of node 4 can be found as follows:

1	2	3	4	5	6	7	8	9
1	2	6	3	4	7	8	9	5
9	2	1	1	4	1	1	1	1
2	3	4	5	3	4	3	1	1

To answer the queries efficiently, it suffices to store the values of the nodes in a binary indexed tree or segment tree. After this, we can both update a value and calculate the sum of values in $O(\log n)$ time.

Path queries

Using a tree traversal array, we can also efficiently calculate sums of values on paths from the root node to any other node in the tree. Let us next consider a problem where our task is to support the following queries:

- change the value of a node

- calculate the sum of values on a path from the root to a node

For example, in the following tree, the sum of values from the root node to node 7 is $4+5+5=14$:
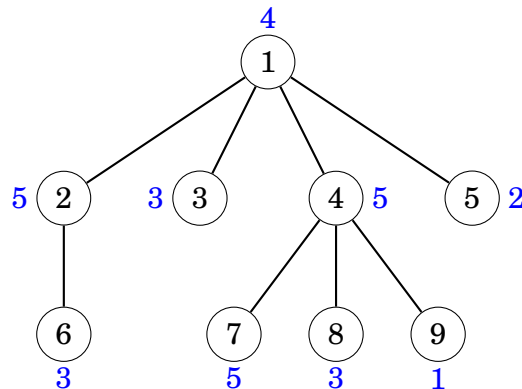
We can solve this problem in a similar way as before, but now each value in the last row of the array is the sum of values on a path from the root to the node. For example, the following array corresponds to the above tree:

1	2	3	4	5	6	7	8	9
1	2	6	3	4	7	8	9	5
9	2	1	1	4	1	1	1	1
4	9	12	7	9	14	12	10	6

When the value of a node increases by x, the sums of all nodes in its subtree increase by x. For example, if the value of node 4 increases by 1, the array changes as follows:

1	2	3	4	5	6	7	8	9
1	2	6	3	4	7	8	9	5
9	2	1	1	4	1	1	1	1
4	9	12	7	10	15	13	11	6

Thus, to support both the operations, we should be able to increase all values in a range and retrieve a single value. This can be done in $O(\log n)$ time using a binary indexed tree or segment tree (see Chapter 9.4).

18.3   Lowest common ancestor

The **lowest common ancestor** of two nodes in the tree is the lowest node whose subtree contains both the nodes. A typical problem is to efficiently process queries that ask to find the lowest common ancestor of given two nodes.

For example, in the following tree, the lowest common ancestor of nodes 5 and 8 is node 2:
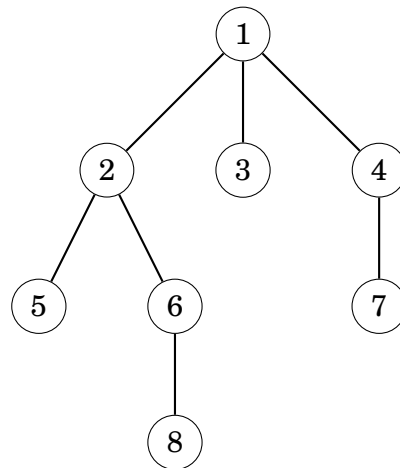
Next we will discuss two efficient techniques for finding the lowest common ancestor of two nodes.

Method 1

One way to solve the problem is to use the fact that we can efficiently find the kth ancestor of any node in the tree. Thus, we can first make sure that both nodes are at the same level in the tree, and then find the smallest value of k such that the kth ancestor of both nodes is the same.

As an example, let us find the lowest common ancestor of nodes 5 and 8:
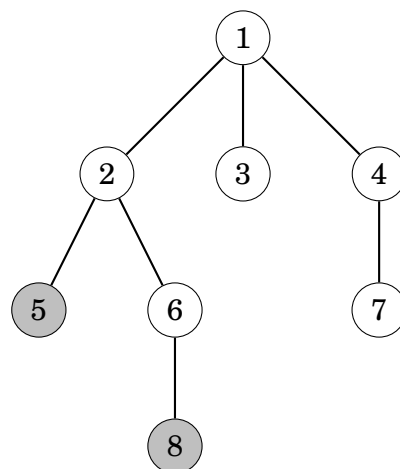
Node 5 is at level 3, while node 8 is at level 4. Thus, we first move one step upwards from node 8 to node 6. After this, it turns out that the parent of both nodes 5 and 6 is node 2, so we have found the lowest common ancestor.

The following picture shows how we move in the tree:
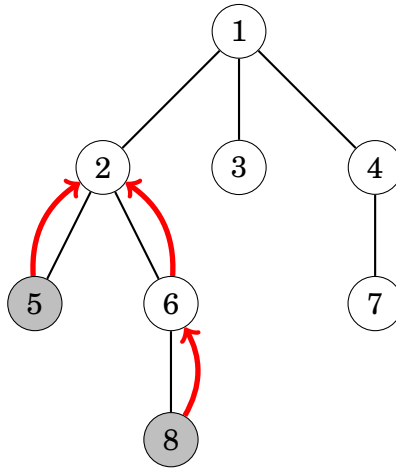
Using this method, we can find the lowest common ancestor of any two nodes in $O(\log n)$ time after an $O(n \log n)$ time preprocessing, because both steps can be performed in $O(\log n)$ time.

Method 2

Another way to solve the problem is based on a tree traversal array [6]. Once again, the idea is to traverse the nodes using a depth-first search:
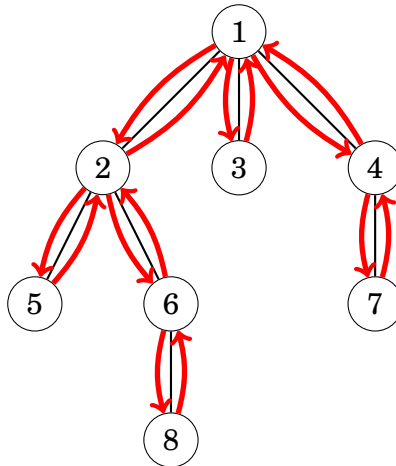
However, we use a bit different technique where we add each node to the tree traversal array *always* when the depth-first search visits the node, and not only at the first visit. Hence, a node that has k children appears $k+1$ times in the array, and there are a total of $2n-1$ nodes in the array.

We store two values in the array: (1) the identifier of the node, and (2) the level of the node in the tree. The following array corresponds to the above tree:

1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
1	2	5	2	6	8	6	2	1	3	1	4	7	4	1
1	2	3	2	3	4	3	2	1	2	1	2	3	2	1

Using this array, we can find the lowest common ancestor of nodes a and b by finding the node with lowest level between nodes a and b in the array. For example, the lowest common ancestor of nodes 5 and 8 can be found as follows:

1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
1	2	5	2	6	8	6	2	1	3	1	4	7	4	1
1	2	3	2	3	4	3	2	1	2	1	2	3	2	1

$\uparrow$

Node 5 is at position 3, node 8 is at position 6, and the node with lowest level between positions $3\ldots6$ is node 2 at position 4 whose level is 2. Thus, the lowest common ancestor of nodes 5 and 8 is node 2.

Thus, to find the lowest common ancestor of two nodes it suffices to process a range minimum query. Since the array is static, we can process such queries in $O(1)$ time after an $O(n \log n)$ time preprocessing.

Distances of nodes

Finally, let us consider the problem of finding the distance between two nodes in the tree, which equals the length of the path between them. It turns out that this problem reduces to finding the lowest common ancestor of the nodes.

First, we root the tree arbitrarily. After this, the distance between nodes a and b can be calculated using the formula

$$d(a) + d(b) - 2 \cdot d(c),$$

where c is the lowest common ancestor of a and b and $d(s)$ denotes the distance from the root node to node s. For example, in the tree
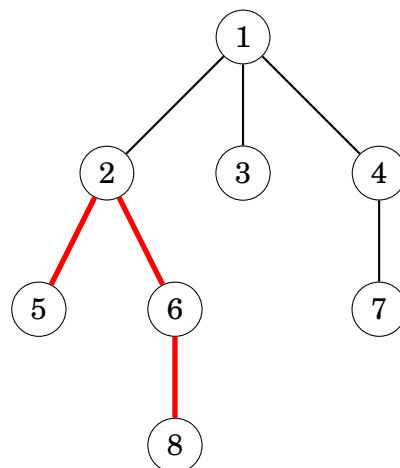
the lowest common ancestor of nodes 5 and 8 is node 2. A path from node 5 to node 8 first ascends from node 5 to node 2 and then descends from node 2 to node 8. The distances of the nodes from the root are $d(5) = 3$, $d(8) = 4$ and $d(2) = 2$, so the distance between nodes 5 and 8 is $3 + 4 - 2 \cdot 2 = 3$.

Chapter 19

Paths and circuits

This chapter focuses on two types of paths in graphs:

- An **Eulerian path** is a path that goes through each edge exactly once.

- A **Hamiltonian path** is a path that visits each node exactly once.

While Eulerian and Hamiltonian paths look like similar concepts at first glance, the computational problems related to them are very different. It turns out that there is a simple rule that determines whether a graph contains an Eulerian path and there is also an efficient algorithm to find a such path if it exists. On the contrary, checking the existence of a Hamiltonian path is a NP-hard problem and no efficient algorithm is known for solving the problem.

19.1   Eulerian paths

An **Eulerian path** is a path that goes exactly once through each edge in the graph. For example, the graph
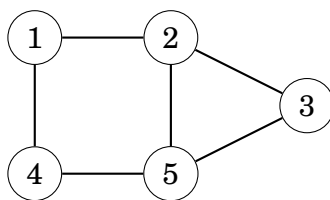
has an Eulerian path from node 2 to node 5:
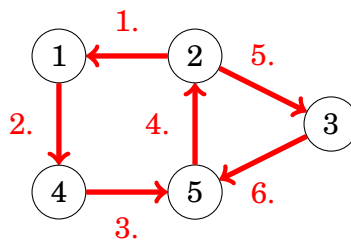
An **Eulerian circuit** is an Eulerian path that starts and ends at the same node. For example, the graph
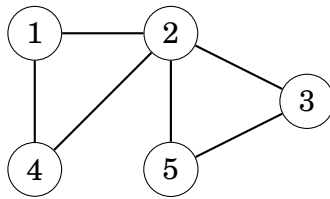
has an Eulerian circuit that starts and ends at node 1:
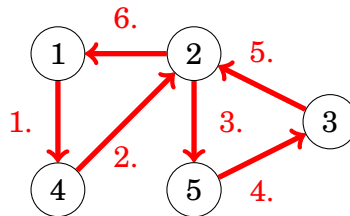
Existence

The existence of Eulerian paths and circuits depends on the degrees of the nodes in the graph. First, an undirected graph has an Eulerian path if all the edges belong to the same connected component and

- the degree of each node is even *or*

- the degree of exactly two nodes is odd, and the degree of all other nodes is even.

In the first case, each Eulerian path in the graph is also an Eulerian circuit. In the second case, the odd-degree nodes are the starting and ending nodes of an Eulerian path which is not an Eulerian circuit.

For example, in the graph
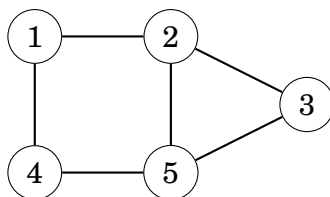
nodes 1, 3 and 4 have a degree of 2, and nodes 2 and 5 have a degree of 3. Exactly two nodes have an even degree, so there is an Eulerian path between nodes 2 and 5, but the graph does not contain an Eulerian circuit.

In a directed graph, we focus on indegrees and outdegrees of the nodes of the graph. A directed graph contains an Eulerian path if all the edges belong to the same strongly connected component and

- in each node, the indegree equals the outdegree, *or*

- in one node, the indegree is one larger than the outdegree, in another node, the outdegree is one larger than the indegree, and all other nodes, the indegree equals the outdegree.

In the first case, each Eulerian path in the graph is also an Eulerian circuit, and in the second case, the graph contains an Eulerian path that begins at the node whose outdegree is larger and ends at the node whose indegree is larger.

For example, in the graph
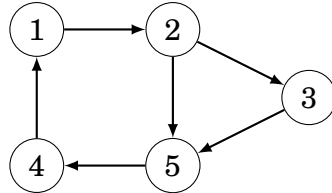
nodes 1, 3 and 4 have both indegree 1 and outdegree 1, node 2 has indegree 1 and outdegree 2, and node 5 has indegree 2 and outdegree 1. Hence, the graph contains an Eulerian path from node 2 to node 5:
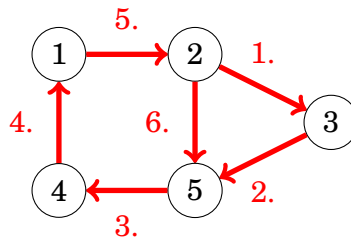
Hierholzer's algorithm

Hierholzer's algorithm is an efficient method for constructing an Eulerian circuit. The algorithm consists of several rounds, each of which adds new edges to the circuit. Of course, we assume that the graph contains an Eulerian circuit; otherwise Hierholzer's algorithm cannot find it.

First, the algorithm constructs a circuit that contains some (not necessarily all) of the edges in the graph. After this, the algorithm extends the circuit step by step by adding subcircuits to it. The process continues until all edges have been added to the circuit.

The algorithm extends the circuit by always finding a node x that belongs to the circuit but has an outgoing edge that is not included in the circuit. The algorithm constructs a new path from node x that only contains edges that are not yet in the circuit. Sooner or later, the path will return to the node x, which creates a subcircuit.

If the graph only contains an Eulerian path, we can still use Hierholzer's algorithm to find it by adding an extra edge to the graph and removing the edge after the circuit has been constructed. For example, in an undirected graph, we add the extra edge between the two odd-degree nodes.

Next we will see how Hierholzer's algorithm constructs an Eulerian circuit in an undirected graph.

Example

Let us consider the following graph:
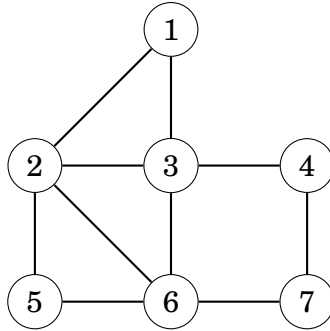
Suppose that the algorithm first creates a circuit that begins at node 1. A possible circuit is $1 \rightarrow 2 \rightarrow 3 \rightarrow 1$:
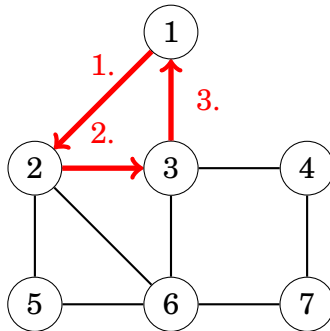
After this, the algorithm adds the subcircuit $2 \rightarrow 5 \rightarrow 6 \rightarrow 2$ to the circuit:
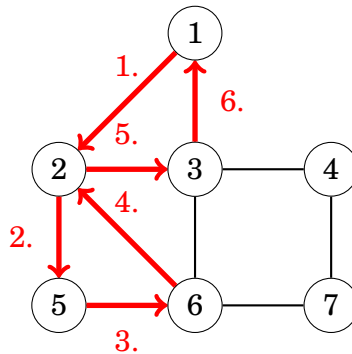
Finally, the algorithm adds the subcircuit $6 \rightarrow 3 \rightarrow 4 \rightarrow 7 \rightarrow 6$ to the circuit:
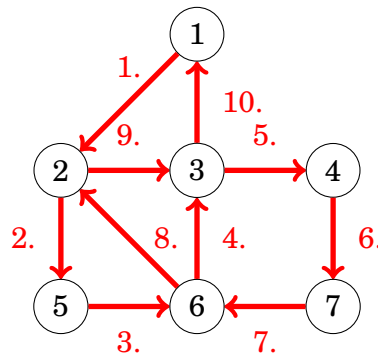
Now all edges are included in the circuit, so we have successfully constructed an Eulerian circuit.

19.2   Hamiltonian paths

A **Hamiltonian path** is a path that visits each node in the graph exactly once. For example, the graph
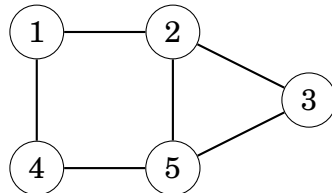
contains a Hamiltonian path from node 1 to node 3:
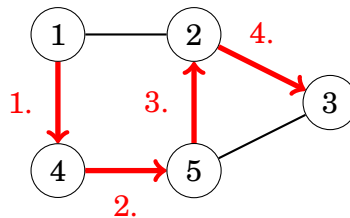
If a Hamiltonian path begins and ends at the same node, it is called a **Hamiltonian circuit**. The graph above also has an Hamiltonian circuit that begins and ends at node 1:
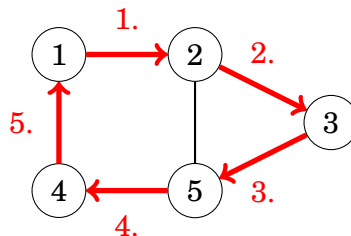
Existence

No efficient method is known for testing if a graph contains a Hamiltonian path, but the problem is NP-hard. Still, in some special cases we can be certain that the graph contains a Hamiltonian path.

A simple observation is that if the graph is complete, i.e., there is an edge between all pairs of nodes, it also contains a Hamiltonian path. Also stronger results have been achieved:

- **Dirac's theorem**: If the degree of each node is at least $n/2$, the graph contains a Hamiltonian path.

- **Ore's theorem**: If the sum of degrees of each non-adjacent pair of nodes is at least n, the graph contains a Hamiltonian path.

A common property in these theorems and other results is that they guarantee the existence of a Hamiltonian if the graph has *a large number* of edges. This makes sense, because the more edges the graph contains, the more possibilities there is to construct a Hamiltonian graph.

Construction

Since there is no efficient way to check if a Hamiltonian path exists, it is clear that there is also no method for constructing the path efficiently, because otherwise we could just try to construct the path and see whether it exists.

A simple way to search for a Hamiltonian path is to use a backtracking algorithm that goes through all possible ways to construct the path. The time complexity of such an algorithm is at least $O(n!)$, because there are $n!$ different ways to choose the order of n nodes.

A more efficient solution is based on dynamic programming (see Chapter 10.4). The idea is to define a function $f(s,x)$, where s is a subset of nodes and x is one of the nodes in the subset. The function indicates whether there is a Hamiltonian path that visits the nodes in s and ends at node x. It is possible to implement this solution in $O(2^n n^2)$ time.

19.3 De Bruijn sequences

A **De Bruijn sequence** is a string that contains every string of length n exactly once as a substring, for a fixed alphabet of k characters. The length of such a string is $k^n + n - 1$ characters. For example, when $n = 3$ and $k = 2$, an example of a De Bruijn sequence is

$$0001011100.$$

The substrings of this string are all combinations of three bits: 000, 001, 010, 011, 100, 101, 110 and 111.

It turns out that each De Bruijn sequence corresponds to an Eulerian circuit in a graph. The idea is to construct the graph so that each node contains a combination of $n - 1$ characters and each edge adds one character to the string. The following graph corresponds to the above example:
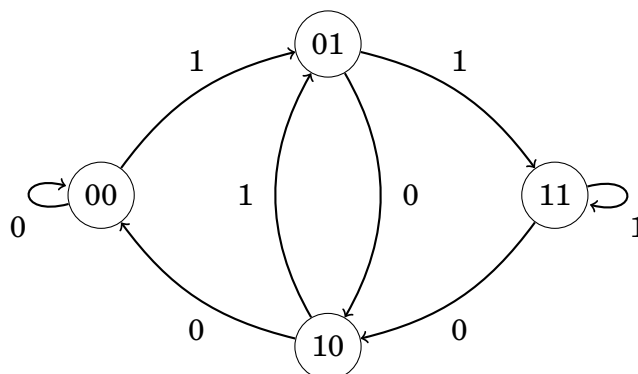
An Eulerian path in this graph corresponds to a string that contains all strings of length n. The string contains the characters of the starting node and all characters in the edges. The starting node has $n-1$ characters and there are k^n characters in the edges, so the length of the string is $k^n + n - 1$.

19.4 Knight's tours

A **knight's tour** is a sequence of moves of a knight on an $n \times n$ chessboard following the rules of chess such that the knight visits each square exactly once. The tour is **closed** if the knight finally returns to the starting square and otherwise the tour is **open**.

For example, here is an open knight's tour on a 5×5 board:

1	4	11	16	25
12	17	2	5	10
3	20	7	24	15
18	13	22	9	6
21	8	19	14	23

A knight's tour corresponds to a Hamiltonian path in a graph whose nodes represent the squares of the board and two nodes are connected with an edge if a knight can move between the squares according to the rules of chess.

A natural way to construct a knight's tour is to use backtracking. The search can be made more efficient by using **heuristics** that attempt to guide the knight so that a complete tour will be found quickly.

Warnsdorff's rule

Warnsdorff's rule is a simple and effective heuristic for finding a knight's tour. Using the rule, it is possible to efficiently construct a tour even on a large board. The idea is to always move the knight so that it ends up in a square where the number of possible moves is as *small* as possible.

For example, in the following situation there are five possible squares to which the knight can move:

1				a
		2		
b				e
	c		d	

In this situation, Warnsdorff's rule moves the knight to square a, because after this choice, there is only a single possible move. The other choices would move the knight to squares where there would be three moves available.

Chapter 20

Flows and cuts

In this chapter, we will focus on the following two problems:

- **Finding a maximum flow**: What is the maximum amount of flow we can send from a node to another node?

- **Finding a minimum cut**: What is a minimum-weight set of edges that separates two nodes of the graph?

The input for both these problems is a directed, weighted graph that contains two special nodes: the **source** is a node with no incoming edges, and the **sink** is a node with no outgoing edges.

As an example, we will use the following graph where node 1 is the source and node 6 is the sink:
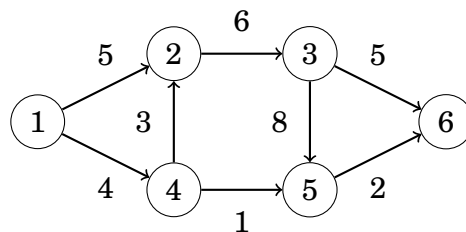
Maximum flow

In the **maximum flow** problem, our task is to send as much flow as possible from the source to the sink. The weight of each edge is a capacity that restricts the flow that can go through the edge. In each intermediate node, the incoming and outgoing flow has to be equal.

For example, the size of the maximum flow in the example graph is 7. The following picture shows how we can route the flow:
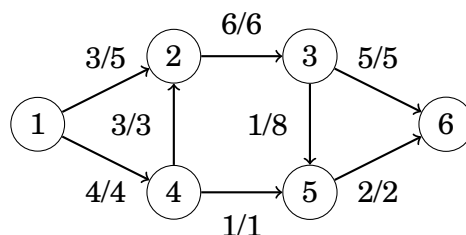
The notation v/k means that a flow of v units is routed through an edge whose capacity is k units. The size of the flow is 7, because the source sends $3+4$ units of flow and the sink receives $5+2$ units of flow. It is easy see that this flow is maximum, because the total capacity of the edges leading to the sink is 7.

Minimum cut

In the **minimum cut** problem, our task is to remove a set of edges from the graph such that there will be no path from the source to the sink after the removal and the total weight of the removed edges is minimum.

The size of the minimum cut in the example graph is 7. It suffices to remove the edges $2 \rightarrow 3$ and $4 \rightarrow 5$:
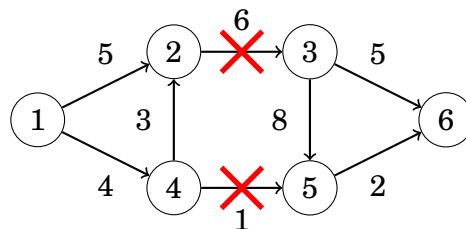
After removing the edges, there will be no path from the source to the sink. The size of the cut is 7, because the weights of the removed edges are 6 and 1. The cut is minimum, because there is no valid way to remove edges from the graph such that their total weight would be less than 7.

It is not a coincidence that both the size of the maximum flow and the size of the minimum cut is 7 in the above example. It turns out that the size of the maximum flow and the minimum cut is *always* the same, so the concepts are two sides of the same coin.

Next we will discuss the Ford–Fulkerson algorithm that can be used to find the maximum flow and minimum cut of a graph. The algorithm also helps us to understand *why* they are equally large.

20.1   Ford–Fulkerson algorithm

The **Ford–Fulkerson algorithm** [17] finds the maximum flow in a graph. The algorithm begins with an empty flow, and at each step finds a path in the graph that generates more flow. Finally, when the algorithm cannot increase the flow anymore, it terminates and the maximum flow has been found.

The algorithm uses a special representation of the graph where each original edge has a reverse edge in another direction. The weight of each edge indicates how much more flow we could route through it. At the beginning of the algorithm, the weight of each original edge equals the capacity of the edge and the weight of each reverse edge is zero.

The new representation for the example graph is as follows:
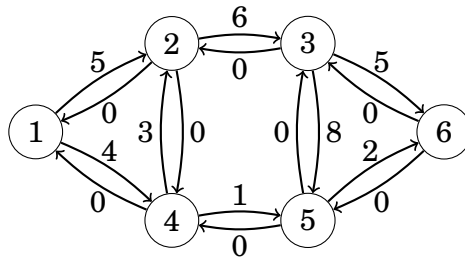

Algorithm description

The Ford–Fulkerson algorithm consists of several rounds. On each round, the algorithm finds a path from the source to the sink such that each edge on the path has a positive weight. If there is more than one possible path available, we can choose any of them.

For example, suppose we choose the following path:
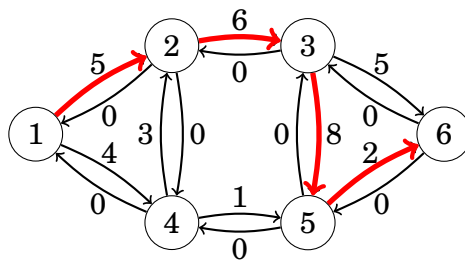
After choosing the path, the flow increases by x units, where x is the smallest edge weight on the path. In addition, the weight of each edge on the path decreases by x and the weight of each reverse edge increases by x.

In the above path, the weights of the edges are 5, 6, 8 and 2. The smallest weight is 2, so the flow increases by 2 and the new graph is as follows:
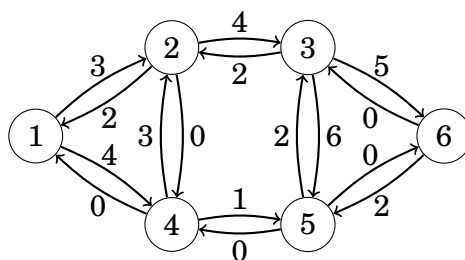
The idea is that increasing the flow decreases the amount of flow that can go through the edges in the future. On the other hand, it is possible to modify the flow later using the reverse edges of the graph if it turns out that it would be beneficial to route the flow in another way.

The algorithm increases the flow as long as there is a path from the source to the sink through positive-weight edges. In the present example, our next path can be as follows:
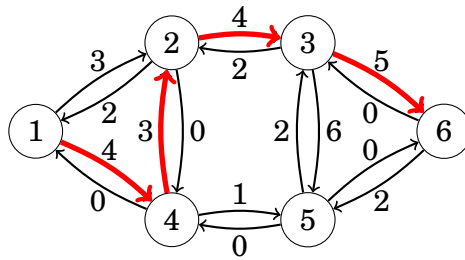
The minimum edge weight on this path is 3, so the path increases the flow by 3, and the total flow after processing the path is 5.

The new graph will be as follows:
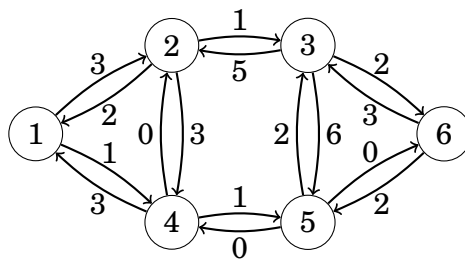
We still need two more rounds before reaching the maximum flow. For example, we can choose the paths $1 \to 2 \to 3 \to 6$ and $1 \to 4 \to 5 \to 3 \to 6$. Both paths increase the flow by 1, and the final graph is as follows:
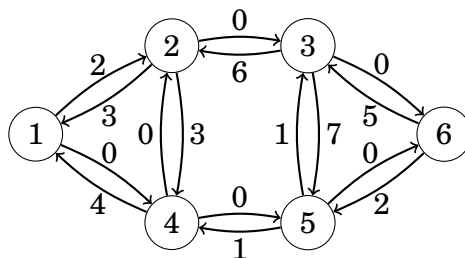
It is not possible to increase the flow anymore, because there is no path from the source to the sink with positive edge weights. Hence, the algorithm terminates and the maximum flow is 7.

Finding paths

The Ford–Fulkerson algorithm does not specify how we should choose the paths that increase the flow. In any case, the algorithm will terminate sooner or later and correctly find the maximum flow. However, the efficiency of the algorithm depends on the way the paths are chosen.

A simple way to find paths is to use depth-first search. Usually, this works well, but in the worst case, each path only increases the flow by 1 and the algorithm is slow. Fortunately, we can avoid this situation by using one of the following techniques:

The **Edmonds–Karp algorithm** [11] chooses each path so that the number of edges on the path is as small as possible. This can be done by using breadth-first search instead of depth-first search for finding paths. It turns out that this guarantees that the flow increases quickly, and the time complexity of the algorithm is $O(m^2 n)$.

The **scaling algorithm** [2] uses depth-first search to find paths where each edge weight is at least a threshold value. Initially, the threshold value is the sum of capacities of the edges that start at the source. Always when a path cannot be found, the threshold value is divided by 2. The time complexity of the algorithm is $O(m^2 \log c)$, where c is the initial threshold value.

In practice, the scaling algorithm is easier to implement, because depth-first search can be used for finding paths. Both algorithms are efficient enough for problems that typically appear in programming contests.

Minimum cuts

It turns out that once the Ford–Fulkerson algorithm has found a maximum flow, it has also found a minimum cut. Let A be the set of nodes that can be reached from the source using positive-weight edges. In the example graph, A contains nodes 1, 2 and 4:
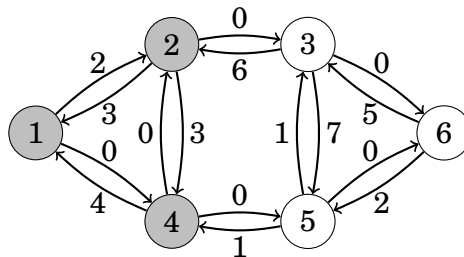
Now the minimum cut consists of the edges of the original graph that start at some node in A, end at some node outside A, and whose capacity is fully used in the maximum flow. In the above graph, such edges are $2 \rightarrow 3$ and $4 \rightarrow 5$, that correspond to the minimum cut $6 + 1 = 7$.

Why is the flow produced by the algorithm maximum and why is the cut minimum? The reason is that a graph cannot contain a flow whose size is larger than the weight of any cut in the graph. Hence, always when a flow and a cut are equally large, they are a maximum flow and a minimum cut.

Let us consider any cut in the graph such that the source belongs to A, the sink belongs to B and there are edges between the sets:
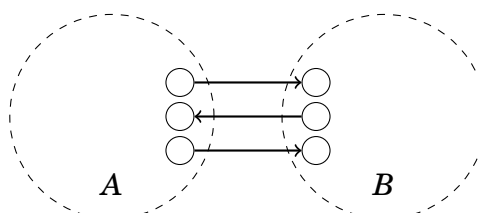
The size of the cut is the sum of the edges that go from the set A to the set B. This is an upper bound for the flow in the graph, because the flow has to proceed from the set A to the set B. Thus, a maximum flow is smaller than or equal to any cut in the graph.

On the other hand, the Ford–Fulkerson algorithm produces a flow that is *exactly* as large as a cut in the graph. Thus, the flow has to be a maximum flow and the cut has to be a minimum cut.
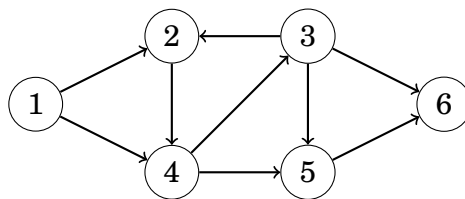
20.2 Disjoint paths

Many graph problems can be solved by reducing them to the maximum flow problem. Our first example of such a problem is as follows: we are given a directed graph with a source and a sink, and our task is to find the maximum number of disjoint paths from the source to the sink.

Edge-disjoint paths

We will first focus on the problem of finding the maximum number of **edge-disjoint paths** from the source to the sink. This means that we should construct a set of paths such that each edge appears in at most one path.

For example, consider the following graph:

In this graph, the maximum number of edge-disjoint paths is 2. We can choose the paths $1 \rightarrow 2 \rightarrow 4 \rightarrow 3 \rightarrow 6$ and $1 \rightarrow 4 \rightarrow 5 \rightarrow 6$ as follows:
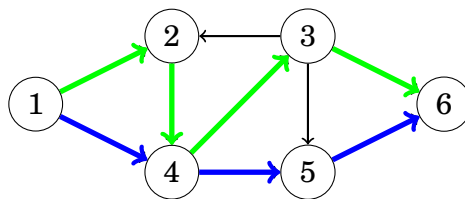
It turns out that the maximum number of edge-disjoint paths equals the maximum flow of the graph, assuming that the capacity of each edge is one. After the maximum flow has been constructed, the edge-disjoint paths can be found greedily by following paths from the source to the sink.

Node-disjoint paths

Let us now consider another problem: finding the maximum number of **node-disjoint paths** from the source to the sink. In this problem, every node, except

for the source and sink, may appear in at most one path. The number of node-disjoint paths is often smaller than the number of edge-disjoint paths.

For example, in the previous graph, the maximum number of node-disjoint paths is 1:
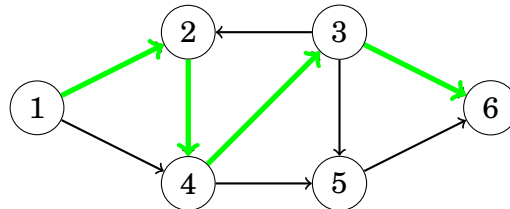

We can reduce also this problem to the maximum flow problem. Since each node can appear in at most one path, we have to limit the flow that goes through the nodes. A standard method for this is to divide each node into two nodes such that the first node has the incoming edges of the original node and the second node has the outgoing edges of the original node. In addition, there is a new edge from the first node to the second node.

In our example, the graph becomes as follows:
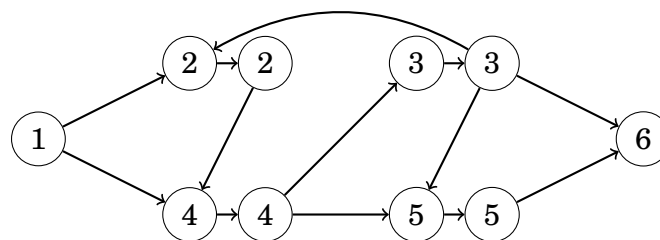
The maximum flow for the graph is as follows:
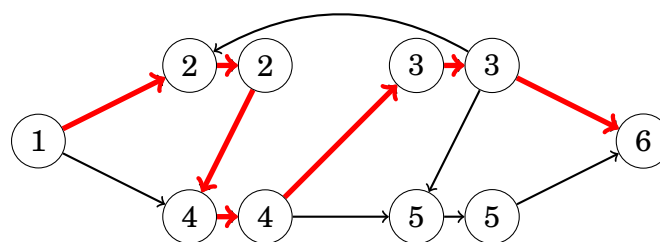
Thus, the maximum number of node-disjoint paths from the source to the sink is 1.

20.3   Maximum matchings

The **maximum matching** problem asks to find a maximum-size set of node pairs in a graph such that each pair is connected with an edge and each node belongs to at most one pair.

There are polynomial algorithms for finding maximum matchings in general graphs [10], but such algorithms are complex and do not appear in programming contests. However, in bipartite graphs, the maximum matching problem is much easier to solve, because we can reduce it to the maximum flow problem.

Finding maximum matchings

The nodes in a bipartite graph can be always divided into two groups such that all edges of the graph go from the left group to the right group. For example, consider the following bipartite graph:
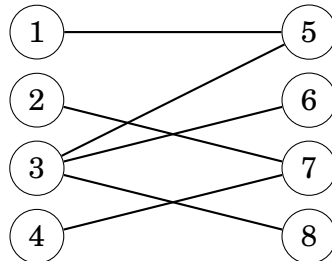

In this graph, the size of a maximum matching is 3:
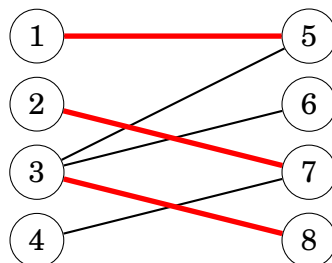
We can reduce the bipartite maximum matching problem to the maximum flow problem by adding two new nodes to the graph: a source and a sink. In addition, we add edges from the source to each left node and from each right node to the sink. After this, the maximum flow of the graph equals the maximum matching of the original graph.

For example, the reduction for the above graph is as follows:
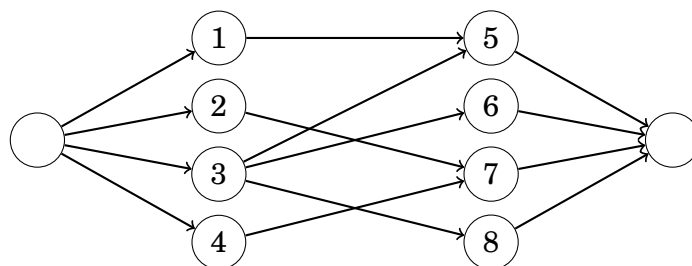


The maximum flow of this graph is as follows:
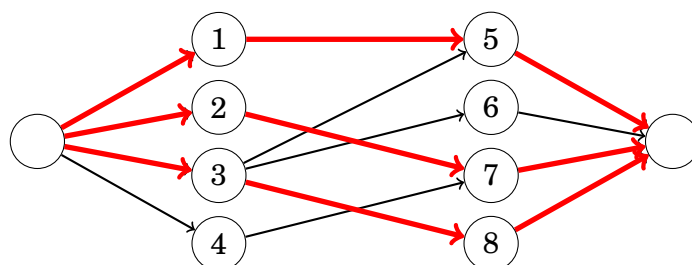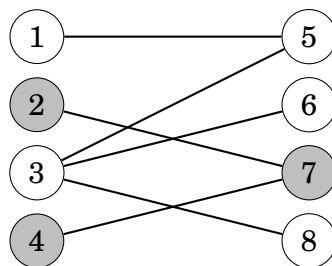
Hall's theorem

Hall's theorem can be used to find out whether a bipartite graph has a matching that contains all left or right nodes. If the number of left and right nodes is the same, Hall's theorem tells us if it is possible to construct a **perfect matching** that contains all nodes of the graph.

Assume that we want to find a matching that contains all left nodes. Let X be any set of left nodes and let $f(X)$ be the set of their neighbors. According to Hall's theorem, a matching that contains all left nodes exists exactly when for each X, the condition $|X| \le |f(X)|$ holds.

Let us study Hall's theorem in the example graph. First, let $X = \{1, 3\}$ and $f(X) = \{5, 6, 8\}$:
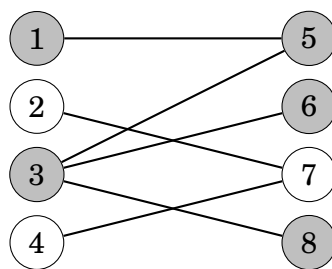
The condition of Hall's theorem holds, because $|X| = 2$ and $|f(X)| = 3$. Next, let $X = \{2, 4\}$ and $f(X) = \{7\}$:

In this case, $|X| = 2$ and $|f(X)| = 1$, so the condition of Hall's theorem does not hold. This means that it is not possible to form a perfect matching in the graph. This result is not surprising, because we already know that the maximum matching of the graph is 3 and not 4.

If the condition of Hall's theorem does not hold, the set X provides an explanation *why* we cannot form such a matching. Since X contains more nodes than $f(X)$, there are no pairs for all nodes in X. For example, in the above graph, both nodes 2 and 4 should be connected with node 7 which is not allowed.

Kőnig's theorem

A **minimum node cover** of a graph is a minimum set of nodes such that each edge of the graph has at least one endpoint in the set. In a general graph, finding a minimum node cover is a NP-hard problem. However, if the graph is bipartite, **Kőnig's theorem** tells us that the size of a minimum node cover and the size

of a maximum matching are always equal. Thus, we can calculate the size of a minimum node cover using a maximum flow algorithm.

Let us consider the following graph with a maximum matching of size 3:
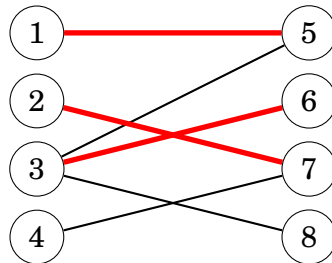
Kőnig's theorem tells us that the size of a minimum node cover is also 3. It can be constructed as follows:
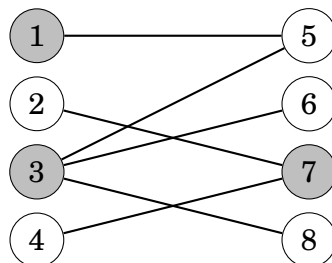
The nodes that do *not* belong to a minimum node cover form a **maximum independent set**. This is the largest possible set of nodes such that no two nodes in the set are connected with an edge. Once again, finding a maximum independent set in a general graph is a NP-hard problem, but in a bipartite graph we can use Kőnig's theorem to solve the problem efficiently. In the example graph, the maximum independent set is as follows:
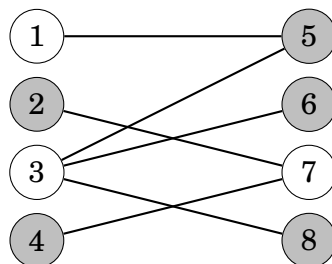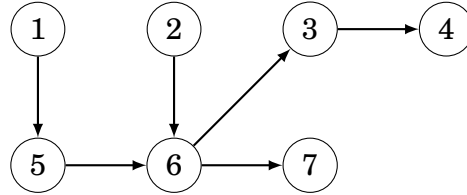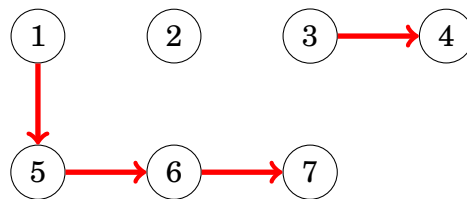

20.4 Path covers

A **path cover** is a set of paths in a graph such that each node of the graph belongs to at least one path. It turns out that in directed, acyclic graphs, we can reduce the problem of finding a minimum path cover to the problem of finding a maximum flow in another graph.

Node-disjoint path cover

In a **node-disjoint path cover**, each node belongs to exactly one path. As an example, consider the following graph:
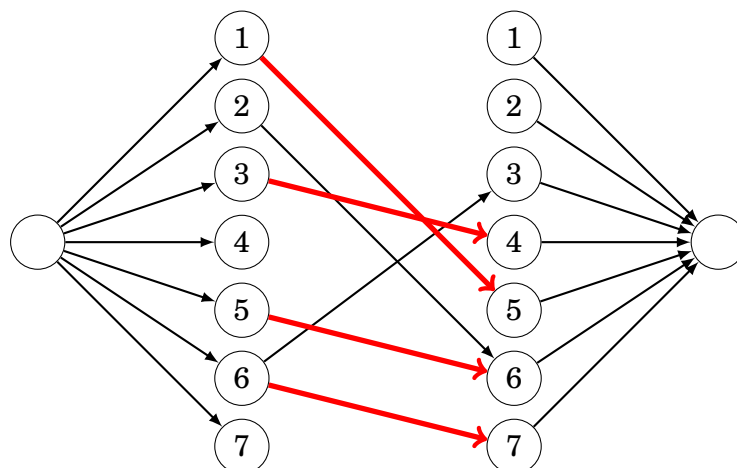

A minimum node-disjoint path cover of this graph consists of three paths. For example, we can choose the following paths:

Note that one of the paths only contains node 2, so it is possible that a path does not contain any edges.

We can find a minimum node-disjoint path cover by constructing a matching graph where each node of the original graph is represented by two nodes: a left node and a right node. There is an edge from a left node to a right node if there is a such an edge in the original graph. In addition, the matching graph contains a source and a sink such that there are edges from the source to all left nodes and from all right nodes to the sink.

A maximum matching in the resulting graph corresponds to a minimum node-disjoint path cover in the original graph. For example, the following graph contains a maximum matching of size 4:


Each edge in the maximum matching of the matching graph corresponds to an edge in the minimum node-disjoint path cover of the original graph. Thus, the size of the minimum node-disjoint path cover is $n - c$, where n is the number of nodes in the original graph and c is the size of the maximum matching.
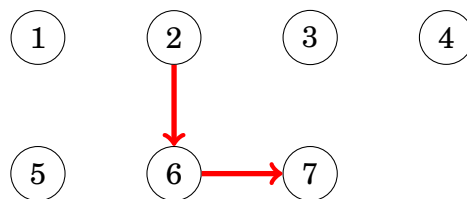
General path cover

A **general path cover** is a path cover where a node can belong to more than one path. A minimum general path cover may be smaller than a minimum node-disjoint path cover, because a node can used multiple times in paths. Consider again the following graph:
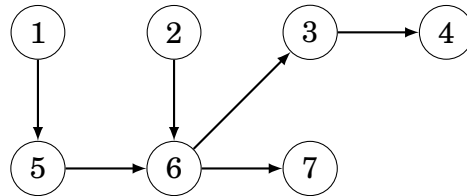
The minimum general path cover in this graph consists of two paths. For example, the first path may be as follows:
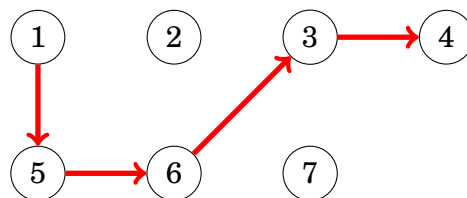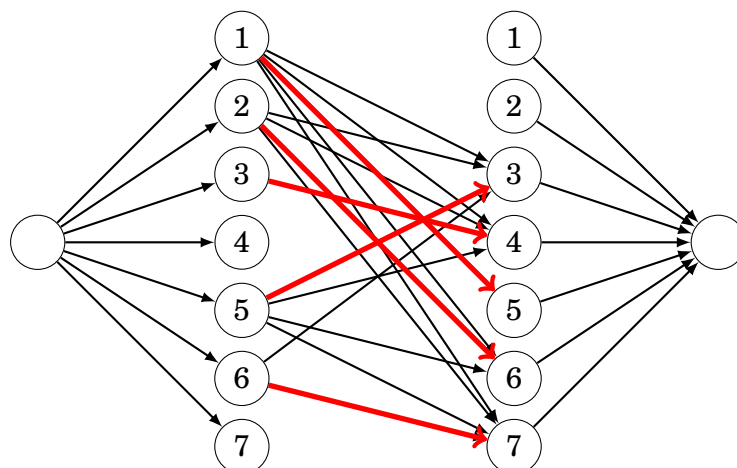

And the second path may be as follows:

A minimum general path cover can be found almost like a minimum node-disjoint path cover. It suffices to add some new edges to the matching graph so that there is an edge $a \rightarrow b$ always when there is a path from a to b in the original graph (possibly through several edges).

The matching graph for the above graph is as follows:

Dilworth's theorem

An **antichain** is a set of nodes of a graph such that there is no path from any node to another node using the edges of the graph. **Dilworth's theorem** states that in a directed acyclic graph, the size of a minimum general path cover equals the size of a maximum antichain.

For example, nodes 3 and 7 form an antichain in the following graph:
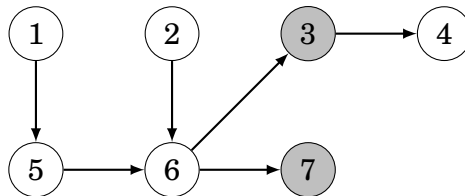
This is a maximum antichain, because it is not possible to construct any antichain that would contain three nodes. We have seen before that the size of a minimum general path cover of this graph consists of two paths.

Part III

Advanced topics

Chapter 21

Number theory

Number theory is a branch of mathematics that studies integers. Number theory is a fascinating field, because many questions involving integers are very difficult to solve even if they seem simple at first glance.

As an example, let us consider the following equation:

$$x^3 + y^3 + z^3 = 33$$

It is easy to find three real numbers x, y and z that satisfy the equation. For example, we can choose

$$x = 3,$$
$$y = \sqrt[3]{3},$$
$$z = \sqrt[3]{3}.$$

However, nobody knows if there are any three *integers* x, y and z that would satisfy the equation, but this is an open problem in number theory.

In this chapter, we will focus on basic concepts and algorithms in number theory. Throughout the chapter, we will assume that all numbers are integers, if not otherwise stated.

21.1   Primes and factors

A number a is a **factor** or **divisor** of a number b if a divides b. If a is a factor of b, we write $a \mid b$, and otherwise we write $a \nmid b$. For example, the factors of the number 24 are 1, 2, 3, 4, 6, 8, 12 and 24.

A number $n > 1$ is a **prime** if its only positive factors are 1 and n. For example, the numbers 7, 19 and 41 are primes. The number 35 is not a prime, because it can be divided into the factors $5 \cdot 7 = 35$. For each number $n > 1$, there is a unique **prime factorization**

$$n = p_1^{\alpha_1} p_2^{\alpha_2} \cdots p_k^{\alpha_k},$$

where $p_1, p_2, \ldots, p_k$ are primes and $\alpha_1, \alpha_2, \ldots, \alpha_k$ are positive numbers. For example, the prime factorization for the number 84 is

$$84 = 2^2 \cdot 3^1 \cdot 7^1.$$

The **number of factors** of a number n is

$$\tau(n) = \prod_{i=1}^{k}(\alpha_i + 1),$$

because for each prime p_i, there are $\alpha_i + 1$ ways to choose how many times it appears in the factor. For example, the number of factors of the number 84 is $\tau(84) = 3 \cdot 2 \cdot 2 = 12$. The factors are 1, 2, 3, 4, 6, 7, 12, 14, 21, 28, 42 and 84.

The **sum of factors** of n is

$$\sigma(n) = \prod_{i=1}^{k}(1 + p_i + \ldots + p_i^{\alpha_i}) = \prod_{i=1}^{k} \frac{p_i^{\alpha_i + 1} - 1}{p_i - 1},$$

where the latter formula is based on the geometric progression formula. For example, the sum of factors of the number 84 is

$$\sigma(84) = \frac{2^3 - 1}{2 - 1} \cdot \frac{3^2 - 1}{3 - 1} \cdot \frac{7^2 - 1}{7 - 1} = 7 \cdot 4 \cdot 8 = 224.$$

The **product of factors** of n is

$$\mu(n) = n^{\tau(n)/2},$$

because we can form $\tau(n)/2$ pairs from the factors, each with product n. For example, the factors of the number 84 produce the pairs $1 \cdot 84$, $2 \cdot 42$, $3 \cdot 28$, etc., and the product of the factors is $\mu(84) = 84^6 = 351298031616$.

A number n is **perfect** if $n = \sigma(n) - n$, i.e., n equals the sum of its factors between 1 and $n - 1$. For example, the number 28 is perfect, because $28 = 1 + 2 + 4 + 7 + 14$.

Number of primes

It is easy to show that there is an infinite number of primes. If the number of primes would be finite, we could construct a set $P = \{p_1, p_2, \ldots, p_n\}$ that would contain all the primes. For example, $p_1 = 2$, $p_2 = 3$, $p_3 = 5$, and so on. However, using P, we could form a new prime

$$p_1 p_2 \cdots p_n + 1$$

that is larger than all elements in P. This is a contradiction, and the number of primes has to be infinite.

Density of primes

The density of primes means how often there are primes among the numbers. Let $\pi(n)$ denote the number of primes between 1 and n. For example, $\pi(10) = 4$, because there are 4 primes between 1 and 10: 2, 3, 5 and 7.

It is possible to show that

$$\pi(n) \approx \frac{n}{\ln n},$$

which means that primes are quite frequent. For example, the number of primes between 1 and 10^6 is $\pi(10^6) = 78498$, and $10^6 / \ln 10^6 \approx 72382$.

Conjectures

There are many *conjectures* involving primes. Most people think that the conjectures are true, but nobody has been able to prove them. For example, the following conjectures are famous:

- **Goldbach's conjecture**: Each even integer $n > 2$ can be represented as a sum $n = a + b$ so that both a and b are primes.

- **Twin prime conjecture**: There is an infinite number of pairs of the form $\{p, p + 2\}$, where both p and $p + 2$ are primes.

- **Legendre's conjecture**: There is always a prime between numbers n^2 and $(n + 1)^2$, where n is any positive integer.

Basic algorithms

If a number n is not prime, it can be represented as a product $a \cdot b$, where $a \leq \sqrt{n}$ or $b \leq \sqrt{n}$, so it certainly has a factor between 2 and $\lfloor \sqrt{n} \rfloor$. Using this observation, we can both test if a number is prime and find the prime factorization of a number in $O(\sqrt{n})$ time.

The following function `prime` checks if the given number n is prime. The function attempts to divide n by all numbers between 2 and $\lfloor \sqrt{n} \rfloor$, and if none of them divides n, then n is prime.

```
bool prime(int n) {
    if (n < 2) return false;
    for (int x = 2; x*x <= n; x++) {
        if (n%x == 0) return false;
    }
    return true;
}
```

The following function `factors` constructs a vector that contains the prime factorization of n. The function divides n by its prime factors, and adds them to the vector. The process ends when the remaining number n has no factors between 2 and $\lfloor \sqrt{n} \rfloor$. If $n > 1$, it is prime and the last factor.

```
vector<int> factors(int n) {
    vector<int> f;
    for (int x = 2; x*x <= n; x++) {
        while (n%x == 0) {
            f.push_back(x);
            n /= x;
        }
    }
    if (n > 1) f.push_back(n);
    return f;
}
```

Note that each prime factor appears in the vector as many times as it divides the number. For example, $24 = 2^3 \cdot 3$, so the result of the function is $[2,2,2,3]$.

Sieve of Eratosthenes

The **sieve of Eratosthenes** is a preprocessing algorithm that builds an array using which we can efficiently check if a given number between $2 \ldots n$ is prime and, if it is not, find one prime factor of the number.

The algorithm builds an array a whose positions $2, 3, \ldots, n$ are used. The value $a[k] = 0$ means that k is prime, and the value $a[k] \neq 0$ means that k is not a prime and one of its prime factors is $a[k]$.

The algorithm iterates through the numbers $2 \ldots n$ one by one. Always when a new prime x is found, the algorithm records that the multiples of x ($2x, 3x, 4x, \ldots$) are not primes, because the number x divides them.

For example, if $n = 20$, the array is as follows:

2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20
0	0	2	0	3	0	2	3	5	0	3	0	7	5	2	0	3	0	5

The following code implements the sieve of Eratosthenes. The code assumes that each element in a is initially zero.

```
for (int x = 2; x <= n; x++) {
    if (a[x]) continue;
    for (int u = 2*x; u <= n; u += x) {
        a[u] = x;
    }
}
```

The inner loop of the algorithm will be executed n/x times for any x. Thus, an upper bound for the running time of the algorithm is the harmonic sum

$$\sum_{x=2}^{n} n/x = n/2 + n/3 + n/4 + \cdots + n/n = O(n \log n).$$

In fact, the algorithm is even more efficient, because the inner loop will be executed only if the number x is prime. It can be shown that the time complexity of the algorithm is only $O(n \log \log n)$, a complexity very near to $O(n)$.

Euclid's algorithm

The **greatest common divisor** of numbers a and b, $\gcd(a, b)$, is the greatest number that divides both a and b, and the **least common multiple** of a and b, $\text{lcm}(a, b)$, is the smallest number that is divisible by both a and b. For example, $\gcd(24, 36) = 12$ and $\text{lcm}(24, 36) = 72$.

The greatest common divisor and the least common multiple are connected as follows:
$$\text{lcm}(a, b) = \frac{ab}{\gcd(a, b)}$$

Euclid's algorithm provides an efficient way to find the greatest common divisor of two numbers. The algorithm is based on the following formula:

$$\gcd(a,b) = \begin{cases} a & b = 0 \\ \gcd(b, a \bmod b) & b \neq 0 \end{cases}$$

For example,

$$\gcd(24, 36) = \gcd(36, 24) = \gcd(24, 12) = \gcd(12, 0) = 12.$$

The time complexity of Euclid's algorithm is $O(\log n)$, where $n = \min(a, b)$. The worst case for the algorithm is the case when a and b are consecutive Fibonacci numbers. For example,

$$\gcd(13, 8) = \gcd(8, 5) = \gcd(5, 3) = \gcd(3, 2) = \gcd(2, 1) = \gcd(1, 0) = 1.$$

Euler's totient function

Numbers a and b are **coprime** if $\gcd(a, b) = 1$. **Euler's totient function** $\varphi(n)$ gives the number of coprime numbers to n between 1 and n. For example, $\varphi(12) = 4$, because 1, 5, 7 and 11 are coprime to 12.

The value of $\varphi(n)$ can be calculated from the prime factorization of n using the formula

$$\varphi(n) = \prod_{i=1}^{k} p_i^{\alpha_i - 1}(p_i - 1).$$

For example, $\varphi(12) = 2^1 \cdot (2 - 1) \cdot 3^0 \cdot (3 - 1) = 4$. Note that $\varphi(n) = n - 1$ if n is prime.

21.2 Modular arithmetic

In **modular arithmetic**, the set of available numbers is limited so that only numbers $0, 1, 2, \ldots, m - 1$ may be used, where m is a constant. Each number x is represented by the number $x \bmod m$: the remainder after dividing x by m. For example, if $m = 17$, then 75 is represented by $75 \bmod 17 = 7$.

Often we can take the remainder before doing calculations. In particular, the following formulas can be used:

$$\begin{aligned} (x + y) \bmod m &= (x \bmod m + y \bmod m) \bmod m \\ (x - y) \bmod m &= (x \bmod m - y \bmod m) \bmod m \\ (x \cdot y) \bmod m &= (x \bmod m \cdot y \bmod m) \bmod m \\ x^n \bmod m &= (x \bmod m)^n \bmod m \end{aligned}$$

Modular exponentiation

There is often need to efficiently calculate the value of $x^n \bmod m$. This can be done in $O(\log n)$ time using the following recursion:

$$x^n = \begin{cases} 1 & n = 0 \\ x^{n/2} \cdot x^{n/2} & n \text{ is even} \\ x^{n-1} \cdot x & n \text{ is odd} \end{cases}$$

It is important that in the case of an even n, the value of $x^{n/2}$ is calculated only once. This guarantees that the time complexity of the algorithm is $O(\log n)$, because n is always halved when it is even.

The following function calculates the value of $x^n \bmod m$:

```
int modpow(int x, int n, int m) {
    if (n == 0) return 1%m;
    int u = modpow(x,n/2,m);
    u = (u*u)%m;
    if (n%2 == 1) u = (u*x)%m;
    return u;
}
```

Fermat's theorem and Euler's theorem

Fermat's theorem states that

$$x^{m-1} \bmod m = 1$$

when m is prime and x and m are coprime. This also yields

$$x^k \bmod m = x^{k \bmod (m-1)} \bmod m.$$

More generally, **Euler's theorem** states that

$$x^{\varphi(m)} \bmod m = 1$$

when x and m are coprime. Fermat's theorem follows from Euler's theorem, because if m is a prime, then $\varphi(m) = m - 1$.

Modular inverse

The inverse of x modulo m is a number x^{-1} such that

$$xx^{-1} \bmod m = 1.$$

For example, if $x = 6$ and $m = 17$, then $x^{-1} = 3$, because $6 \cdot 3 \bmod 17 = 1$.

Using modular inverses, we can divide numbers modulo m, because division by x corresponds to multiplication by x^{-1}. For example, to evaluate the value of $36/6 \bmod 17$, we can use the formula $2 \cdot 3 \bmod 17$, because $36 \bmod 17 = 2$ and $6^{-1} \bmod 17 = 3$.

However, a modular inverse does not always exist. For example, if $x = 2$ and $m = 4$, the equation

$$xx^{-1} \bmod m = 1$$

cannot be solved, because all multiples of the number 2 are even and the remainder can never be 1 when $m = 4$. It turns out that the value of $x^{-1} \bmod m$ can be calculated exactly when x and m are coprime.

If a modular inverse exists, it can be calculated using the formula

$$x^{-1} = x^{\varphi(m)-1}.$$

If m is prime, the formula becomes

$$x^{-1} = x^{m-2}.$$

For example, if $x = 6$ and $m = 17$, then

$$x^{-1} = 6^{17-2} \bmod 17 = 3.$$

Using this formula, we can calculate the modular inverse efficiently using the modular exponentation algorithm.

The above formula can be derived using Euler's theorem. First, the modular inverse should satisfy the following equation:

$$xx^{-1} \bmod m = 1.$$

On the other hand, according to Euler's theorem,

$$x^{\varphi(m)} \bmod m = xx^{\varphi(m)-1} \bmod m = 1,$$

so the numbers x^{-1} and $x^{\varphi(m)-1}$ are equal.

Computer arithmetic

In programming, unsigned integers are represented modulo 2^k, where k is the number of bits of the data type. A usual consequence of this is that a number wraps around if it becomes too large.

For example, in C++, numbers of type `unsigned int` are represented modulo 2^{32}. The following code declares an `unsigned int` variable whose value is 123456789. After this, the value will be multiplied by itself, and the result is $123456789^2 \bmod 2^{32} = 2537071545$.

```
unsigned int x = 123456789;
cout << x*x << "\n"; // 2537071545
```

21.3   Solving equations

A **Diophantine equation** is an equation of the form

$$ax + by = c,$$

where a, b and c are constants and we should find the values of x and y. Each number in the equation has to be an integer. For example, one solution for the equation $5x + 2y = 11$ is $x = 3$ and $y = -2$.

We can efficiently solve a Diophantine equation by using Euclid's algorithm. It turns out that we can extend Euclid's algorithm so that it will find numbers x and y that satisfy the following equation:

$$ax + by = \gcd(a, b)$$

A Diophantine equation can be solved if c is divisible by $\gcd(a, b)$, and otherwise the equation cannot be solved.

Extended Euclid's algorithm

As an example, let us find numbers x and y that satisfy the following equation:

$$39x + 15y = 12$$

The equation can be solved, because $\gcd(39, 15) = 3$ and $3 \mid 12$. When Euclid's algorithm calculates the greatest common divisor of 39 and 15, it produces the following sequence of function calls:

$$\gcd(39, 15) = \gcd(15, 9) = \gcd(9, 6) = \gcd(6, 3) = \gcd(3, 0) = 3$$

This corresponds to the following equations:

$$
\begin{aligned}
39 - 2 \cdot 15 &= 9 \\
15 - 1 \cdot 9 &= 6 \\
9 - 1 \cdot 6 &= 3
\end{aligned}
$$

Using these equations, we can derive

$$39 \cdot 2 + 15 \cdot (-5) = 3$$

and by multiplying this by 4, the result is

$$39 \cdot 8 + 15 \cdot (-20) = 12,$$

so a solution to the equation is $x = 8$ and $y = -20$.

A solution to a Diophantine equation is not unique, but we can form an infinite number of solutions if we know one solution. If a pair (x, y) is a solution, then also all pairs

$$\left(x + \frac{kb}{\gcd(a, b)}, y - \frac{ka}{\gcd(a, b)}\right)$$

are solutions, where k is any integer.

Chinese remainder theorem

The **Chinese remainder theorem** solves a group of equations of the form

$$
\begin{aligned}
x &= a_1 \bmod m_1 \\
x &= a_2 \bmod m_2 \\
&\ldots \\
x &= a_n \bmod m_n
\end{aligned}
$$

where all pairs of $m_1, m_2, \ldots, m_n$ are coprime.

Let x_m^{-1} be the inverse of x modulo m, and

$$X_k = \frac{m_1 m_2 \cdots m_n}{m_k}.$$

Using this notation, a solution to the equations is

$$x = a_1 X_1 X_{1_{m_1}}^{-1} + a_2 X_2 X_{2_{m_2}}^{-1} + \cdots + a_n X_n X_{n_{m_n}}^{-1}.$$

In this solution, it holds for each number $k = 1, 2, \ldots, n$ that

$$a_k X_k X_{k_{m_k}}^{-1} \bmod m_k = a_k,$$

because

$$X_k X_{k_{m_k}}^{-1} \bmod m_k = 1.$$

Since all other terms in the sum are divisible by m_k, they have no effect on the remainder, and the remainder by m_k for the whole sum is a_k.

For example, a solution for

$$
\begin{aligned}
x &= 3 \bmod 5 \\
x &= 4 \bmod 7 \\
x &= 2 \bmod 3
\end{aligned}
$$

is

$$3 \cdot 21 \cdot 1 + 4 \cdot 15 \cdot 1 + 2 \cdot 35 \cdot 2 = 263.$$

Once we have found a solution x, we can create an infinite number of other solutions, because all numbers of the form

$$x + m_1 m_2 \cdots m_n$$

are solutions.

21.4 Other results

Lagrange's theorem

Lagrange's theorem states that every positive integer can be represented as a sum of four squares, i.e., $a^2 + b^2 + c^2 + d^2$. For example, the number 123 can be represented as the sum $8^2 + 5^2 + 5^2 + 3^2$.

Zeckendorf's theorem

Zeckendorf's theorem states that every positive integer has a unique representation as a sum of Fibonacci numbers such that no two numbers are equal or consecutive Fibonacci numbers. For example, the number 74 can be represented as the sum $55 + 13 + 5 + 1$.

Pythagorean triples

A **Pythagorean triple** is a triple (a,b,c) that satisfies the Pythagorean theorem $a^2 + b^2 = c^2$, which means that there is a right triangle with side lengths a, b and c. For example, $(3,4,5)$ is a Pythagorean triple.

If (a,b,c) is a Pythagorean triple, all triples of the form (ka,kb,kc) are also Pythagorean triples where $k > 1$. A Pythagorean triple is **primitive** if a, b and c are coprime, and all Pythagorean triples can be constructed from primitive triples using a multiplier k.

Euclid's formula can be used to produce all primitive Pythagorean triples. Each such triple is of the form

$$(n^2 - m^2, 2nm, n^2 + m^2),$$

where $0 < m < n$, n and m are coprime and at least one of n and m is even. For example, when $m = 1$ and $n = 2$, the formula produces the smallest Pythagorean triple

$$(2^2 - 1^2, 2 \cdot 2 \cdot 1, 2^2 + 1^2) = (3,4,5).$$

Wilson's theorem

Wilson's theorem states that a number n is prime exactly when

$$(n-1)! \bmod n = n - 1.$$

For example, the number 11 is prime, because

$$10! \bmod 11 = 10,$$

and the number 12 is not prime, because

$$11! \bmod 12 = 0 \neq 11.$$

Hence, Wilson's theorem can be used to find out whether a number is prime. However, in practice, the theorem cannot be applied to large values of n, because it is difficult to calculate the value of $(n-1)!$ when n is large.

Chapter 22

Combinatorics

Combinatorics studies methods for counting combinations of objects. Usually, the goal is to find a way to count the combinations efficiently without generating each combination separately.

As an example, let us consider the problem of counting the number of ways to represent an integer n as a sum of positive integers. For example, there are 8 representations for the number 4:

- $1+1+1+1$
- $1+1+2$
- $1+2+1$
- $2+1+1$

- $2+2$
- $3+1$
- $1+3$
- 4

A combinatorial problem can often be solved using a recursive function. In this problem, we can define a function $f(n)$ that gives the number of representations for n. For example, $f(4) = 8$ according to the above example. The values of the function can be recursively calculated as follows:

$$f(n) = \begin{cases} 1 & n = 1 \\ f(1) + f(2) + \ldots + f(n-1) + 1 & n > 1 \end{cases}$$

The base case is $f(1) = 1$, because there is only one way to represent the number 1. When $n > 1$, we go through all ways to choose the last number in the sum. For example, in when $n = 4$, the sum can end with $+1$, $+2$ or $+3$. In addition, we also count the representation that only contains n.

The first values for the function are:

$$
\begin{aligned}
f(1) &= 1 \\
f(2) &= 2 \\
f(3) &= 4 \\
f(4) &= 8 \\
f(5) &= 16
\end{aligned}
$$

It turns out that the function also has a closed-form formula

$$f(n) = 2^{n-1},$$

which is based on the fact that there are $n-1$ possible positions for +-signs in the sum and we can choose any subset of them.

22.1 Binomial coefficients

The **binomial coefficient** $\binom{n}{k}$ equals the number of ways we can choose a subset of k elements from a set of n elements. For example, $\binom{5}{3} = 10$, because the set $\{1,2,3,4,5\}$ has 10 subsets of 3 elements:

$$\{1,2,3\}, \{1,2,4\}, \{1,2,5\}, \{1,3,4\}, \{1,3,5\}, \{1,4,5\}, \{2,3,4\}, \{2,3,5\}, \{2,4,5\}, \{3,4,5\}$$

Formula 1

Binomial coefficients can be recursively calculated as follows:

$$\binom{n}{k} = \binom{n-1}{k-1} + \binom{n-1}{k}$$

The idea is to fix an element x in the set. If x is included in the subset, we have to choose $k-1$ elements from $n-1$ elements, and if x is not included in the subset, we have to choose k elements from $n-1$ elements.

The base cases for the recursion are

$$\binom{n}{0} = \binom{n}{n} = 1,$$

because there is always exactly one way to construct an empty subset and a subset that contains all the elements.

Formula 2

Another way to calculate binomial coefficients is as follows:

$$\binom{n}{k} = \frac{n!}{k!(n-k)!}.$$

There are $n!$ permutations of n elements. We go through all permutations and always select the first k elements of the permutation to the subset. Since the order of the elements in the subset and outside the subset does not matter, the result is divided by $k!$ and $(n-k)!$

Properties

For binomial coefficients,

$$\binom{n}{k} = \binom{n}{n-k},$$

because we can either select k elements that belong to the subset or $n-k$ elements that do not belong to the subset.

The sum of binomial coefficients is

$$\binom{n}{0} + \binom{n}{1} + \binom{n}{2} + \ldots + \binom{n}{n} = 2^n.$$

The reason for the name "binomial coefficient" is that

$$(a+b)^n = \binom{n}{0}a^n b^0 + \binom{n}{1}a^{n-1}b^1 + \ldots + \binom{n}{n-1}a^1 b^{n-1} + \binom{n}{n}a^0 b^n.$$

Binomial coefficients also appear in **Pascal's triangle** where each value equals the sum of two above values:

```
                1
             1     1
          1     2     1
       1     3     3     1
    1     4     6     4     1
   ...   ...   ...   ...   ...
```

Boxes and balls

"Boxes and balls" is a useful model, where we count the ways to place k balls in n boxes. Let us consider three scenarios:

Scenario 1: Each box can contain at most one ball. For example, when $n = 5$ and $k = 2$, there are 10 solutions:
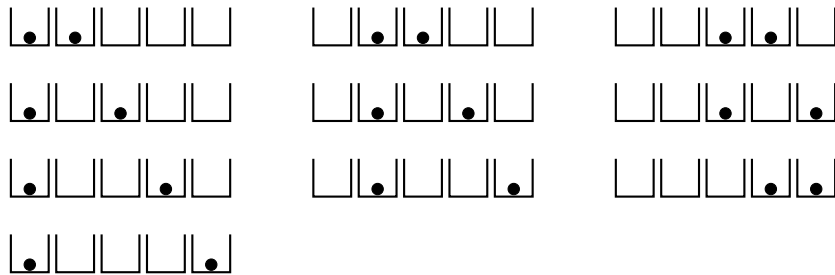
In this scenario, the answer is directly the binomial coefficient $\binom{n}{k}$.

Scenario 2: A box can contain multiple balls. For example, when $n = 5$ and $k = 2$, there are 15 solutions:
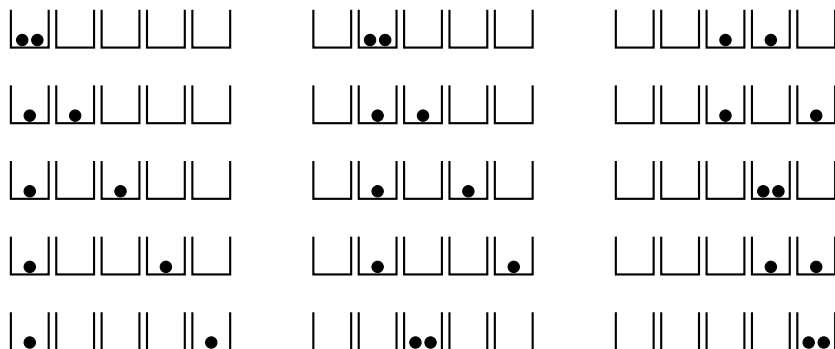
The process of placing the balls in the boxes can be represented as a string that consists of symbols "o" and "→". Initially, assume that we are standing at the leftmost box. The symbol "o" means that we place a ball in the current box, and the symbol "→" means that we move to the next box to the right.

Using this notation, each solution is a string that contains k times the symbol "o" and $n-1$ times the symbol "→". For example, the upper-right solution in the above picture corresponds to the string "→ → o → o →". Thus, the number of solutions is $\binom{k+n-1}{k}$.

Scenario 3: Each box may contain at most one ball, and in addition, no two adjacent boxes may both contain a ball. For example, when $n = 5$ and $k = 2$, there are 6 solutions:
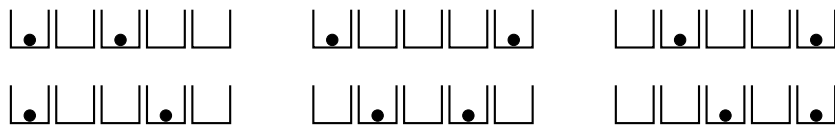
In this scenario, we can assume that k balls are initially placed in boxes and there is an empty box between each two such boxes. The remaining task is to choose the positions for $n - k - (k-1) = n - 2k + 1$ empty boxes. There are $k + 1$ positions, so the number of solutions is $\binom{n-2k+1+k+1-1}{n-2k+1} = \binom{n-k+1}{n-2k+1}$.

Multinomial coefficients

The **multinomial coefficient**

$$\binom{n}{k_1, k_2, \ldots, k_m} = \frac{n!}{k_1! k_2! \cdots k_m!},$$

equals the number of ways we can divide n elements into subsets of sizes $k_1, k_2, \ldots, k_m$, where $k_1 + k_2 + \cdots + k_m = n$. Multinomial coefficients can be seen as a generalization of binomial cofficients; if $m = 2$, the above formula corresponds to the binomial coefficient formula.

22.2 Catalan numbers

The **Catalan number** C_n equals the number of valid parenthesis expressions that consist of n left parentheses and n right parentheses.

For example, $C_3 = 5$, because using three left parentheses and three right parentheses, we can construct the following parenthesis expressions:

- ()()()
- (())()
- ()(())
- ((()))
- (()())

Parenthesis expressions

What is exactly a *valid parenthesis expression*? The following rules precisely define all valid parenthesis expressions:

- The empty expression is valid.

- If an expression A is valid, then also the expression (A) is valid.

- If expressions A and B are valid, then also the expression AB is valid.

Another way to characterize valid parenthesis expressions is that if we choose any prefix of such an expression, it has to contain at least as many left parentheses as right parentheses. In addition, the complete expression has to contain an equal number of left and right parentheses.

Formula 1

Catalan numbers can be calculated using the formula

$$C_n = \sum_{i=0}^{n-1} C_i C_{n-i-1}.$$

The sum goes through the ways to divide the expression into two parts such that both parts are valid expressions and the first part is as short as possible but not empty. For any i, the first part contains $i+1$ pairs of parentheses and the number of expressions is the product of the following values:

- C_i: number of ways to construct an expression using the parentheses in the first part, not counting the outermost parentheses

- C_{n-i-1}: number of ways to construct an expression using the parentheses in the second part

In addition, the base case is $C_0 = 1$, because we can construct an empty parenthesis expression using zero pairs of parentheses.

Formula 2

Catalan numbers can also be calculated using binomial coefficients:

$$C_n = \frac{1}{n+1}\binom{2n}{n}$$

The formula can be explained as follows:

There are a total of $\binom{2n}{n}$ ways to construct a (not necessarily valid) parenthesis expression that contains n left parentheses and n right parentheses. Let us calculate the number of such expressions that are *not* valid.

If a parenthesis expression is not valid, it has to contain a prefix where the number of right parentheses exceeds the number of left parentheses. The

idea is to reverse each parenthesis that belongs to such a prefix. For example, the expression ())()(contains a prefix ()), and after reversing the prefix, the expression becomes)((()(.

The resulting expression consists of $n+1$ left parentheses and $n-1$ right parentheses. The number of such expressions is $\binom{2n}{n+1}$ that equals the number of non-valid parenthesis expressions. Thus the number of valid parenthesis expressions can be calculated using the formula
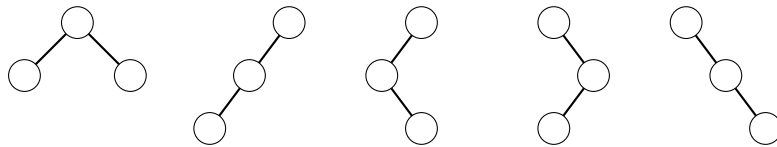
$$\binom{2n}{n} - \binom{2n}{n+1} = \binom{2n}{n} - \frac{n}{n+1}\binom{2n}{n} = \frac{1}{n+1}\binom{2n}{n}.$$

Counting trees

Catalan numbers are also related to trees:

- there are C_n binary trees of n nodes

- there are C_{n-1} rooted trees of n nodes

For example, for $C_3 = 5$, the binary trees are

and the rooted trees are
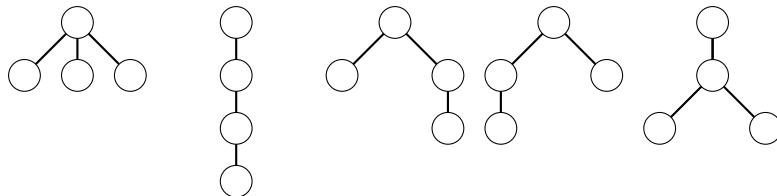
22.3   Inclusion-exclusion

Inclusion-exclusion is a technique that can be used for counting the size of a union of sets when the sizes of the intersections are known, and vice versa. A simple example of the technique is the formula

$$|A \cup B| = |A| + |B| - |A \cap B|,$$

where A and B are sets and $|X|$ is the size of a set X. The formula can be illustrated as follows:
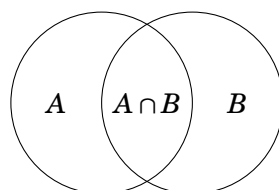
Our goal is to calculate the size of the union $A \cup B$ that corresponds to the area of the region that belongs to at least one circle. The picture shows that we can calculate the area of $A \cup B$ by first summing the areas of A and B and then subtracting the area of $A \cap B$.

The same idea can be applied when the number of sets is larger. When there are three sets, the inclusion-exclusion formula is

$$|A \cup B \cup C| = |A| + |B| + |C| - |A \cap B| - |A \cap C| - |B \cap C| + |A \cap B \cap C|$$

and the corresponding picture is
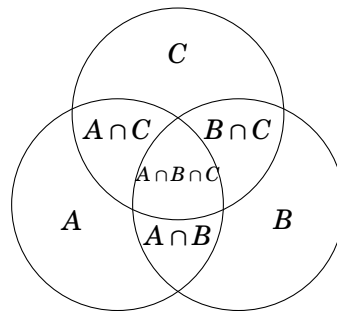
In the general case, the size of the union $X_1 \cup X_2 \cup \cdots \cup X_n$ can be calculated by going through all possible intersections that contain some of the sets $X_1, X_2, \ldots, X_n$. If the intersection contains an odd number of sets, its size is added to the answer, and otherwise its size is subtracted from the answer.

Note that there are similar formulas for calculating the size of an intersection from the sizes of unions. For example,

$$|A \cap B| = |A| + |B| - |A \cup B|$$

and

$$|A \cap B \cap C| = |A| + |B| + |C| - |A \cup B| - |A \cup C| - |B \cup C| + |A \cup B \cup C|.$$

Derangements

As an example, let us count the number of **derangements** of elements $\{1, 2, \ldots, n\}$, i.e., permutations where no element remains in its original place. For example, when $n = 3$, there are two possible derangements: $(2, 3, 1)$ and $(3, 1, 2)$.

One approach for solving the problem is to use inclusion-exclusion. Let X_k be the set of permutations that contain the element k at position k. For example, when $n = 3$, the sets are as follows:

$$
\begin{aligned}
X_1 &= \{(1,2,3),(1,3,2)\} \\
X_2 &= \{(1,2,3),(3,2,1)\} \\
X_3 &= \{(1,2,3),(2,1,3)\}
\end{aligned}
$$

Using these sets, the number of derangements equals

$$n! - |X_1 \cup X_2 \cup \cdots \cup X_n|,$$

so it suffices to calculate the size of the union. Using inclusion-exclusion, this reduces to calculating sizes of intersections which can be done efficiently. For example, when $n = 3$, the size of $|X_1 \cup X_2 \cup X_3|$ is

$$
\begin{aligned}
& |X_1| + |X_2| + |X_3| - |X_1 \cap X_2| - |X_1 \cap X_3| - |X_2 \cap X_3| + |X_1 \cap X_2 \cap X_3| \\
= \ & 2 + 2 + 2 - 1 - 1 - 1 + 1 \\
= \ & 4,
\end{aligned}
$$

so the number of solutions is $3! - 4 = 2$.

It turns out that the problem can also be solved without using inclusion-exclusion. Let $f(n)$ denote the number of derangements for $\{1, 2, \ldots, n\}$. We can use the following recursive formula:

$$
f(n) = \begin{cases} 0 & n = 1 \\ 1 & n = 2 \\ (n-1)(f(n-2) + f(n-1)) & n > 2 \end{cases}
$$

The formula can be derived by going through the possibilities how the element 1 changes in the derangement. There are $n - 1$ ways to choose an element x that replaces the element 1. In each such choice, there are two options:

Option 1: We also replace the element x with the element 1. After this, the remaining task is to construct a derangement of $n - 2$ elements.

Option 2: We replace the element x with some other element than 1. Now we have to construct a derangement of $n - 1$ element, because we cannot replace the element x with the element 1, and all other elements should be changed.

22.4   Burnside's lemma

Burnside's lemma can be used to count the number of combinations so that only one representative is counted for each group of symmetric combinations. Burnside's lemma states that the number of combinations is

$$
\sum_{k=1}^{n} \frac{c(k)}{n},
$$

where there are n ways to change the position of a combination, and there are $c(k)$ combinations that remain unchanged when the kth way is applied.

As an example, let us calculate the number of necklaces of n pearls, where the color of each pearl is one of $1, 2, \ldots, m$. Two necklaces are symmetric if they are similar after rotating them. For example, the necklace
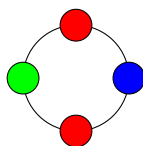
has the following symmetric necklaces:
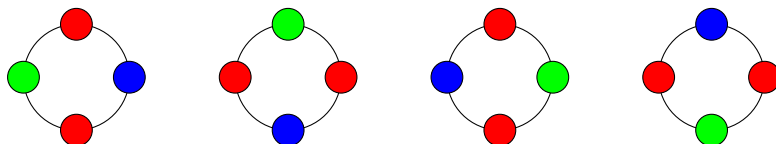
There are n ways to change the position of a necklace, because we can rotate it $0, 1, \ldots, n-1$ steps clockwise. If the number of steps is 0, all m^n necklaces remain the same, and if the number of steps is 1, only the m necklaces where each pearl has the same color remain the same.

More generally, when the number of steps is k, a total of

$$m^{\gcd(k,n)},$$

necklaces remain the same, where $\gcd(k,n)$ is the greatest common divisor of k and n. The reason for this is that blocks of pearls of size $\gcd(k,n)$ will replace each other. Thus, according to Burnside's lemma, the number of necklaces is

$$\sum_{i=0}^{n-1} \frac{m^{\gcd(i,n)}}{n}.$$

For example, the number of necklaces of length 4 with 3 colors is

$$\frac{3^4 + 3 + 3^2 + 3}{4} = 24.$$

22.5 Cayley's formula

Cayley's formula states that there are n^{n-2} labeled trees that contain n nodes. The nodes are labeled $1, 2, \ldots, n$, and two trees are different if either their structure or labeling is different.

For example, when $n = 4$, the number of labeled trees is $4^{4-2} = 16$:
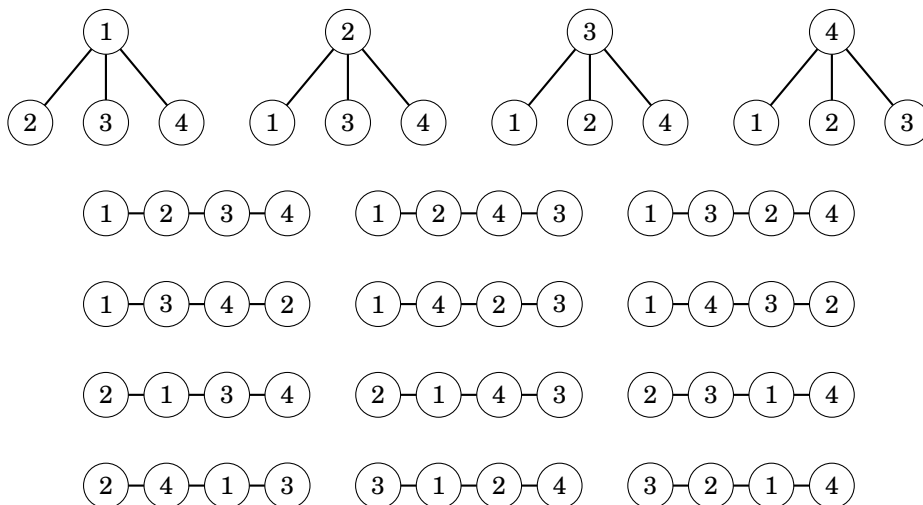
Next we will see how Cayley's formula can be derived using Prüfer codes.

Prüfer code

A **Prüfer code** is a sequence of $n-2$ numbers that describes a labeled tree. The code is constructed by following a process that removes $n-2$ leaves from the tree. At each step, the leaf with the smallest label is removed, and the label of its only neighbor is added to the code.

For example, the Prüfer code for
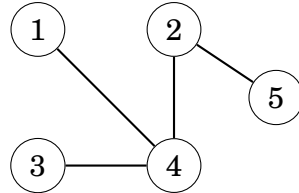
is $[4, 4, 2]$, because we first remove node 1, then node 3 and finally node 5.

We can construct a Prüfer code for any tree, and more importantly, the original tree can be reconstructed from a Prüfer code. Hence, the number of labeled trees of n nodes equals n^{n-2}, the number of Prüfer codes of size n.

Chapter 23

Matrices

A **matrix** is a mathematical concept that corresponds to a two-dimensional array in programming. For example,

$$A = \begin{bmatrix} 6 & 13 & 7 & 4 \\ 7 & 0 & 8 & 2 \\ 9 & 5 & 4 & 18 \end{bmatrix}$$

is a matrix of size 3×4, i.e., it has 3 rows and 4 columns. The notation $[i,j]$ refers to the element in row i and column j in a matrix. For example, in the above matrix, $A[2,3] = 8$ and $A[3,1] = 9$.

A special case of a matrix is a **vector** that is a one-dimensional matrix of size $n \times 1$. For example,

$$V = \begin{bmatrix} 4 \\ 7 \\ 5 \end{bmatrix}$$

is a vector that contains three elements.

The **transpose** A^T of a matrix A is obtained when the rows and columns in A are swapped, i.e., $A^T[i,j] = A[j,i]$:

$$A^T = \begin{bmatrix} 6 & 7 & 9 \\ 13 & 0 & 5 \\ 7 & 8 & 4 \\ 4 & 2 & 18 \end{bmatrix}$$

A matrix is a **square matrix** if it has the same number of rows and columns. For example, the following matrix is a square matrix:

$$S = \begin{bmatrix} 3 & 12 & 4 \\ 5 & 9 & 15 \\ 0 & 2 & 4 \end{bmatrix}$$

23.1 Operations

The sum $A + B$ of matrices A and B is defined if the matrices are of the same size. The result is a matrix where each element is the sum of the corresponding elements in A and B.

For example,

$$\begin{bmatrix} 6 & 1 & 4 \\ 3 & 9 & 2 \end{bmatrix} + \begin{bmatrix} 4 & 9 & 3 \\ 8 & 1 & 3 \end{bmatrix} = \begin{bmatrix} 6+4 & 1+9 & 4+3 \\ 3+8 & 9+1 & 2+3 \end{bmatrix} = \begin{bmatrix} 10 & 10 & 7 \\ 11 & 10 & 5 \end{bmatrix}.$$

Multiplying a matrix A by a value x means that each element of A is multiplied by x. For example,

$$2 \cdot \begin{bmatrix} 6 & 1 & 4 \\ 3 & 9 & 2 \end{bmatrix} = \begin{bmatrix} 2 \cdot 6 & 2 \cdot 1 & 2 \cdot 4 \\ 2 \cdot 3 & 2 \cdot 9 & 2 \cdot 2 \end{bmatrix} = \begin{bmatrix} 12 & 2 & 8 \\ 6 & 18 & 4 \end{bmatrix}.$$

Matrix multiplication

The product AB of matrices A and B is defined if A is of size $a \times n$ and B is of size $n \times b$, i.e., the width of A equals the height of B. The result is a matrix of size $a \times b$ whose elements are calculated using the formula

$$AB[i,j] = \sum_{k=1}^{n} A[i,k] \cdot B[k,j].$$

The idea is that each element in AB is a sum of products of elements in A and B according to the following picture:
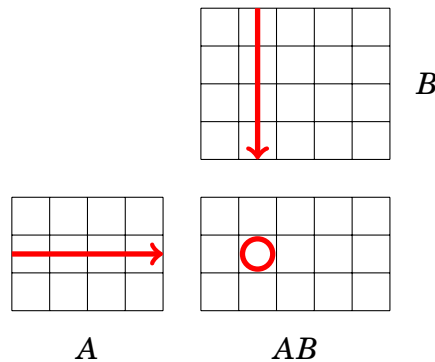
For example,

$$\begin{bmatrix} 1 & 4 \\ 3 & 9 \\ 8 & 6 \end{bmatrix} \cdot \begin{bmatrix} 1 & 6 \\ 2 & 9 \end{bmatrix} = \begin{bmatrix} 1 \cdot 1 + 4 \cdot 2 & 1 \cdot 6 + 4 \cdot 9 \\ 3 \cdot 1 + 9 \cdot 2 & 3 \cdot 6 + 9 \cdot 9 \\ 8 \cdot 1 + 6 \cdot 2 & 8 \cdot 6 + 6 \cdot 9 \end{bmatrix} = \begin{bmatrix} 9 & 42 \\ 21 & 99 \\ 20 & 102 \end{bmatrix}.$$

Matrix multiplication is associative, so $A(BC) = (AB)C$ holds, but it is not commutative, so $AB = BA$ does not usually hold.

An **identity matrix** is a square matrix where each element on the diagonal is 1 and all other elements are 0. For example, the following matrix is the 3×3 identity matrix:

$$I = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix}.$$

Multiplying a matrix by an identity matrix does not change it. For example,

$$
\begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix} \cdot \begin{bmatrix} 1 & 4 \\ 3 & 9 \\ 8 & 6 \end{bmatrix} = \begin{bmatrix} 1 & 4 \\ 3 & 9 \\ 8 & 6 \end{bmatrix} \quad \text{and} \quad \begin{bmatrix} 1 & 4 \\ 3 & 9 \\ 8 & 6 \end{bmatrix} \cdot \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix} = \begin{bmatrix} 1 & 4 \\ 3 & 9 \\ 8 & 6 \end{bmatrix}.
$$

Using a straightforward algorithm, we can calculate the product of two $n \times n$ matrices in $O(n^3)$ time. There are also more efficient algorithms for matrix multiplication[1], but they are mostly of theoretical interest and such special algorithms are not needed in competitive programming.

Matrix power

The power A^k of a matrix A is defined if A is a square matrix. The definition is based on matrix multiplication:

$$
A^k = \underbrace{A \cdot A \cdot A \cdots A}_{k \text{ times}}
$$

For example,

$$
\begin{bmatrix} 2 & 5 \\ 1 & 4 \end{bmatrix}^3 = \begin{bmatrix} 2 & 5 \\ 1 & 4 \end{bmatrix} \cdot \begin{bmatrix} 2 & 5 \\ 1 & 4 \end{bmatrix} \cdot \begin{bmatrix} 2 & 5 \\ 1 & 4 \end{bmatrix} = \begin{bmatrix} 48 & 165 \\ 33 & 114 \end{bmatrix}.
$$

In addition, A^0 is an identity matrix. For example,

$$
\begin{bmatrix} 2 & 5 \\ 1 & 4 \end{bmatrix}^0 = \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix}.
$$

The matrix A^k can be efficiently calculated in $O(n^3 \log k)$ time using the algorithm in Chapter 21.2. For example,

$$
\begin{bmatrix} 2 & 5 \\ 1 & 4 \end{bmatrix}^8 = \begin{bmatrix} 2 & 5 \\ 1 & 4 \end{bmatrix}^4 \cdot \begin{bmatrix} 2 & 5 \\ 1 & 4 \end{bmatrix}^4.
$$

Determinant

The **determinant** $\det(A)$ of a matrix A is defined if A is a square matrix. If A is of size 1×1, then $\det(A) = A[1,1]$. The determinant of a larger matrix is calculated recursively using the formula

$$
\det(A) = \sum_{j=1}^{n} A[1,j] C[1,j],
$$

where $C[i,j]$ is the **cofactor** of A at $[i,j]$. The cofactor is calculated using the formula

$$
C[i,j] = (-1)^{i+j} \det(M[i,j]),
$$

---

[1]The first such algorithm, with time complexity $O(n^{2.80735})$, was published in 1969 [37], and the best current algorithm works in $O(n^{2.37286})$ time [18].

where $M[i,j]$ is obtained by removing row i and column j from A. Due to the coefficient $(-1)^{i+j}$ in the cofactor, every other determinant is positive and negative. For example,

$$\det(\begin{bmatrix} 3 & 4 \\ 1 & 6 \end{bmatrix}) = 3 \cdot 6 - 4 \cdot 1 = 14$$

and

$$\det(\begin{bmatrix} 2 & 4 & 3 \\ 5 & 1 & 6 \\ 7 & 2 & 4 \end{bmatrix}) = 2 \cdot \det(\begin{bmatrix} 1 & 6 \\ 2 & 4 \end{bmatrix}) - 4 \cdot \det(\begin{bmatrix} 5 & 6 \\ 7 & 4 \end{bmatrix}) + 3 \cdot \det(\begin{bmatrix} 5 & 1 \\ 7 & 2 \end{bmatrix}) = 81.$$

The determinant of A tells us whether there is an **inverse matrix** A^{-1} such that $A \cdot A^{-1} = I$, where I is an identity matrix. It turns out that A^{-1} exists exactly when $\det(A) \neq 0$, and it can be calculated using the formula

$$A^{-1}[i,j] = \frac{C[j,i]}{det(A)}.$$

For example,

$$\underbrace{\begin{bmatrix} 2 & 4 & 3 \\ 5 & 1 & 6 \\ 7 & 2 & 4 \end{bmatrix}}_{A} \cdot \underbrace{\frac{1}{81} \begin{bmatrix} -8 & -10 & 21 \\ 22 & -13 & 3 \\ 3 & 24 & -18 \end{bmatrix}}_{A^{-1}} = \underbrace{\begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix}}_{I}.$$

23.2 Linear recurrences

A **linear recurrence** can be represented as a function $f(n)$ such that the initial values are $f(0), f(1), \ldots, f(k-1)$ and the larger values are calculated recursively using the formula

$$f(n) = c_1 f(n-1) + c_2 f(n-2) + \ldots + c_k f(n-k),$$

where $c_1, c_2, \ldots, c_k$ are constant coefficients.

We can use dynamic programming to calculate any value of $f(n)$ in $O(kn)$ time by calculating all values of $f(0), f(1), \ldots, f(n)$ one after another. However, if k is small, it is possible to calculate $f(n)$ much more efficiently in $O(k^3 \log n)$ time using matrix operations.

Fibonacci numbers

A simple example of a linear recurrence is the following function that defines the Fibonacci numbers:

$$\begin{aligned} f(0) &= 0 \\ f(1) &= 1 \\ f(n) &= f(n-1) + f(n-2) \end{aligned}$$

In this case, $k = 2$ and $c_1 = c_2 = 1$.

The idea is to represent the Fibonacci formula as a square matrix X of size 2×2, for which the following holds:

$$X \cdot \begin{bmatrix} f(i) \\ f(i+1) \end{bmatrix} = \begin{bmatrix} f(i+1) \\ f(i+2) \end{bmatrix}$$

Thus, values $f(i)$ and $f(i+1)$ are given as "input" for X, and X calculates values $f(i+1)$ and $f(i+2)$ from them. It turns out that such a matrix is

$$X = \begin{bmatrix} 0 & 1 \\ 1 & 1 \end{bmatrix}.$$

For example,

$$\begin{bmatrix} 0 & 1 \\ 1 & 1 \end{bmatrix} \cdot \begin{bmatrix} f(5) \\ f(6) \end{bmatrix} = \begin{bmatrix} 0 & 1 \\ 1 & 1 \end{bmatrix} \cdot \begin{bmatrix} 5 \\ 8 \end{bmatrix} = \begin{bmatrix} 8 \\ 13 \end{bmatrix} = \begin{bmatrix} f(6) \\ f(7) \end{bmatrix}.$$

Thus, we can calculate $f(n)$ using the formula

$$\begin{bmatrix} f(n) \\ f(n+1) \end{bmatrix} = X^n \cdot \begin{bmatrix} f(0) \\ f(1) \end{bmatrix} = \begin{bmatrix} 0 & 1 \\ 1 & 1 \end{bmatrix}^n \cdot \begin{bmatrix} 0 \\ 1 \end{bmatrix}.$$

The value of X^n can be calculated in $O(k^3 \log n)$ time, so the value of $f(n)$ can also be calculated in $O(k^3 \log n)$ time.

General case

Let us now consider the general case where $f(n)$ is any linear recurrence. Again, our goal is to construct a matrix X for which

$$X \cdot \begin{bmatrix} f(i) \\ f(i+1) \\ \vdots \\ f(i+k-1) \end{bmatrix} = \begin{bmatrix} f(i+1) \\ f(i+2) \\ \vdots \\ f(i+k) \end{bmatrix}.$$

Such a matrix is

$$X = \begin{bmatrix} 0 & 1 & 0 & 0 & \cdots & 0 \\ 0 & 0 & 1 & 0 & \cdots & 0 \\ 0 & 0 & 0 & 1 & \cdots & 0 \\ \vdots & \vdots & \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & 0 & 0 & \cdots & 1 \\ c_k & c_{k-1} & c_{k-2} & c_{k-3} & \cdots & c_1 \end{bmatrix}.$$

In the first $k-1$ rows, each element is 0 except that one element is 1. These rows replace $f(i)$ with $f(i+1)$, $f(i+1)$ with $f(i+2)$, and so on. The last row contains the coefficients of the recurrence to calculate the new value $f(i+k)$.

Now, $f(n)$ can be calculated in $O(k^3 \log n)$ time using the formula

$$\begin{bmatrix} f(n) \\ f(n+1) \\ \vdots \\ f(n+k-1) \end{bmatrix} = X^n \cdot \begin{bmatrix} f(0) \\ f(1) \\ \vdots \\ f(k-1) \end{bmatrix}.$$

23.3   Graphs and matrices

Counting paths

The powers of an adjacency matrix of a graph have an interesting property. When V is an adjacency matrix of an unweighted graph, the matrix V^n contains the numbers of paths of n edges between the nodes in the graph.

For example, for the graph
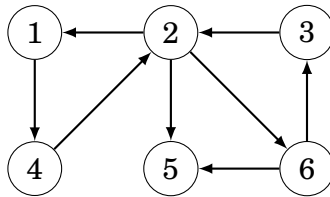
the adjacency matrix is

$$V = \begin{bmatrix} 0 & 0 & 0 & 1 & 0 & 0 \\ 1 & 0 & 0 & 0 & 1 & 1 \\ 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 1 & 0 \end{bmatrix}.$$

Now, for example, the matrix

$$V^4 = \begin{bmatrix} 0 & 0 & 1 & 1 & 1 & 0 \\ 2 & 0 & 0 & 0 & 2 & 2 \\ 0 & 2 & 0 & 0 & 0 & 0 \\ 0 & 2 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 1 & 1 & 0 \end{bmatrix}$$

contains the numbers of paths of 4 edges between the nodes. For example, $V^4[2,5] = 2$, because there are two paths of 4 edges from node 2 to node 5: $2 \to 1 \to 4 \to 2 \to 5$ and $2 \to 6 \to 3 \to 2 \to 5$.

Shortest paths

Using a similar idea in a weighted graph, we can calculate for each pair of nodes the shortest path between them that contains exactly n edges. To calculate this, we have to define matrix multiplication in a new way, so that we do not calculate the numbers of paths but minimize the lengths of paths.

As an example, consider the following graph:
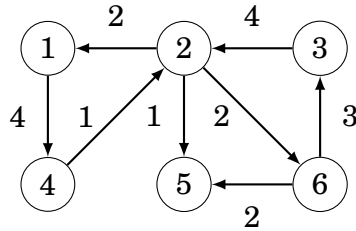
Let us construct an adjacency matrix where ∞ means that an edge does not exist, and other values correspond to edge weights. The matrix is

$$V = \begin{bmatrix} \infty & \infty & \infty & 4 & \infty & \infty \\ 2 & \infty & \infty & \infty & 1 & 2 \\ \infty & 4 & \infty & \infty & \infty & \infty \\ \infty & 1 & \infty & \infty & \infty & \infty \\ \infty & \infty & \infty & \infty & \infty & \infty \\ \infty & \infty & 3 & \infty & 2 & \infty \end{bmatrix}.$$

Instead of the formula

$$AB[i,j] = \sum_{k=1}^{n} A[i,k] \cdot B[k,j]$$

we now use the formula

$$AB[i,j] = \min_{k=1}^{n} A[i,k] + B[k,j]$$

for matrix multiplication, so we calculate a minimum instead of a sum, and a sum of elements instead of a product. After this modification, matrix powers correspond to shortest paths in the graph.

For example, as

$$V^4 = \begin{bmatrix} \infty & \infty & 10 & 11 & 9 & \infty \\ 9 & \infty & \infty & \infty & 8 & 9 \\ \infty & 11 & \infty & \infty & \infty & \infty \\ \infty & 8 & \infty & \infty & \infty & \infty \\ \infty & \infty & \infty & \infty & \infty & \infty \\ \infty & \infty & 12 & 13 & 11 & \infty \end{bmatrix},$$

we can conclude that the shortest path of 4 edges from node 2 to node 5 has length 8. This path is $2 \to 1 \to 4 \to 2 \to 5$.

Kirchhoff's theorem

Kirchhoff's theorem provides a way to calculate the number of spanning trees of a graph as a determinant of a special matrix. For example, the graph
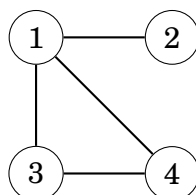
has three spanning trees:
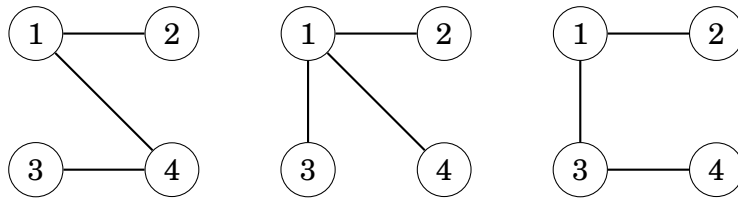
To calculate the number of spanning trees, we construct a **Laplacean matrix** L, where $L[i,i]$ is the degree of node i and $L[i,j] = -1$ if there is an edge between nodes i and j, and otherwise $L[i,j] = 0$. The Laplacean matrix for the above graph is as follows:

$$L = \begin{bmatrix} 3 & -1 & -1 & -1 \\ -1 & 1 & 0 & 0 \\ -1 & 0 & 2 & -1 \\ -1 & 0 & -1 & 2 \end{bmatrix}$$

The number of spanning trees equals the determinant of a matrix that is obtained when we remove any row and any column from L. For example, if we remove the first row and column, the result is

$$\det\left(\begin{bmatrix} 1 & 0 & 0 \\ 0 & 2 & -1 \\ 0 & -1 & 2 \end{bmatrix}\right) = 3.$$

The determinant is always the same, regardless of which row and column we remove from L.

Note that a special case of Kirchhoff's theorem is Cayley's formula in Chapter 22.5, because in a complete graph of n nodes

$$\det\left(\begin{bmatrix} n-1 & -1 & \cdots & -1 \\ -1 & n-1 & \cdots & -1 \\ \vdots & \vdots & \ddots & \vdots \\ -1 & -1 & \cdots & n-1 \end{bmatrix}\right) = n^{n-2}.$$

Chapter 24

Probability

A **probability** is a real number between 0 and 1 that indicates how probable an event is. If an event is certain to happen, its probability is 1, and if an event is impossible, its probability is 0. The probability of an event is denoted $P(\cdots)$ where the three dots describe the event.

For example, when throwing a dice, the outcome is an integer between 1 and 6, and it is assumed that the probability of each outcome is 1/6. For example, we can calculate the following probabilities:

- $P(\text{"the result is 4"}) = 1/6$
- $P(\text{"the result is not 6"}) = 5/6$
- $P(\text{"the result is even"}) = 1/2$

24.1   Calculation

To calculate the probability of an event, we can either use combinatorics or simulate the process that generates the event. As an example, let us calculate the probability of drawing three cards with the same value from a shuffled deck of cards (for example, ♠8, ♣8 and ♢8).

Method 1

We can calculate the probability using the formula

$$\frac{\text{number of desired outcomes}}{\text{total number of outcomes}}.$$

In this problem, the desired outcomes are those in which the value of each card is the same. There are $13\binom{4}{3}$ such outcomes, because there are 13 possibilities for the value of the cards and $\binom{4}{3}$ ways to choose 3 suits from 4 possible suits.

There are a total of $\binom{52}{3}$ outcomes, because we choose 3 cards from 52 cards. Thus, the probability of the event is

$$\frac{13\binom{4}{3}}{\binom{52}{3}} = \frac{1}{425}.$$

217

Method 2

Another way to calculate the probability is to simulate the process that generates the event. In this case, we draw three cards, so the process consists of three steps. We require that each step in the process is successful.

Drawing the first card certainly succeeds, because there are no restrictions. The second step succeeds with probability 3/51, because there are 51 cards left and 3 of them have the same value as the first card. In a similar way, the third step succeeds with probability 2/50.

The probability that the entire process succeeds is

$$1 \cdot \frac{3}{51} \cdot \frac{2}{50} = \frac{1}{425}.$$

24.2   Events

An event in probability can be represented as a set

$$A \subset X,$$

where X contains all possible outcomes and A is a subset of outcomes. For example, when drawing a dice, the outcomes are

$$X = \{1, 2, 3, 4, 5, 6\}.$$

Now, for example, the event "the result is even" corresponds to the set

$$A = \{2, 4, 6\}.$$

Each outcome x is assigned a probability $p(x)$. Furthermore, the probability $P(A)$ of an event that corresponds to a set A can be calculated as a sum of probabilities of outcomes using the formula

$$P(A) = \sum_{x \in A} p(x).$$

For example, when throwing a dice, $p(x) = 1/6$ for each outcome x, so the probability of the event "the result is even" is

$$p(2) + p(4) + p(6) = 1/2.$$

The total probability of the outcomes in X must be 1, i.e., $P(X) = 1$.

Since the events in probability are sets, we can manipulate them using standard set operations:

- The **complement** $\bar{A}$ means "A does not happen". For example, when throwing a dice, the complement of $A = \{2, 4, 6\}$ is $\bar{A} = \{1, 3, 5\}$.

- The **union** $A \cup B$ means "A or B happen". For example, the union of $A = \{2, 5\}$ and $B = \{4, 5, 6\}$ is $A \cup B = \{2, 4, 5, 6\}$.

- The **intersection** $A \cap B$ means "A and B happen". For example, the intersection of $A = \{2, 5\}$ and $B = \{4, 5, 6\}$ is $A \cap B = \{5\}$.

Complement

The probability of the complement $\bar{A}$ is calculated using the formula

$$P(\bar{A}) = 1 - P(A).$$

Sometimes, we can solve a problem easily using complements by solving the opposite problem. For example, the probability of getting at least one six when throwing a dice ten times is

$$1 - (5/6)^{10}.$$

Here 5/6 is the probability that the outcome of a single throw is not six, and $(5/6)^{10}$ is the probability that none of the ten throws is a six. The complement of this is the answer to the problem.

Union

The probability of the union $A \cup B$ is calculated using the formula

$$P(A \cup B) = P(A) + P(B) - P(A \cap B).$$

For example, when throwing a dice, the union of the events

$$A = \text{"the result is even"}$$

and

$$B = \text{"the result is less than 4"}$$

is

$$A \cup B = \text{"the result is even or less than 4"},$$

and its probability is

$$P(A \cup B) = P(A) + P(B) - P(A \cap B) = 1/2 + 1/2 - 1/6 = 5/6.$$

If the events A and B are **disjoint**, i.e., $A \cap B$ is empty, the probability of the event $A \cup B$ is simply

$$P(A \cup B) = P(A) + P(B).$$

Conditional probability

The **conditional probability**

$$P(A|B) = \frac{P(A \cap B)}{P(B)}$$

is the probability of A assuming that B happens. In this situation, when calculating the probability of A, we only consider the outcomes that also belong to B.

Using the above sets,

$$P(A|B) = 1/3,$$

because the outcomes of B are $\{1,2,3\}$, and one of them is even. This is the probability of an even result if we know that the result is between $1 \dots 3$.

Intersection

Using conditional probability, the probability of the intersection $A \cap B$ can be calculated using the formula

$$P(A \cap B) = P(A)P(B|A).$$

Events A and B are **independent** if

$$P(A|B) = P(A) \quad \text{and} \quad P(B|A) = P(B),$$

which means that the fact that B happens does not change the probability of A, and vice versa. In this case, the probability of the intersection is

$$P(A \cap B) = P(A)P(B).$$

For example, when drawing a card from a deck, the events

$$A = \text{"the suit is clubs"}$$

and

$$B = \text{"the value is four"}$$

are independent. Hence the event

$$A \cap B = \text{"the card is the four of clubs"}$$

happens with probability

$$P(A \cap B) = P(A)P(B) = 1/4 \cdot 1/13 = 1/52.$$

24.3   Random variables

A **random variable** is a value that is generated by a random process. For example, when throwing two dice, a possible random variable is

$$X = \text{"the sum of the results"}.$$

For example, if the results are $[4, 6]$ (meaning that we first throw a four and then a six), then the value of X is 10.

We denote $P(X = x)$ the probability that the value of a random variable X is x. For example, when throwing two dice, $P(X = 10) = 3/36$, because the total number of outcomes is 36 and there are three possible ways to obtain the sum 10: $[4, 6]$, $[5, 5]$ and $[6, 4]$.

Expected value

The **expected value** $E[X]$ indicates the average value of a random variable X. The expected value can be calculated as the sum

$$\sum_x P(X = x)x,$$

where x goes through all possible values of X.

For example, when throwing a dice, the expected result is

$$1/6 \cdot 1 + 1/6 \cdot 2 + 1/6 \cdot 3 + 1/6 \cdot 4 + 1/6 \cdot 5 + 1/6 \cdot 6 = 7/2.$$

A useful property of expected values is **linearity**. It means that the sum $E[X_1 + X_2 + \cdots + X_n]$ always equals the sum $E[X_1] + E[X_2] + \cdots + E[X_n]$. This formula holds even if random variables depend on each other.

For example, when throwing two dice, the expected sum is

$$E[X_1 + X_2] = E[X_1] + E[X_2] = 7/2 + 7/2 = 7.$$

Let us now consider a problem where n balls are randomly placed in n boxes, and our task is to calculate the expected number of empty boxes. Each ball has an equal probability to be placed in any of the boxes. For example, if $n = 2$, the possibilities are as follows:

In this case, the expected number of empty boxes is

$$\frac{0 + 0 + 1 + 1}{4} = \frac{1}{2}.$$

In the general case, the probability that a single box is empty is

$$\left(\frac{n-1}{n}\right)^n,$$

because no ball should be placed in it. Hence, using linearity, the expected number of empty boxes is

$$n \cdot \left(\frac{n-1}{n}\right)^n.$$

Distributions

The **distribution** of a random variable X shows the probability of each value that X may have. The distribution consists of values $P(X = x)$. For example, when throwing two dice, the distribution for their sum is:

x	2	3	4	5	6	7	8	9	10	11	12
$P(X = x)$	1/36	2/36	3/36	4/36	5/36	6/36	5/36	4/36	3/36	2/36	1/36

In a **uniform distribution**, the random variable X has n possible values $a, a+1, \ldots, b$ and the probability of each value is $1/n$. For example, when throwing a dice, $a = 1$, $b = 6$ and $P(X = x) = 1/6$ for each value x.

The expected value for X in a uniform distribution is

$$E[X] = \frac{a+b}{2}.$$

In a **binomial distribution**, n attempts are made and the probability that a single attempt succeeds is p. The random variable X counts the number of successful attempts, and the probability for a value x is

$$P(X = x) = p^x (1-p)^{n-x} \binom{n}{x},$$

where p^x and $(1-p)^{n-x}$ correspond to successful and unsuccessful attemps, and $\binom{n}{x}$ is the number of ways we can choose the order of the attempts.

For example, when throwing a dice ten times, the probability of throwing a six exactly three times is $(1/6)^3 (5/6)^7 \binom{10}{3}$.

The expected value for X in a binomial distribution is

$$E[X] = pn.$$

In a **geometric distribution**, the probability that an attempt succeeds is p, and we continue until the first success happens. The random variable X counts the number of attempts needed, and the probability for a value x is

$$P(X = x) = (1-p)^{x-1} p,$$

where $(1-p)^{x-1}$ corresponds to unsuccessful attemps and p corresponds to the first successful attempt.

For example, if we throw a dice until we throw a six, the probability that the number of throws is exactly 4 is $(5/6)^3 1/6$.

The expected value for X in a geometric distribution is

$$E[X] = \frac{1}{p}.$$

24.4 Markov chains

A **Markov chain** is a random process that consists of states and transitions between them. For each state, we know the probabilities for moving to other states. A Markov chain can be represented as a graph whose nodes are states and edges are transitions.

As an example, let us consider a problem where we are in floor 1 in an n floor building. At each step, we randomly walk either one floor up or one floor down,

except that we always walk one floor up from floor 1 and one floor down from floor n. What is the probability of being in floor m after k steps?

In this problem, each floor of the building corresponds to a state in a Markov chain. For example, if $n = 5$, the graph is as follows:
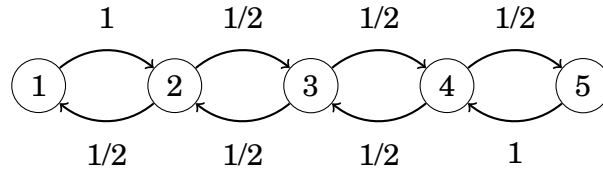
The probability distribution of a Markov chain is a vector $[p_1, p_2, \ldots, p_n]$, where p_k is the probability that the current state is k. The formula $p_1 + p_2 + \cdots + p_n = 1$ always holds.

In the example, the initial distribution is $[1, 0, 0, 0, 0]$, because we always begin in floor 1. The next distribution is $[0, 1, 0, 0, 0]$, because we can only move from floor 1 to floor 2. After this, we can either move one floor up or one floor down, so the next distribution is $[1/2, 0, 1/2, 0, 0]$, and so on.

An efficient way to simulate the walk in a Markov chain is to use dynamic programming. The idea is to maintain the probability distribution and at each step go through all possibilities how we can move. Using this method, we can simulate m steps in $O(n^2 m)$ time.

The transitions of a Markov chain can also be represented as a matrix that updates the probability distribution. In this example, the matrix is

$$
\begin{bmatrix}
0 & 1/2 & 0 & 0 & 0 \\
1 & 0 & 1/2 & 0 & 0 \\
0 & 1/2 & 0 & 1/2 & 0 \\
0 & 0 & 1/2 & 0 & 1 \\
0 & 0 & 0 & 1/2 & 0
\end{bmatrix}.
$$

When we multiply a probability distribution by this matrix, we get the new distribution after moving one step. For example, we can move from the distribution $[1, 0, 0, 0, 0]$ to the distribution $[0, 1, 0, 0, 0]$ as follows:

$$
\begin{bmatrix}
0 & 1/2 & 0 & 0 & 0 \\
1 & 0 & 1/2 & 0 & 0 \\
0 & 1/2 & 0 & 1/2 & 0 \\
0 & 0 & 1/2 & 0 & 1 \\
0 & 0 & 0 & 1/2 & 0
\end{bmatrix}
\begin{bmatrix}
1 \\
0 \\
0 \\
0 \\
0
\end{bmatrix}
=
\begin{bmatrix}
0 \\
1 \\
0 \\
0 \\
0
\end{bmatrix}.
$$

By calculating matrix powers efficiently, we can calculate the distribution after m steps in $O(n^3 \log m)$ time.

24.5 Randomized algorithms

Sometimes we can use randomness for solving a problem, even if the problem is not related to probabilities. A **randomized algorithm** is an algorithm that is based on randomness.

A **Monte Carlo algorithm** is a randomized algorithm that may sometimes give a wrong answer. For such an algorithm to be useful, the probability of a wrong answer should be small.

A **Las Vegas algorithm** is a randomized algorithm that always gives the correct answer, but its running time varies randomly. The goal is to design an algorithm that is efficient with high probability.

Next we will go through three example problems that can be solved using randomness.

Order statistics

The kth **order statistic** of an array is the element at position k after sorting the array in increasing order. It is easy to calculate any order statistic in $O(n \log n)$ time by sorting the array, but is it really needed to sort the entire array just to find one element?

It turns out that we can find order statistics using a randomized algorithm without sorting the array. The algorithm is a Las Vegas algorithm: its running time is usually $O(n)$ but $O(n^2)$ in the worst case.

The algorithm chooses a random element x in the array, and moves elements smaller than x to the left part of the array, and all other elements to the right part of the array. This takes $O(n)$ time when there are n elements. Assume that the left part contains a elements and the right part contains b elements. If $a = k - 1$, element x is the kth order statistic. Otherwise, if $a > k - 1$, we recursively find the kth order statistic for the left part, and if $a < k - 1$, we recursively find the rth order statistic for the right part where $r = k - a$. The search continues in a similar way, until the element has been found.

When each element x is randomly chosen, the size of the array about halves at each step, so the time complexity for finding the kth order statistic is about

$$n + n/2 + n/4 + n/8 + \cdots = O(n).$$

The worst case for the algorithm is still $O(n^2)$, because it is possible that x is always chosen in such a way that it is one of the smallest or largest elements in the array and $O(n)$ steps are needed. However, the probability for this is so small that this never happens in practice.

Verifying matrix multiplication

Our next problem is to *verify* if $AB = C$ holds when A, B and C are matrices of size $n \times n$. Of course, we can solve the problem by calculating the product AB again (in $O(n^3)$ time using the basic algorithm), but one could hope that verifying the answer would by easier than to calculate it from scratch.

It turns out that we can solve the problem using a Monte Carlo algorithm whose time complexity is only $O(n^2)$. The idea is simple: we choose a random vector X of n elements, and calculate the matrices ABX and CX. If $ABX = CX$, we report that $AB = C$, and otherwise we report that $AB \neq C$.

The time complexity of the algorithm is $O(n^2)$, because we can calculate the matrices ABX and CX in $O(n^2)$ time. We can calculate the matrix ABX efficiently by using the representation $A(BX)$, so only two multiplications of $n \times n$ and $n \times 1$ size matrices are needed.

The drawback of the algorithm is that there is a small chance that the algorithm makes a mistake when it reports that $AB = C$. For example,

$$\begin{bmatrix} 2 & 4 \\ 1 & 6 \end{bmatrix} \neq \begin{bmatrix} 0 & 5 \\ 7 & 4 \end{bmatrix},$$

but

$$\begin{bmatrix} 2 & 4 \\ 1 & 6 \end{bmatrix} \begin{bmatrix} 1 \\ 3 \end{bmatrix} = \begin{bmatrix} 0 & 5 \\ 7 & 4 \end{bmatrix} \begin{bmatrix} 1 \\ 3 \end{bmatrix}.$$

However, in practice, the probability that the algorithm makes a mistake is small, and we can decrease the probability by verifying the result using multiple random vectors X before reporting that $AB = C$.

Graph coloring

Given a graph that contains n nodes and m edges, our task is to find a way to color the nodes of the graph using two colors so that for at least $m/2$ edges, the endpoints have different colors. For example, in the graph
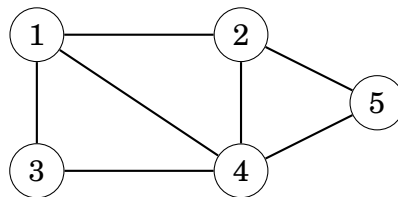
a valid coloring is as follows:
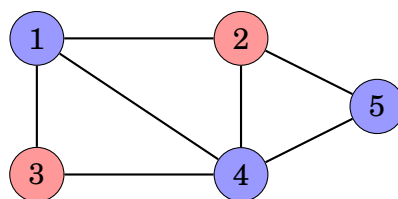
The above graph contains 7 edges, and for 5 of them, the endpoints have different colors, so the coloring is valid.

The problem can be solved using a Las Vegas algorithm that generates random colorings until a valid coloring has been found. In a random coloring, the color of each node is independently chosen so that the probability of both colors is 1/2.

In a random coloring, the probability that the endpoints of a single edge have different colors is 1/2. Hence, the expected number of edges whose endpoints have different colors is $m/2$. Since it is excepted that a random coloring is valid, we will quickly find a valid coloring in practice.

Chapter 25

Game theory

In this chapter, we will focus on two-player games that do not contain random elements. Our goal is to find a strategy that we can follow to win the game no matter what the opponent does, if such a strategy exists.

It turns out that there is a general strategy for all such games, and we can analyze the games using the **nim theory**. First, we will analyze simple games where players remove sticks from heaps, and after this, we will generalize the strategy used in those games to all other games.

25.1 Game states

Let us consider a game where there is initially a heap of n sticks. Players A and B move alternatively, and player A begins. On each move, the player has to remove 1, 2 or 3 sticks from the heap, and the player who removes the last stick wins the game.

For example, if $n = 10$, the game may proceed as follows:

1. Player A removes 2 sticks (8 sticks left).
2. Player B removes 3 sticks (5 sticks left).
3. Player A removes 1 stick (4 sticks left).
4. Player B removes 2 sticks (2 sticks left).
5. Player A removes 2 sticks and wins.

This game consists of states $0, 1, 2, \ldots, n$, where the number of the state corresponds to the number of sticks left.

Winning and losing states

A **winning state** is a state where the player will win the game if they play optimally, and a **losing state** is a state where the player will lose the game if the opponent plays optimally. It turns out that we can classify all states of a game so that each state is either a winning state or a losing state.

In the above game, state 0 is clearly a losing state, because the player cannot make any moves. States 1, 2 and 3 are winning states, because we can remove 1,

2 or 3 sticks and win the game. State 4, in turn, is a losing state, because any move leads to a state that is a winning state for the opponent.

More generally, if there is a move that leads from the current state to a losing state, the current state is a winning state, and otherwise it is a losing state. Using this observation, we can classify all states of a game starting with losing states where there are no possible moves.

The states $0 \dots 15$ of the above game can be classified as follows (W denotes a winning state and L denotes a losing state):

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
L	W	W	W	L	W	W	W	L	W	W	W	L	W	W	W

It is easy to analyze this game: a state k is a losing state if k is divisible by 4, and otherwise it is winning state. An optimal way to play the game is to always choose a move after which the number of sticks in the heap is divisible by 4. Finally, there are no sticks left and the opponent has lost.

Of course, this strategy requires that the number of sticks is *not* divisible by 4 when it is our move. If it is, there is nothing we can do, but the opponent will win the game if they play optimally.

State graph

Let us now consider another stick game, where in each state k, it is allowed to remove any number x of sticks such that x is smaller than k and divides k. For example, in state 8 we may remove 1, 2 or 4 sticks, but in state 7 the only allowed move is to remove 1 stick.

The following picture shows the states $1 \dots 9$ of the game as a **state graph**, whose nodes are the states and edges are the moves between them:
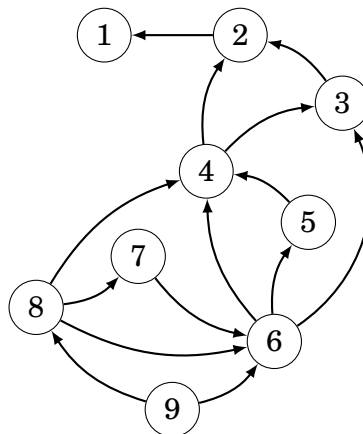
The final state in this game is always state 1, which is a losing state, because there are no valid moves. The classification of states $1 \dots 9$ is as follows:

1	2	3	4	5	6	7	8	9
L	W	L	W	L	W	L	W	L

Surprisingly, in this game, all even-numbered states are winning states, and all odd-numbered states are losing states.

25.2   Nim game

The **nim game** is a simple game that has an important role in game theory, because many other games can be played using the same strategy. First, we focus on nim, and then we generalize the strategy to other games.

There are n heaps in nim, and each heap contains some number of sticks. The players move alternatively, and on each turn, the player chooses a heap that still contains sticks and removes any number of sticks from it. The winner is the player who removes the last stick.

The states in nim are of the form $[x_1, x_2, \ldots, x_n]$, where x_k denotes the number of sticks in heap k. For example, $[10, 12, 5]$ is a game where there are three heaps with 10, 12 and 5 sticks. The state $[0, 0, \ldots, 0]$ is a losing state, because it is not possible to remove any sticks, and this is always the final state.

Analysis

It turns out that we can easily classify any nim state by calculating the **nim sum** $x_1 \oplus x_2 \oplus \cdots \oplus x_n$, where $\oplus$ is the xor operation. The states whose nim sum is 0 are losing states, and all other states are winning states. For example, the nim sum for $[10, 12, 5]$ is $10 \oplus 12 \oplus 5 = 3$, so the state is a winning state.

But how is the nim sum related to the nim game? We can explain this by looking at how the nim sum changes when the nim state changes.

Losing states: The final state $[0, 0, \ldots, 0]$ is a losing state, and its nim sum is 0, as expected. In other losing states, any move leads to a winning state, because when a single value x_k changes, the nim sum also changes, so the nim sum is different from 0 after the move.

Winning states: We can move to a losing state if there is any heap k for which $x_k \oplus s < x_k$. In this case, we can remove sticks from heap k so that it will contain $x_k \oplus s$ sticks, which will lead to a losing state. There is always such a heap, where x_k has a one bit at the position of the leftmost one bit in s.

As an example, consider the state $[10, 2, 5]$. This state is a winning state, because its nim sum is 3. Thus, there has to be a move which leads to a losing state. Next we will find out such a move.

The nim sum of the state is as follows:

10	1010
12	1100
5	0101
3	0011

In this case, the heap with 10 sticks is the only heap that has a one bit at the position of the leftmost one bit in the nim sum:

```
10 | 10①0
12 | 1100
 5 | 0101
─────────
 3 | 00①1
```

The new size of the heap has to be $10 \oplus 3 = 9$, so we will remove just one stick. After this, the state will be $[9, 12, 5]$, which is a losing state:

```
 9 | 1001
12 | 1100
 5 | 0101
─────────
 0 | 0000
```

Misère game

In a **misère game**, the goal is opposite, so the player who removes the last stick loses the game. It turns out that a misère nim game can be optimally played almost like the standard nim game.

The idea is to first play the misère game like a standard game, but change the strategy at the end of the game. The new strategy will be introduced in a situation where each heap would contain at most one stick after the next move.

In the standard game, we should choose a move after which there is an even number of heaps with one stick. However, in the misère game, we choose a move so that there is an odd number of heaps with one stick.

This strategy works because a state where the strategy changes always appears in the game, and this state is a winning state, because it contains exactly one heap that has more than one stick so the nim sum is not 0.

25.3 Sprague–Grundy theorem

The **Sprague–Grundy theorem** generalizes the strategy used in nim to all games that fulfil the following requirements:

- There are two players who move alternatively.
- The game consists of states, and the possible moves in a state do not depend on whose turn it is.
- The game ends when a player cannot make a move.
- The game surely ends sooner or later.
- The players have complete information about the states and allowed moves, and there is no randomness in the game.

The idea is to calculate for each game state a Grundy number that corresponds to the number of sticks in a nim heap. When we know the Grundy numbers for all states, we can play the game like the nim game.

Grundy numbers

The **Grundy number** for a game state is

$$\text{mex}(\{g_1, g_2, \ldots, g_n\}),$$

where $g_1, g_2, \ldots, g_n$ are Grundy numbers for states to which we can move from the state, and the mex function gives the smallest nonnegative number that is not in the set. For example, $\text{mex}(\{0, 1, 3\}) = 2$. If there are no possible moves in a state, its Grundy number is 0, because $\text{mex}(\emptyset) = 0$.

For example, in the state graph
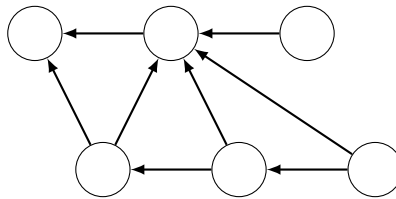
the Grundy numbers are as follows:
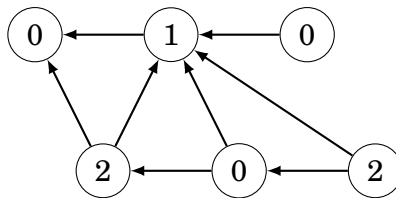
The Grundy number of a losing state is 0, and the Grundy number of a winning state is a positive number.

The Grundy number of a state corresponds to a number of sticks in a nim heap. If the Grundy number is 0, we can only move to states whose Grundy numbers are positive, and if the Grundy number is $x > 0$, we can move to states whose Grundy numbers incluce all numbers $0, 1, \ldots, x-1$.

As an example, let us consider a game where the players move a figure in a maze. Each square in the maze is either floor or wall. On each turn, the player has to move the figure some number of steps left or up. The winner of the game is the player who makes the last move.

The following picture shows a possible initial state of the game, where @ denotes the figure and * denotes a square where it can move.
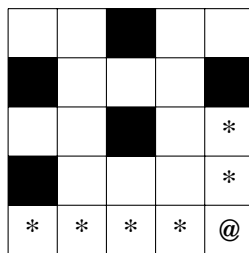
The states of the game are all floor squares in the maze. In this situation, the Grundy numbers are as follows:

0	1	■	0	1
■	0	1	2	■
0	2	■	1	0
■	3	0	4	1
0	4	1	3	2

Thus, each state of the maze game corresponds to a heap in the nim game. For example, the Grundy number for the lower-right square is 2, so it is a winning state. We can reach a losing state and win the game by moving either four steps left or two steps up.

Note that unlike in the original nim game, it may be possible to move to a state whose Grundy number is larger than the Grundy number of the current state. However, the opponent can always choose a move that cancels such a move, so it is not possible to escape from a losing state.

Subgames

Next we will assume that our game consists of subgames, and on each turn, the player first chooses a subgame and then a move in the subgame. The game ends when it is not possible to make any move in any subgame.

In this case, the Grundy number of a game is the nim sum of the Grundy numbers of the subgames. The game can be played like a nim game by calculating all Grundy numbers for subgames and then their nim sum.

As an example, consider a game that consists of three mazes. In this game, on each turn the player chooses one of the mazes and then moves the figure in the maze. Assume that the initial state of the game is as follows:
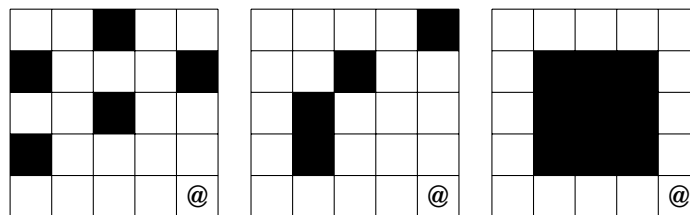
The Grundy numbers for the mazes are as follows:

0	1	■	0	1
■	0	1	2	■
0	2	■	1	0
■	3	0	4	1
0	4	1	3	2

0	1	2	3	■
1	0	■	0	1
2	■	0	1	2
3	■	1	2	0
4	0	2	5	3

0	1	2	3	4
1	■	■	■	0
2	■	■	■	1
3	■	■	■	2
4	0	1	2	3

In the initial state, the nim sum of the Grundy numbers is $2 \oplus 3 \oplus 3 = 2$, so the first player can win the game. An optimal move is to move two steps up in the first maze, which produces the nim sum $0 \oplus 3 \oplus 3 = 0$.

Grundy's game

Sometimes a move in the game divides the game into subgames that are independent of each other. In this case, the Grundy number of the game is

$$\text{mex}(\{g_1, g_2, \ldots, g_n\}),$$

where n is the number of possible moves and

$$g_k = a_{k,1} \oplus a_{k,2} \oplus \ldots \oplus a_{k,m},$$

where move k generates subgames with Grundy numbers $a_{k,1}, a_{k,2}, \ldots, a_{k,m}$.

An example of such a game is **Grundy's game**. Initially, there is a single heap that contains n sticks. On each turn, the player chooses a heap and divides it into two nonempty heaps such that the heaps are of different size. The player who makes the last move wins the game.

Let $f(n)$ be the Grundy number of a heap that contains n sticks. The Grundy number can be calculated by going through all ways to divide the heap into two heaps. For example, when $n = 8$, the possibilities are $1 + 7$, $2 + 6$ and $3 + 5$, so

$$f(8) = \text{mex}(\{f(1) \oplus f(7), f(2) \oplus f(6), f(3) \oplus f(5)\}).$$

In this game, the value of $f(n)$ is based on the values of $f(1), \ldots, f(n-1)$. The base cases are $f(1) = f(2) = 0$, because it is not possible to divide the heaps of 1 and 2 sticks. The first Grundy numbers are:

$$
\begin{aligned}
f(1) &= 0 \\
f(2) &= 0 \\
f(3) &= 1 \\
f(4) &= 0 \\
f(5) &= 2 \\
f(6) &= 1 \\
f(7) &= 0 \\
f(8) &= 2
\end{aligned}
$$

The Grundy number for $n = 8$ is 2, so it is possible to win the game. The winning move is to create heaps $1 + 7$, because $f(1) \oplus f(7) = 0$.

Chapter 26

String algorithms

This chapter deals with efficient algorithms for string processing. Many string problems can be easily solved in $O(n^2)$ time, but the challenge is to find algorithms that work in $O(n)$ or $O(n \log n)$ time.

For example, a fundamental problem related to strings is the **pattern matching** problem: given a string of length n and a pattern of length m, our task is to find the positions where the pattern occurs in the string. For example, the pattern ABC occurs two times in the string ABABCBABC.

The pattern matching problem is easy to solve in $O(nm)$ time by a brute force algorithm that goes through all positions where the pattern may occur in the string. However, in this chapter, we will see that there are more efficient algorithms that require only $O(n + m)$ time.

26.1 String terminology

An **alphabet** is a set of characters that may appear in strings. For example, the alphabet {A, B, ..., Z} consists of the capital letters of English.

A **substring** is a sequence of consecutive characters of a string. The number of substrings of a string is $n(n + 1)/2$. For example, the substrings of the string ABCD are A, B, C, D, AB, BC, CD, ABC, BCD and ABCD.

A **subsequence** is a sequence of (not necessarily consecutive) characters of a string in their original order. The number of subsequences of a string is $2^n - 1$. For example, the subsequences of the string ABCD are A, B, C, D, AB, AC, AD, BC, BD, CD, ABC, ABD, ACD, BCD and ABCD.

A **prefix** is a subtring that starts at the beginning of a string, and a **suffix** is a substring that ends at the end of a string. For example, for the string ABCD, the prefixes are A, AB, ABC and ABCD and the suffixes are D, CD, BCD and ABCD.

A **rotation** can be generated by moving characters one by one from the beginning to the end of a string (or vice versa). For example, the rotations of the string ABCD are ABCD, BCDA, CDAB and DABC.

A **period** is a prefix of a string such that the string can be constructed by repeating the period. The last repetition may be partial and contain only a prefix of the period. For example, the shortest period of ABCABCA is ABC.

A **border** is a string that is both a prefix and a suffix of a string. For example, the borders of the string `ABACABA` are `A`, `ABA` and `ABACABA`.

Strings are usually compared using the **lexicographical order** that corresponds to the alphabetical order. It means that $x < y$ if either $x \neq y$ and x is a prefix of y, or there is a position k such that $x[i] = y[i]$ when $i < k$ and $x[k] < y[k]$.

26.2 Trie structures

A **trie** is a tree structure that maintains a set of strings. Each string in a trie corresponds to a chain of characters starting at the root node. If two strings have a common prefix, they also have a common chain in the tree.

For example, consider the following trie:
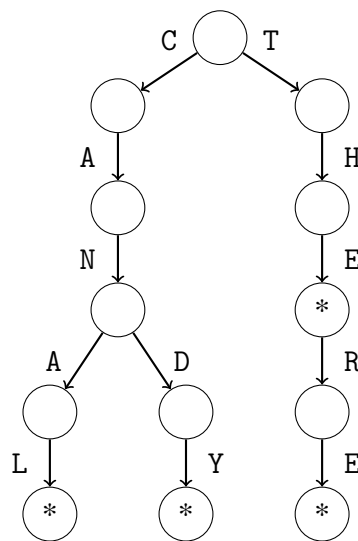
This trie corresponds to the set {`CANAL`, `CANDY`, `THE`, `THERE`}. The character * in a node means that one of the strings in the set ends at the node. This character is needed, because a string may be a prefix of another string. For example, in this trie, `THE` is a prefix of `THERE`.

We can check if a trie contains a string in $O(n)$ time where n is the length of the string, because we can follow the chain that starts at the root node. We can also add a new string to the trie in $O(n)$ time using a similar idea. If needed, new nodes will be added to the trie.

Using a trie, we can also find for a given string the longest prefix that belongs to the set. In addition, by storing additional information in each node, it is possible to calculate the number of strings that have a given prefix.

A trie can be stored in an array

```
int t[N][A];
```

where N is the maximum number of nodes (the maximum total length of the strings in the set) and A is the size of the alphabet. The nodes of a trie are numbered $1, 2, 3, \ldots$ so that the number of the root is 1, and `t[s][c]` is the next node in the chain from node s using character c.

26.3 String hashing

String hashing is a technique that allows us to efficiently check whether two substrings in a string are equal[1]. The idea is to compare the hash values of the substrings instead of their individual characters.

Calculating hash values

A **hash value** of a string is a number that is calculated from the characters of the string. If two strings are the same, their hash values are also the same, which makes it possible to compare strings based on their hash values.

A usual way to implement string hashing is polynomial hashing, which means that the hash value is calculated using the formula

$$(c[1]A^{n-1} + c[2]A^{n-2} + \cdots + c[n]A^0) \bmod B,$$

where $c[1], c[2], \ldots, c[n]$ are the codes of the characters in the string, and A and B are pre-chosen constants.

For example, the codes of the characters in the string `ALLEY` are:

A	L	L	E	Y
65	76	76	69	89

Thus, if $A = 3$ and $B = 97$, the hash value for the string `ALLEY` is

$$(65 \cdot 3^4 + 76 \cdot 3^3 + 76 \cdot 3^2 + 69 \cdot 3^1 + 89 \cdot 3^0) \bmod 97 = 52.$$

Preprocessing

To efficiently calculate hash values of substrings, we need to preprocess the string. It turns out that using polynomial hashing, we can calculate the hash value of any substring in $O(1)$ time after an $O(n)$ time preprocessing.

The idea is to construct an array h such that $h[k]$ contains the hash value of the prefix of the string that ends at position k. The array values can be recursively calculated as follows:

$$
\begin{aligned}
h[0] &= 0 \\
h[k] &= (h[k-1]A + c[k]) \bmod B
\end{aligned}
$$

In addition, we construct an array p where $p[k] = A^k \bmod B$:

$$
\begin{aligned}
p[0] &= 1 \\
p[k] &= (p[k-1]A) \bmod B.
\end{aligned}
$$

Constructing these arrays takes $O(n)$ time. After this, the hash value of a substring that begins at position a and ends at position b can be calculated in $O(1)$ time using the formula

$$(h[b] - h[a-1]p[b-a+1]) \bmod B.$$

---

[1]The technique was popularized by the Karp–Rabin pattern matching algorithm [28].

Using hash values

We can efficiently compare strings using hash values. Instead of comparing the individual characters of the strings, the idea is to compare their hash values. If the hash values are equal, the strings are *probably* equal, and if the hash values are different, the strings are *certainly* different.

Using hashing, we can often make a brute force algorithm efficient. As an example, consider the pattern matching problem: given a string s and a pattern p, find the positions where p occurs in s. A brute force algorithm goes through all positions where p may occur and compares the strings character by character. The time complexity of such an algorithm is $O(n^2)$.

We can make the brute force algorithm more efficient using hashing, because the algorithm compares substrings of strings. Using hashing, each comparison only takes $O(1)$ time, because only hash values of the strings are compared. This results in an algorithm with time complexity $O(n)$, which is the best possible time complexity for this problem.

By combining hashing and *binary search*, it is also possible to find out the lexicographic order of two strings in logarithmic time. This can be done by calculating the length of the common prefix of the strings using binary search. Once we know the length of the common prefix, we can just check the next character after the prefix, because this determines the order of the strings.

Collisions and parameters

An evident risk when comparing hash values is a **collision**, which means that two strings have different contents but equal hash values. In this case, an algorithm that relies on the hash values concludes that the strings are equal, but in reality they are not, and the algorithm may give incorrect results.

Collisions are always possible, because the number of different strings is larger than the number of different hash values. However, the probability of a collision is small if the constants A and B are carefully chosen. A usual way is to choose random constants near 10^9, for example as follows:

$$A = 911382323$$
$$B = 972663749$$

Using such constants, the `long long` type can be used when calculating the hash values, because the products AB and BB will fit in `long long`. But is it enough to have about 10^9 different hash values?

Let us consider three scenarios where hashing can be used:

Scenario 1: Strings x and y are compared with each other. The probability of a collision is $1/B$ assuming that all hash values are equally probable.

Scenario 2: A string x is compared with strings $y_1, y_2, \ldots, y_n$. The probability of one or more collisions is

$$1 - (1 - \frac{1}{B})^n.$$

Scenario 3: Strings $x_1, x_2, \ldots, x_n$ are compared with each other. The probability of one or more collisions is

$$1 - \frac{B \cdot (B-1) \cdot (B-2) \cdots (B-n+1)}{B^n}.$$

The following table shows the collision probabilities when $n = 10^6$ and the value of B varies:

constant B	scenario 1	scenario 2	scenario 3
10^3	0.001000	1.000000	1.000000
10^6	0.000001	0.632121	1.000000
10^9	0.000000	0.001000	1.000000
10^{12}	0.000000	0.000000	0.393469
10^{15}	0.000000	0.000000	0.000500
10^{18}	0.000000	0.000000	0.000001

The table shows that in scenario 1, the probability of a collision is negligible when $B \approx 10^9$. In scenario 2, a collision is possible but the probability is still quite small. However, in scenario 3 the situation is very different: a collision will almost always happen when $B \approx 10^9$.

The phenomenon in scenario 3 is known as the **birthday paradox**: if there are n people in a room, the probability that some two people have the same birthday is large even if n is quite small. In hashing, correspondingly, when all hash values are compared with each other, the probability that some two hash values are equal is large.

We can make the probability of a collision smaller by calculating *multiple* hash values using different parameters. It is unlikely that a collision would occur in all hash values at the same time. For example, two hash values with parameter $B \approx 10^9$ correspond to one hash value with parameter $B \approx 10^{18}$, which makes the probability of a collision very small.

Some people use constants $B = 2^{32}$ and $B = 2^{64}$, which is convenient, because operations with 32 and 64 bit integers are calculated modulo 2^{32} and 2^{64}. However, this is not a good choice, because it is possible to construct inputs that always generate collisions when constants of the form 2^x are used [34].

26.4 Z-algorithm

The **Z-array** of a string gives for each position k in the string the length of the longest substring that begins at position k and is a prefix of the string. Such an array can be efficiently constructed using the **Z-algorithm**[2].

For example, the Z-array for the string ACBACDACBACBACDA is as follows:

1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
A	C	B	A	C	D	A	C	B	A	C	B	A	C	D	A
−	0	0	2	0	0	5	0	0	7	0	0	2	0	0	1

---

[2] The Z-algorithm was presented in [22] as the simplest known method for linear-time pattern matching, and the original idea was attributed to [32].

For example, the value at position 7 in the above Z-array is 5, because the substring `ACBAC` of length 5 is a prefix of the string, but the substring `ACBACB` of length 6 is not a prefix of the string.

It is often a matter of taste whether to use string hashing or the Z-algorithm. Unlike hashing, the Z-algorithm always works and there is no risk for collisions. On the other hand, the Z-algorithm is more difficult to implement and some problems can only be solved using hashing.

Algorithm description

The Z-algorithm scans the string from left to right, and calculates for each position the length of the longest substring that is a prefix of the string. A straightforward algorithm would have a time complexity of $O(n^2)$, but the Z-algorithm has an important optimization which ensures that the time complexity is only $O(n)$.

The idea is to maintain a range $[x, y]$ such that the substring from x to y is a prefix of the string and y is as large as possible. Since the Z-array already contains information about the characters in the range $[x, y]$, we can use this information to calculate values for elements in the range $[x, y]$.

The time complexity of the Z-algorithm is $O(n)$, because the algorithm only compares strings character by character starting at position $y + 1$. If the characters match, the value of y increases, and it is not needed to compare the character at position y again but the information in the Z-array can be used.

For example, let us construct the following Z-array:

1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
A	C	B	A	C	D	A	C	B	A	C	B	A	C	D	A
−	?	?	?	?	?	?	?	?	?	?	?	?	?	?	?

The first interesting position is 7 where the length of the common prefix is 5. After calculating this value, the current $[x, y]$ range will be $[7, 11]$:

1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
A	C	B	A	C	D	A	C	B	A	C	B	A	C	D	A
−	0	0	2	0	0	5	?	?	?	?	?	?	?	?	?

Now, it is possible to calculate the subsequent values of the Z-array more efficiently, because we know that the ranges $[1, 5]$ and $[7, 11]$ contain the same characters. First, since the values at positions 2 and 3 are 0, we immediately know that the values at positions 8 and 9 are also 0:

1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
A	C	B	A	C	D	A	C	B	A	C	B	A	C	D	A
–	0	0	2	0	0	5	0	0	?	?	?	?	?	?	?

After this, we know that the value at position 10 will be at least 2, because the value at position 4 is 2:

1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
A	C	B	A	C	D	A	C	B	A	C	B	A	C	D	A
–	0	0	2	0	0	5	0	0	?	?	?	?	?	?	?

Since we have no information about the characters after position 11, we have to begin to compare the strings character by character:

1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
A	C	B	A	C	D	A	C	B	A	C	B	A	C	D	A
–	0	0	2	0	0	5	0	0	?	?	?	?	?	?	?

It turns out that the length of the common prefix at position 10 is 7, and thus the new range $[x, y]$ is $[10, 16]$:

1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
A	C	B	A	C	D	A	C	B	A	C	B	A	C	D	A
–	0	0	2	0	0	5	0	0	7	?	?	?	?	?	?

After this, all subsequent values of the Z-array can be calculated using the values already stored in the array. All the remaining values can be directly retrieved from the beginning of the Z-array:

1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
A	C	B	A	C	D	A	C	B	A	C	B	A	C	D	A
–	0	0	2	0	0	5	0	0	7	0	0	2	0	0	1

Using the Z-array

As an example, let us once again consider the pattern matching problem, where our task is to find the positions where a pattern p occurs in a string s. We already solved this problem efficiently using string hashing, but the Z-algorithm provides another way to solve the problem.

A usual idea in string processing is to construct a string that consists of multiple strings separated by special characters. In this problem, we can construct a string $p\#s$, where p and s are separated by a special character # that does not occur in the strings. The Z-array of $p\#s$ tells us the positions where p occurs in s, because such positions contain the value p.

For example, if $s =$ HATTIVATTI and $p =$ ATT, the Z-array is as follows:

1	2	3	4	5	6	7	8	9	10	11	12	13	14
A	T	T	#	H	A	T	T	I	V	A	T	T	I
–	0	0	0	0	3	0	0	0	0	3	0	0	0

The positions 6 and 11 contain the value 3, which means that the pattern ATT occurs in the corresponding positions in the string HATTIVATTI.

The time complexity of the resulting algorithm is $O(n)$, because it suffices to construct the Z-array and go through its values.

Chapter 27

Square root algorithms

A **square root algorithm** is an algorithm that has a square root in its time complexity. A square root can be seen as a "poor man's logarithm": the complexity $O(\sqrt{n})$ is better than $O(n)$ but worse than $O(\log n)$. In any case, many square root algorithms are fast in practice and have small constant factors.

As an example, let us consider the problem of creating a data structure that supports two operations on an array: modifying an element at a given position and calculating the sum of elements in the given range.

We have previously solved the problem using a binary indexed tree and segment tree, that support both operations in $O(\log n)$ time. However, now we will solve the problem in another way using a square root structure that allows us to modify elements in $O(1)$ time and calculate sums in $O(\sqrt{n})$ time.

The idea is to divide the array into blocks of size $\sqrt{n}$ so that each block contains the sum of elements inside the block. For example, an array of 16 elements will be divided into blocks of 4 elements as follows:

21				17				20				13			
5	8	6	3	2	7	2	6	7	1	7	5	6	2	3	2

Using this structure, it is easy to modify the array, because it is only needed to update the sum of a single block after each modification, which can be done in $O(1)$ time. For example, the following picture shows how the value of an element and the sum of the corresponding block change:

21				15				20				13			
5	8	6	3	2	5	2	6	7	1	7	5	6	2	3	2

Calculating the sum of elements in a range is a bit more difficult. It turns out that we can always divide the range into three parts such that the sum consists of values of single elements and sums of blocks between them:

21				15				20				13			
5	8	6	3	2	5	2	6	7	1	7	5	6	2	3	2

Since the number of single elements is $O(\sqrt{n})$ and also the number of blocks is $O(\sqrt{n})$, the time complexity of the sum query is $O(\sqrt{n})$. Thus, the parameter $\sqrt{n}$ balances two things: the array is divided into $\sqrt{n}$ blocks, each of which contains $\sqrt{n}$ elements.

In practice, it is not needed to use the exact value of $\sqrt{n}$ as a parameter, but it may be better to use parameters k and n/k where k is different from $\sqrt{n}$. The optimal parameter depends on the problem and input. For example, if an algorithm often goes through the blocks but rarely inspects single elements inside the blocks, it may be a good idea to divide the array into $k < \sqrt{n}$ blocks, each of which contains $n/k > \sqrt{n}$ elements.

27.1 Batch processing

Sometimes the operations of an algorithm can be divided into batches, each of which can be processed separately. Some precalculation is done between the batches in order to process the future operations more efficiently. If there are $O(\sqrt{n})$ batches of size $O(\sqrt{n})$, this results in a square root algorithm.

As an example, let us consider a problem where a grid of size $k \times k$ initially consists of white squares and our task is to perform n operations, each of which is one of the following:

- paint square (y, x) black

- find the nearest black square to square (y, x) where the distance between squares (y_1, x_1) and (y_2, x_2) is $|y_1 - y_2| + |x_1 - x_2|$

We can solve the problem by dividing the operations into $O(\sqrt{n})$ batches, each of which consists of $O(\sqrt{n})$ operations. At the beginning of each batch, we calculate for each square of the grid the smallest distance to a black square. This can be done in $O(k^2)$ time using breadth-first search.

When processing a batch, we maintain a list of squares that have been painted black in the current batch. The list contains $O(\sqrt{n})$ elements, because there are $O(\sqrt{n})$ operations in each batch. Now, the distance from a square to the nearest black square is either the precalculated distance or the distance to a square that appears in the list.

The algorithm works in $O((k^2 + n)\sqrt{n})$ time. First, there are $O(\sqrt{n})$ breadth-first searches and each search takes $O(k^2)$ time. Second, the total number of squares processed during the algorithm is $O(n)$, and at each square, we go through a list of $O(\sqrt{n})$ squares.

If the algorithm would perform a breadth-first search at each operation, the time complexity would be $O(k^2 n)$. And if the algorithm would go through all painted squares at each operation, the time complexity would be $O(n^2)$. Thus, the time complexity of the square root algorithm is a combination of these time complexities, but in addition, a factor of n is replaced by $\sqrt{n}$.

27.2 Subalgorithms

Some square root algorithms consists of subalgorithms that are specialized for different input parameters. Typically, there are two subalgorithms: one algorithm is efficient when some parameter is smaller than $\sqrt{n}$, and another algorithm is efficient when the parameter is larger than $\sqrt{n}$.

As an example, let us consider a problem where we are given a tree of n nodes, each with some color. Our task is to find two nodes that have the same color and whose distance is as large as possible.

For example, in the following tree, the maximum distance is 4 between the red nodes 3 and 4:
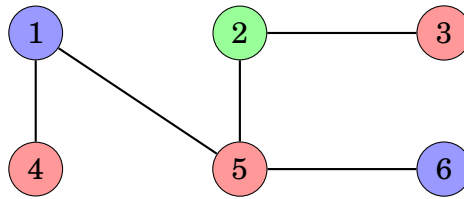
The problem can be solved by going through all colors and calculating the maximum distance between two nodes separately for each color. Assume that the current color is x and there are c nodes whose color is x. There are two subalgorithms that are specialized for small and large values of c:

Case 1: $c \le \sqrt{n}$. If the number of nodes is small, we go through all pairs of nodes whose color is x and select the pair that has the maximum distance. For each node, it is needed to calculate the distance to $O(\sqrt{n})$ other nodes (see Chapter 18.3), so the total time needed for processing all nodes is $O(n\sqrt{n})$.

Case 2: $c > \sqrt{n}$. If the number of nodes is large, we go through the whole tree and calculate the maximum distance between two nodes with color x. The time complexity of the tree traversal is $O(n)$, and this will be done at most $O(\sqrt{n})$ times, so the total time needed is $O(n\sqrt{n})$.

The time complexity of the algorithm is $O(n\sqrt{n})$, because both cases take a total of $O(n\sqrt{n})$ time.

27.3 Mo's algorithm

Mo's algorithm [1] can be used in many problems that require processing range queries in a *static* array. Before processing the queries, the algorithm sorts them in a special order which guarantees that the algorithm works efficiently.

At each moment in the algorithm, there is an active range and the algorithm maintains the answer to a query related to that range. The algorithm processes the queries one by one, and always moves the endpoints of the active range by inserting and removing elements. The time complexity of the algorithm is $O(n\sqrt{n}f(n))$ when the array contains n elements, there are n queries and each insertion and removal of an element takes $O(f(n))$ time.

---

[1]According to [8], this algorithm is named after Mo Tao, a Chinese competitive programmer. However, the technique has appeared earlier in the literature [30].

The trick in Mo's algorithm is the order in which the queries are processed: The array is divided into blocks of $O(\sqrt{n})$ elements, and the queries are sorted primarily by the number of the block that contains the first element in the range, and secondarily by the position of the last element in the range. It turns out that using this order, the algorithm only performs $O(n\sqrt{n})$ operations, because the left endpoint of the range moves n times $O(\sqrt{n})$ steps, and the right endpoint of the range moves $\sqrt{n}$ times $O(n)$ steps. Thus, both endpoints move a total of $O(n\sqrt{n})$ steps during the algorithm.

Example

As an example, consider a problem where we are given a set of queries, each of them corresponding to a range in an array, and our task is to calculate for each query the number of *distinct* elements in the range.

In Mo's algorithm, the queries are always sorted in the same way, but it depends on the problem how the answer to the query is maintained. In this problem, we can maintain an array c where c[x] indicates the number of times an element x occurs in the active range.

When we move from one query to another query, the active range changes. For example, if the current range is

4	2	5	4	2	4	3	3	4

and the next range is

4	2	5	4	2	4	3	3	4

there will be three steps: the left endpoint moves one step to the left, and the right endpoint moves two steps to the right.

After each step, the array c needs to be updated. After adding an element x, we increase the value of c[x] by one, and if c[x] = 1 after this, we also increase the answer to the query by one. Similarly, after removing an element x, we decrease the value of c[x] by one, and if c[x] = 0 after this, we also decrease the answer to the query by one.

In this problem, the time needed to perform each step is $O(1)$, so the total time complexity of the algorithm is $O(n\sqrt{n})$.

Chapter 28

Segment trees revisited

A segment tree is a versatile data structure that can be used in many different situations. However, there are many topics related to segment trees that we have not touched yet. Now it is time to discuss some more advanced variants of segment trees.

So far, we have implemented the operations of a segment tree by walking *from bottom to top* in the tree. For example, we have calculated the sum of elements in a range $[a,b]$ as follows (Chapter 9.3):

```
int sum(int a, int b) {
    a += N; b += N;
    int s = 0;
    while (a <= b) {
        if (a%2 == 1) s += p[a++];
        if (b%2 == 0) s += p[b--];
        a /= 2; b /= 2;
    }
    return s;
}
```

However, in more advanced segment trees, it is often needed to implement the operations in another way, *from top to bottom*. Using this approach, the function becomes as follows:

```
int sum(int a, int b, int k, int x, int y) {
    if (b < x || a > y) return 0;
    if (a <= x && y <= b) return p[k];
    int d = (x+y)/2;
    return sum(a,b,2*k,x,d) + sum(a,b,2*k+1,d+1,y);
}
```

Now we can calculate the sum of elements in $[a,b]$ as follows:

```
int s = sum(a, b, 1, 0, N-1);
```

The parameter k indicates the current position in p. Initially k equals 1,

because we begin at the root of the segment tree. The range $[x,y]$ corresponds to k and is initially $[0,N-1]$. When calculating the sum, if $[x,y]$ is outside $[a,b]$, the sum is 0, and if $[x,y]$ is completely inside $[a,b]$, the sum can be found in p. If $[x,y]$ is partially inside $[a,b]$, the search continues recursively to the left and right half of $[x,y]$. The left half is $[x,d]$ and the right half is $[d+1,y]$ where $d = \lfloor \frac{x+y}{2} \rfloor$.

The following picture shows how the search proceeds when calculating the sum of elements in $[a,b]$. The gray nodes indicate nodes where the recursion stops and the sum can be found in p.
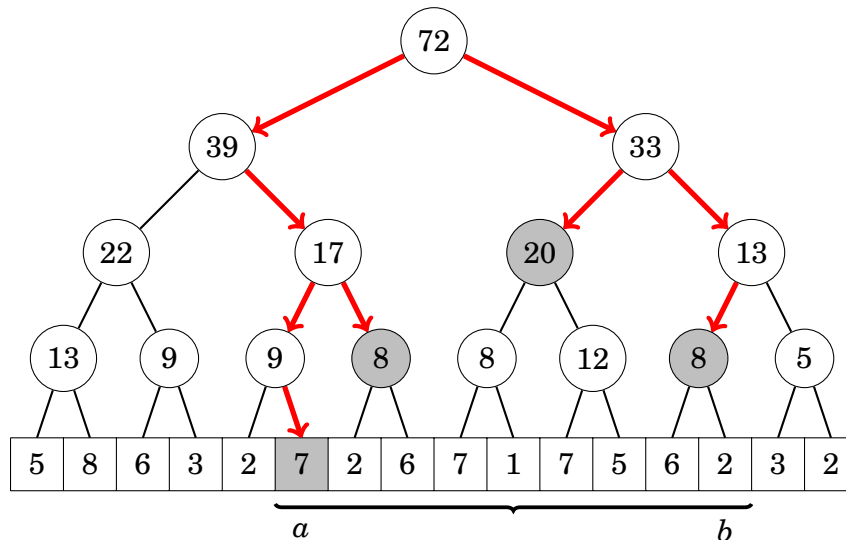
Also using this implementation, operations take $O(\log n)$ time, because the total number of visited nodes is $O(\log n)$.

28.1 Lazy propagation

Using **lazy propagation**, we can build a segment tree that supports both range updates and range queries in $O(\log n)$ time. The idea is to perform updates and queries from top to bottom and perform updates *lazily* so that they are propagated down the tree only when it is necessary.

In a lazy segment tree, nodes contain two types of information. Like in an ordinary segment tree, each node contains the sum or some other value related to the corresponding subarray. In addition, the node may contain information related to lazy updates, which has not been propagated to its children.

There are two types of range updates: each element in the range is either *increased* by some value or *assigned* some value. Both operations can be implemented using similar ideas, and it is even possible to construct a tree that supports both operations at the same time.

Lazy segment trees

Let us consider an example where our goal is to construct a segment tree that supports two operations: increasing each element in $[a,b]$ by u and calculating the sum of elements in $[a,b]$.

We will construct a tree where each node contains two values s/z: s denotes the sum of elements in the range and z denotes the value of a lazy update, which means that all elements in the range should be increased by z. In the following tree, $z = 0$ for all nodes, so there are no ongoing lazy updates.
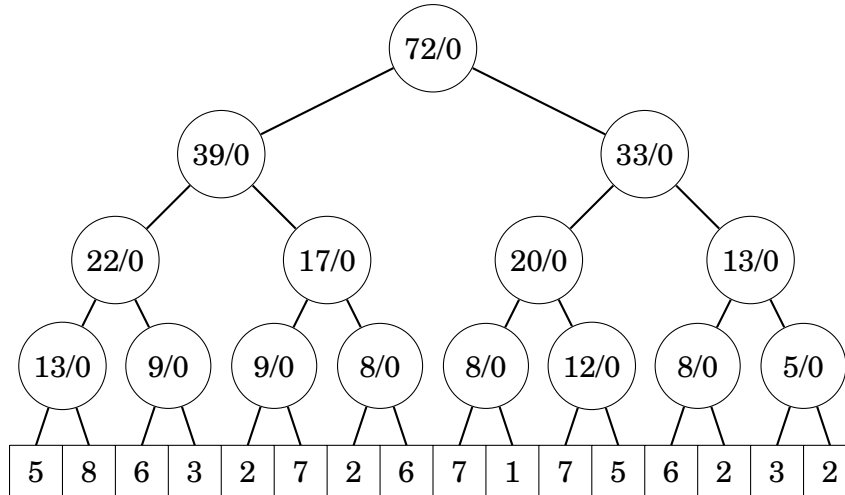
When the elements in $[a,b]$ are increased by u, we walk from the root towards the leaves and modify the nodes in the tree as follows: If the range $[x,y]$ of a node is completely inside the range $[a,b]$, we increase the z value of the node by u and stop. If $[x,y]$ only partially belongs to $[a,b]$, we increase the s value of the node by hu, where h is the size of the intersection of $[a,b]$ and $[x,y]$, and continue our walk recursively in the tree.

For example, the following picture shows the tree after increasing the elements in $[a,b]$ by 2:
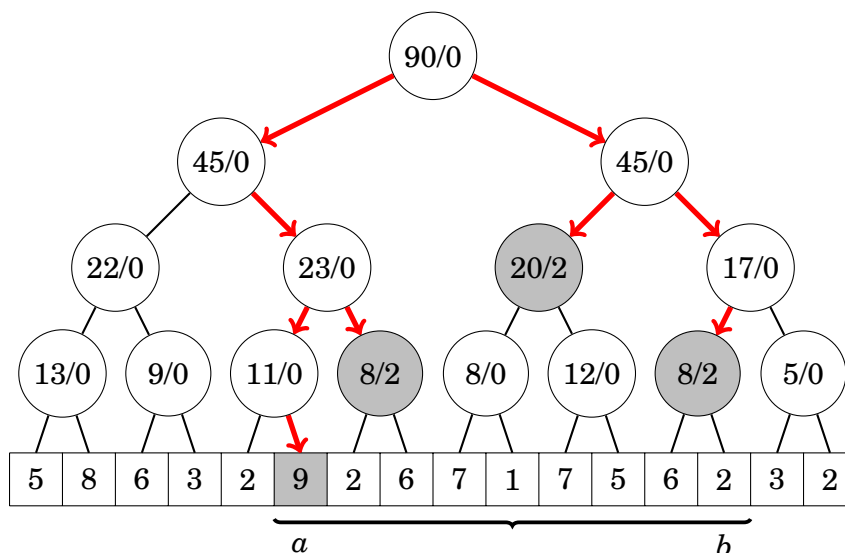
We also calculate the sum of elements in a range $[a,b]$ by walking in the tree from top to bottom. If the range $[x,y]$ of a node completely belongs to $[a,b]$, we add the s value of the node to the sum. Otherwise, we continue the search recursively downwards in the tree.

Both in updates and queries, the value of a lazy update is always propagated to the children of the node before processing the node. The idea is that updates will be propagated downwards only when it is necessary, which guarantees that the operations are always efficient.

The following picture shows how the tree changes when we calculate the sum of elements in $[a,b]$. The rectangle shows the nodes whose values change, because a lazy update is propagated downwards, which is necessary to calculate the sum in $[a,b]$.
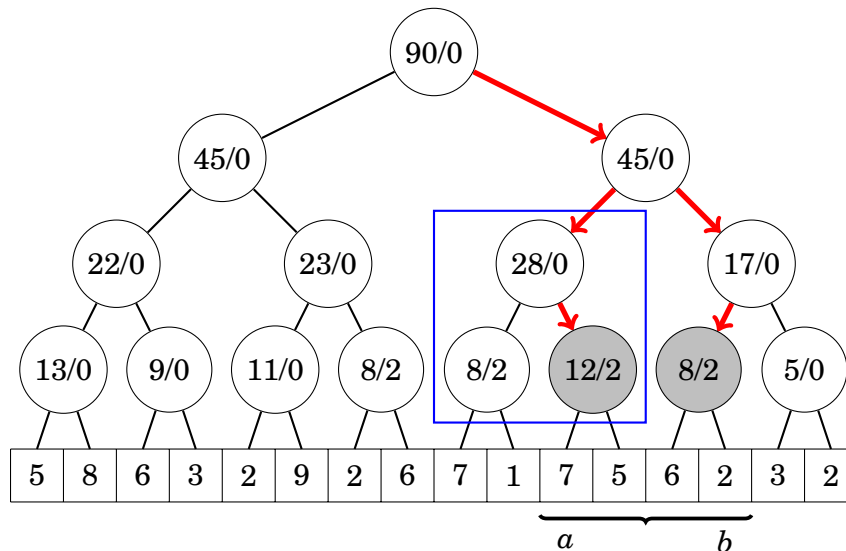
Note that sometimes it is needed to combine lazy updates. This happens when a node that already has a lazy update is assigned another lazy update. In this problem, it is easy to combine lazy updates, because the combination of updates z_1 and z_2 corresponds to an update $z_1 + z_2$.

Polynomial updates

Lazy updates can be generalized so that it is possible to update ranges using polynomials of the form

$$p(u) = t_k u^k + t_{k-1} u^{k-1} + \cdots + t_0.$$

In this case, the update for an element i in the range $[a,b]$ is $p(i-a)$. For example, adding $p(u) = u + 1$ to $[a,b]$ means that the element at position a is increased by 1, the element at position $a+1$ is increased by 2, and so on.

To support polynomial updates, each node is assigned $k+2$ values, where k equals the degree of the polynomial. The value s is the sum of the elements in the range, and the values $z_0, z_1, \ldots, z_k$ are the coefficients of a polynomial that corresponds to a lazy update.

Now, the sum of elements in a range $[x,y]$ equals

$$s + \sum_{u=0}^{y-x} z_k u^k + z_{k-1} u^{k-1} + \cdots + z_0.$$

The value of such a sum can be efficiently calculated using sum formulas. For example, the term z_0 corresponds to the sum $(y - x + 1)z_0$, and the term $z_1 u$ corresponds to the sum

$$z_1(0 + 1 + \cdots + y - x) = z_1 \frac{(y-x)(y-x+1)}{2}.$$

When propagating an update in the tree, the indices of $p(u)$ change, because in each range $[x, y]$, the values are calculated for $u = 0, 1, \ldots, y - x$. However, this is not a problem, because $p'(u) = p(u + h)$ is a polynomial of equal degree as $p(u)$. For example, if $p(u) = t_2 u^2 + t_1 u - t_0$, then

$$p'(u) = t_2(u + h)^2 + t_1(u + h) - t_0 = t_2 u^2 + (2ht_2 + t_1)u + t_2 h^2 + t_1 h - t_0.$$

28.2 Dynamic trees

An ordinary segment tree is static, which means that each node has a fixed position in the array and the tree requires a fixed amount of memory. However, if most nodes are not used, memory is wasted. In a **dynamic segment tree**, memory is allocated only for nodes that are actually needed.

The nodes of a dynamic tree can be represented as structs:

```
struct node {
    int s;
    int x, y;
    node *l, *r;
    node(int s, int x, int y) : s(s), x(x), y(y) {}
};
```

Here s is the value of the node, $[x, y]$ is the corresponding range, and l and r point to the left and right subtree.

After this, nodes can be created as follows:

```
// create new node
node *u = new node(0, 0, 15);
// change value
u->s = 5;
```

Sparse segment trees

A dynamic segment tree is useful if the underlying array is *sparse*. This means that the range $[0, N - 1]$ of allowed indices is large, but only a small portion of the indices are used and most elements in the array are empty. While an ordinary segment tree uses $O(N)$ memory, a dynamic segment tree only requires $O(n \log N)$ memory, where n is the number of operations performed.

A **sparse segment tree** is initially empty and its only node is $[0, N - 1]$. After updates, new nodes are added dynamically when needed. For example, if $N = 16$

and the elements in positions 3 and 10 have been modified, the tree contains the following nodes:
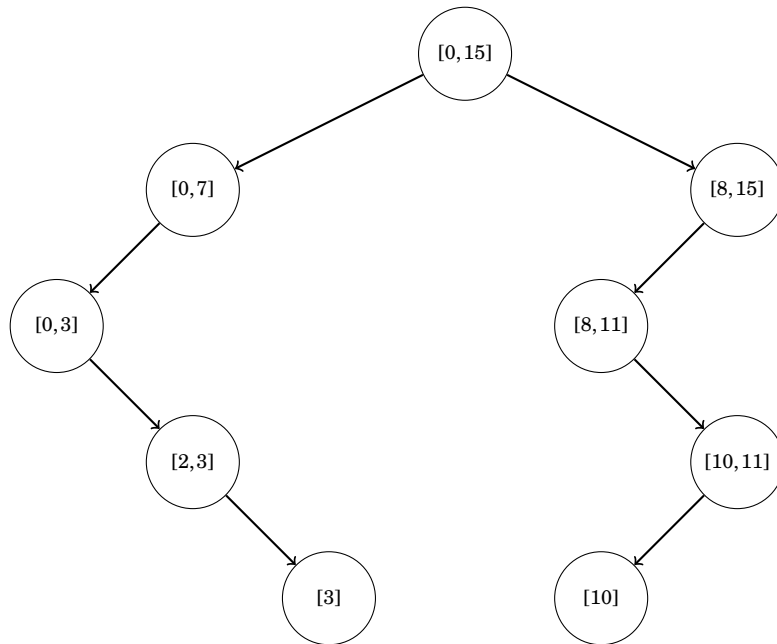
Any path from the root node to a leaf contains $O(\log N)$ nodes, so each operation adds at most $O(\log N)$ new nodes to the tree. Thus, after n operations, the tree contains at most $O(n \log N)$ nodes.

Note that if all indices of the elements are known at the beginning of the algorithm, a dynamic segment tree is not needed, but we can use an ordinary segment tree with index compression (Chapter 9.4). However, this is not possible if the indices are generated during the algorithm.

Persistent segment trees

Using a dynamic implementation, it is also possible to create a **persistent segment tree** that stores the *modification history* of the tree. In such an implementation, we can efficiently access all versions of the tree that have existed during the algorithm.

When the modification history is available, we can perform queries in any previous tree like in an ordinary segment tree, because the full structure of each tree is stored. We can also create new trees based on previous trees and modify them independently.

Consider the following sequence of updates, where red nodes change and other nodes remain the same:

step 1   step 2   step 3
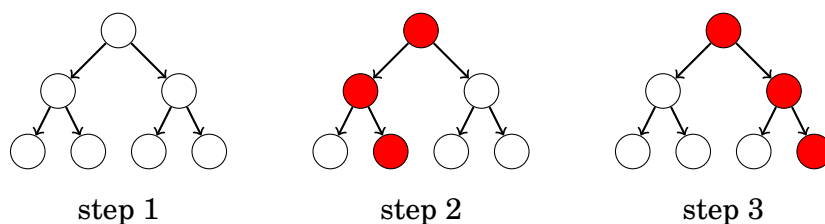
After each update, most nodes in the tree remain the same, so an efficient way to store the modification history is to represent each tree in the history as a combination of new nodes and subtrees of previous trees. In this example, the modification history can be stored as follows:
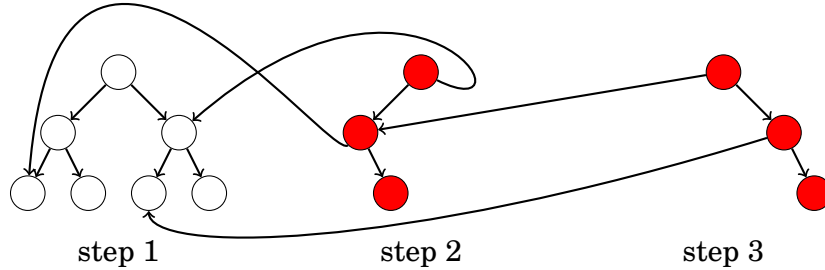
The structure of each previous tree can be reconstructed by following the pointers starting at the corresponding root node. Since each operation during the algorithm adds only $O(\log N)$ new nodes to the tree, it is possible to store the full modification history of the tree.

28.3 Data structures

Instead of single values, nodes in a segment tree can also contain data structures that maintain information about the corresponding ranges. In such a tree, the operations take $O(f(n) \log n)$ time, where $f(n)$ is the time needed for processing a single node during an operation.

As an example, consider a segment tree that supports queries of the form "how many times does an element x appear in the range $[a, b]$?" For example, the element 1 appears three times in the following range:

The idea is to build a segment tree where each node is assigned a data structure that gives the number of any element x in the corresponding range. Using such a segment tree, the answer to a query can be calculated by combining the results from the nodes that belong to the range.

For example, the following segment tree corresponds to the above array:

253
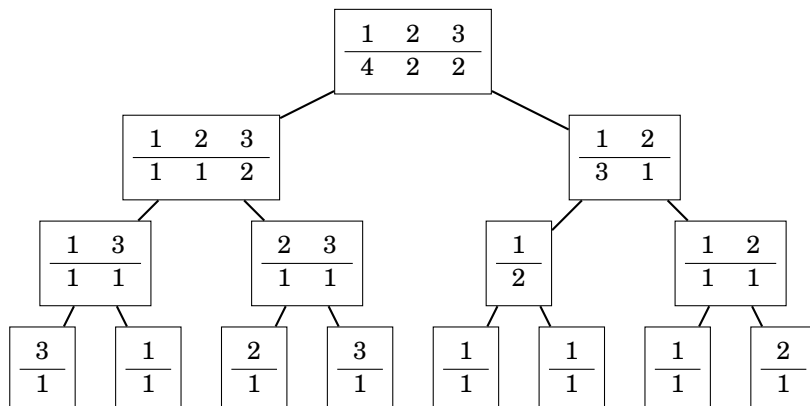
We can build the tree so that each node contains a `map` structure. In this case, the time needed for processing each node is $O(\log n)$, so the total time complexity of a query is $O(\log^2 n)$. The tree uses $O(n \log n)$ memory, because there are $O(\log n)$ levels and each level contains $O(n)$ elements.

28.4 Two-dimensionality

A **two-dimensional segment tree** supports queries related to rectangular sub-arrays of a two-dimensional array. Such a tree can be implemented as nested segment trees: a big tree corresponds to the rows in the array, and each node contains a small tree that corresponds to a column.

For example, in the array

7	6	1	6
8	7	5	2
3	9	7	1
8	5	3	8

the sum of any subarray can be calculated from the following segment tree:
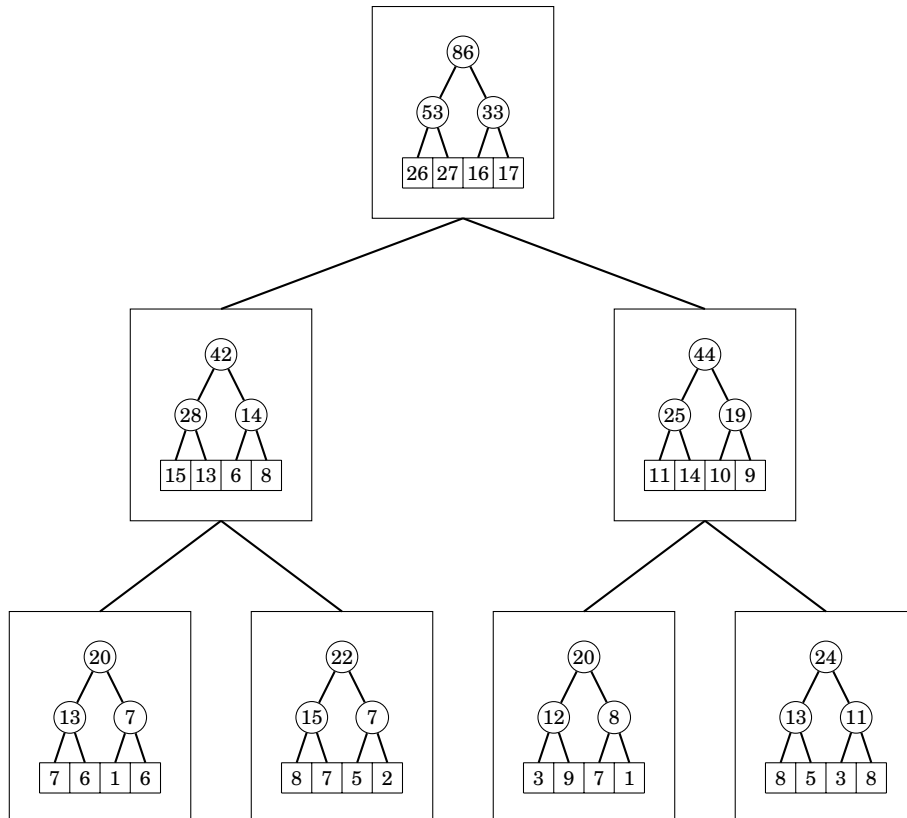
The operations in a two-dimensional segment tree take $O(\log^2 n)$ time, because the big tree and each small tree consist of $O(\log n)$ levels. The tree requires $O(n^2)$ memory, because each small tree contains $O(n)$ values.

Chapter 29

Geometry

In geometric problems, it is often challenging to find a way to approach the problem so that the solution to the problem can be conveniently implemented and the number of special cases is small.

As an example, consider a problem where we are given the vertices of a quadrilateral (a polygon that has four vertices), and our task is to calculate its area. For example, a possible input for the problem is as follows:
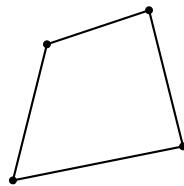
One way to approach the problem is to divide the quadrilateral into two triangles by a straight line between two opposite vertices:
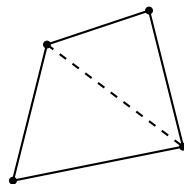
After this, it suffices to sum the areas of the triangles. The area of a triangle can be calculated, for example, using **Heron's formula**

$$\sqrt{s(s-a)(s-b)(s-c)},$$

where a, b and c are the lengths of the triangle's sides and $s = (a+b+c)/2$.

This is a possible way to solve the problem, but there is one pitfall: how to divide the quadrilateral into triangles? It turns out that sometimes we cannot just pick two arbitrary opposite vertices. For example, in the following situation, the division line is outside the quadrilateral:
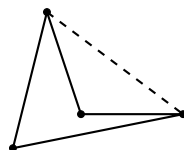
However, another way to draw the line works:
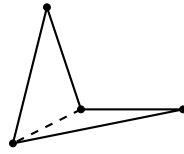
It is clear for a human which of the lines is the correct choice, but the situation is difficult for a computer.

However, it turns out that we can solve the problem using another method that is much easier to implement and does not involve any special cases. Namely, there is a general formula

$$x_1 y_2 - x_2 y_1 + x_2 y_3 - x_3 y_2 + x_3 y_4 - x_4 y_3 + x_4 y_1 - x_1 y_4,$$

that calculates the area of a quadrilateral whose vertices are (x_1, y_1), (x_2, y_2), (x_3, y_3) and (x_4, y_4). This formula is easy to implement, there are no special cases, and it turns out that we can even generalize the formula to *all* polygons.

29.1   Complex numbers

A **complex number** is a number of the form $x + yi$, where $i = \sqrt{-1}$ is the **imaginary unit**. A geometric interpretation of a complex number is that it represents a two-dimensional point (x, y) or a vector from the origin to a point (x, y).

For example, $4 + 2i$ corresponds to the following point and vector:
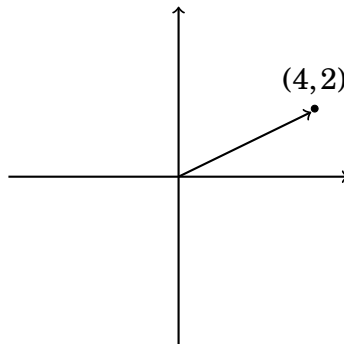
The complex number class `complex` in C++ is useful when solving geometric problems. Using the class we can represent points and vectors as complex numbers, and the class also contains tools that are useful in geometry.

In the following code, `C` is the type of a coordinate and `P` is the type of a point or vector. In addition, the code defines the macros `X` and `Y` that can be used to refer to x and y coordinates.

```
typedef long long C;
typedef complex<C> P;
#define X real()
#define Y imag()
```

For example, the following code defines a point $p = (4, 2)$ and prints its x and y coordinates:

```
P p = {4,2};
cout << p.X << " " << p.Y << "\n"; // 4 2
```

The following code defines vectors $v = (3, 1)$ and $u = (2, 2)$, and after that calculates the sum $s = v + u$.

```
P v = {3,1};
P u = {2,2};
P s = v+u;
cout << s.X << " " << s.Y << "\n"; // 5 3
```

An appropriate coordinate type is `long long` (integer) or `long double` (real number), depending on the situation. Integers should be used whenever possible, because calculations with integers are exact. If real numbers are needed, precision errors should be taken into account when comparing them. A safe way to check if numbers a and b are equal is to compare them using $|a - b| < \epsilon$, where ϵ is a small number (for example, $\epsilon = 10^{-9}$).

Functions

In the following examples, the coordinate type is `long double`.

The function `abs(v)` calculates the length $|v|$ of a vector $v = (x, y)$ using the formula $\sqrt{x^2 + y^2}$. The function can also be used for calculating the distance between points (x_1, y_1) and (x_2, y_2), because that distance equals the length of the vector $(x_2 - x_1, y_2 - y_1)$.

The following code calculates the distance between points $(4, 2)$ and $(3, -1)$:

```
P a = {4,2};
P b = {3,-1};
cout << abs(b-a) << "\n"; // 3.60555
```

The function `arg(v)` calculates the angle of a vector $v = (x, y)$ with respect to the x axis. The function gives the angle in radians, where r radians equals $180r/\pi$ degrees. The angle of a vector that points to the right is 0, and angles decrease clockwise and increase counterclockwise.

The function `polar(s,a)` constructs a vector whose length is s and that points to an angle a. In addition, a vector can be rotated by an angle a by multiplying it by a vector with length 1 and angle a.

The following code calculates the angle of the vector $(4, 2)$, rotates it 1/2 radians counterclockwise, and then calculates the angle again:

```
P v = {4,2};
cout << arg(v) << "\n"; // 0.463648
v *= polar(1.0,0.5);
cout << arg(v) << "\n"; // 0.963648
```

29.2 Points and lines

The **cross product** $a \times b$ of vectors $a = (x_1, y_1)$ and $b = (x_2, y_2)$ equals $x_1 y_2 - x_2 y_1$. The cross product tells us whether b turns left (positive value), does not turn (zero) or turns to right (negative value) when it is placed directly after a.

The following picture illustrates the above cases:

$$a \times b = 6 \qquad a \times b = 0 \qquad a \times b = -8$$
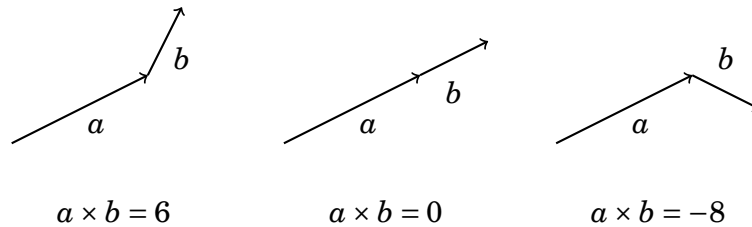
For example, in the first picture $a = (4,2)$ and $b = (1,2)$. The following code calculates the cross product using the class `complex`:

```
P a = {4,2};
P b = {1,2};
C r = (conj(a)*b).Y; // 6
```

The above code works, because the function `conj` negates the y coordinate of a vector, and when the vectors $(x_1, -y_1)$ and (x_2, y_2) are multiplied together, the y coordinate of the result is $x_1 y_2 - x_2 y_1$.

Point location

Cross products can be used for testing whether a point is located on the left or right side of a line. Assume that the line goes through points s_1 and s_2, we are looking from s_1 to s_2 and the point is p.

For example, in the following picture, p is on the left side of the line:
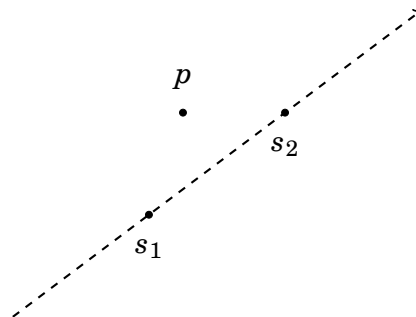
In this situation, the cross product $(p - s_1) \times (p - s_2)$ tells us the location of the point p. If the cross product is positive, p is located on the left side, and if the cross product is negative, p is located on the right side. Finally, if the cross product is zero, points s_1, s_2 and p are on the same line.

Line segment intersection

Consider a problem where we are given two line segments ab and cd and our task is to check whether they intersect. The possible cases are:

Case 1: The line segments are on the same line and they overlap each other. In this case, there is an infinite number of intersection points. For example, in the following picture, all points between c and b are intersection points:
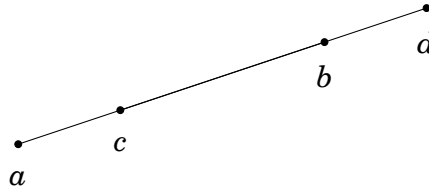
In this case, we can use cross products to check if all points are on the same line. After this, we can sort the points and check whether the line segments overlap each other.

Case 2: The line segments have a common vertex that is the only intersection point. For example, in the following picture the intersection point is $b = c$:
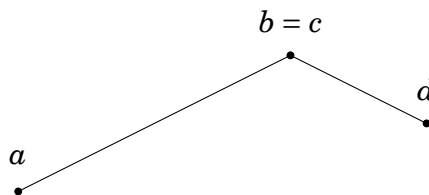
This case is easy to check, because there are only four possibilities for the intersection point: $a = c$, $a = d$, $b = c$ and $b = d$.

Case 3: There is exactly one intersection point that is not a vertex of any line segment. In the following picture, the point p is the intersection point:
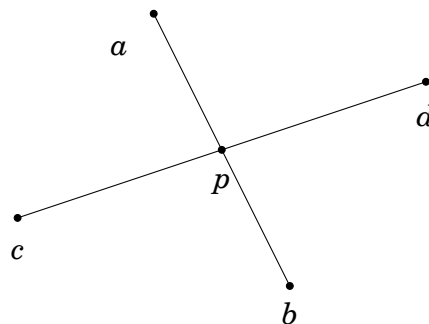
In this case, the line segments intersect exactly when both points c and d are on different sides of a line through a and b, and points a and b are on different sides of a line through c and d. Hence, we can use cross products to check this.

Point distance from a line

The area of a triangle can be calculated using the formula

$$\frac{|(a-c) \times (b-c)|}{2},$$

where a, b and c are the vertices of the triangle.

Using this formula, it is possible to calculate the shortest distance between a point and a line. For example, in the following picture d is the shortest distance between the point p and the line that is defined by the points s_1 and s_2:
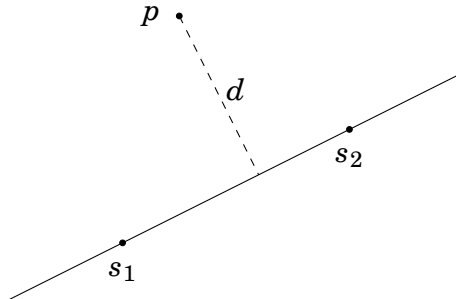
The area of the triangle whose vertices are s_1, s_2 and p can be calculated in two ways: it is both $\frac{1}{2}|s_2-s_1|d$ and $\frac{1}{2}((s_1-p)\times(s_2-p))$. Thus, the shortest distance is

$$d = \frac{(s_1-p)\times(s_2-p)}{|s_2-s_1|}.$$

Point inside a polygon

Let us now consider a problem where our task is to find out whether a point is located inside or outside a polygon. For example, in the following picture point a is inside the polygon and point b is outside the polygon.
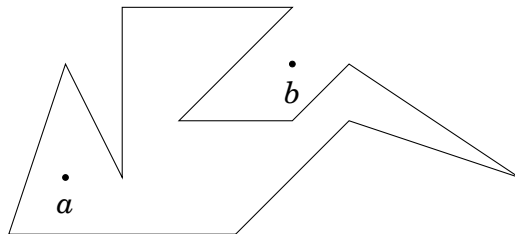
A convenient way to solve the problem is to send a ray from the point to an arbitrary direction and calculate the number of times it touches the boundary of the polygon. If the number is odd, the point is inside the polygon, and if the number is even, the point is outside the polygon.

For example, we could send the following rays:
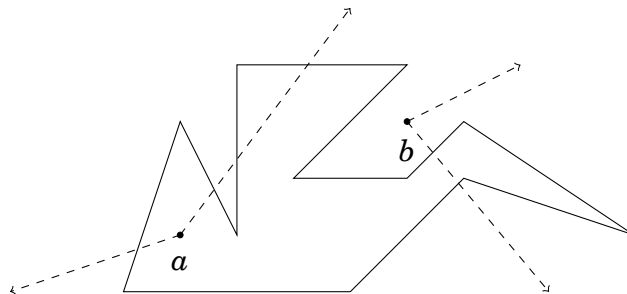
The rays from a touch 1 and 3 times the boundary of the polygon, so a is inside the polygon. Correspondingly, the rays from b touch 0 and 2 times the

boundary of the polygon, so b is outside the polygon.

29.3 Polygon area

A general formula for calculating the area of a polygon is

$$\frac{1}{2} | \sum_{i=1}^{n-1} (p_i \times p_{i+1}) | = \frac{1}{2} | \sum_{i=1}^{n-1} (x_i y_{i+1} - x_{i+1} y_i) |,$$

where the vertices are $p_1 = (x_1, y_1)$, $p_2 = (x_2, y_2)$, ..., $p_n = (x_n, y_n)$ in such an order that p_i and p_{i+1} are adjacent vertices on the boundary of the polygon, and the first and last vertex is the same, i.e., $p_1 = p_n$.

For example, the area of the polygon
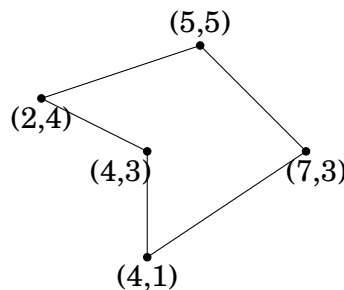
is

$$\frac{|(2 \cdot 5 - 5 \cdot 4) + (5 \cdot 3 - 7 \cdot 5) + (7 \cdot 1 - 4 \cdot 3) + (4 \cdot 3 - 4 \cdot 1) + (4 \cdot 4 - 2 \cdot 3)|}{2} = 17/2.$$

The idea in the formula is to go through trapezoids whose one side is a side of the polygon, and another side lies on the horizontal line $y = 0$. For example:
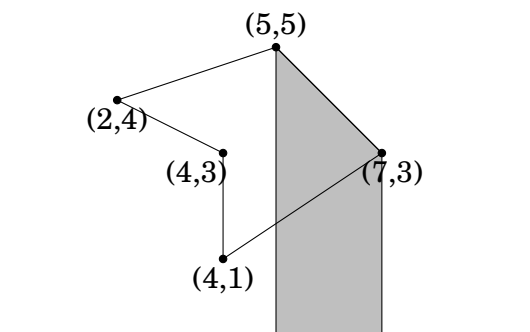
The area of such a trapezoid is

$$(x_{i+1} - x_i) \frac{y_i + y_{i+1}}{2},$$

where the vertices of the polygon are p_i and p_{i+1}. If $x_{i+1} > x_i$, the area is positive, and if $x_{i+1} < x_i$, the area is negative.

The area of the polygon is the sum of areas of all such trapezoids, which yields the formula

$$\left| \sum_{i=1}^{n-1} (x_{i+1} - x_i) \frac{y_i + y_{i+1}}{2} \right| = \frac{1}{2} \left| \sum_{i=1}^{n-1} (x_i y_{i+1} - x_{i+1} y_i) \right|.$$

Note that the absolute value of the sum is taken, because the value of the sum may be positive or negative depending on whether we walk clockwise or counterclockwise along the boundary of the polygon.

Pick's theorem

Pick's theorem provides another way to calculate the area of a polygon provided that all vertices of the polygon have integer coordinates. According to Pick's theorem, the area of the polygon is

$$a + b/2 - 1,$$

where a is the number of integer points inside the polygon and b is the number of integer points on the boundary of the polygon.

For example, the area of the polygon
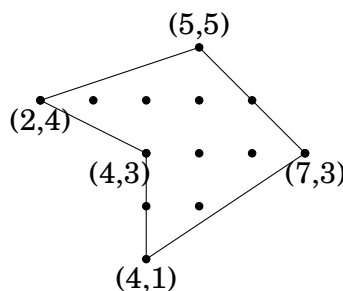
is $6 + 7/2 - 1 = 17/2$.

29.4   Distance functions

A **distance function** defines the distance between two points. The usual distance function in geometry is the **Euclidean distance** where the distance between points (x_1, y_1) and (x_2, y_2) is

$$\sqrt{(x_2 - x_1)^2 + (y_2 - y_1)^2}.$$

An alternative distance function is the **Manhattan distance** where the distance between points (x_1, y_1) and (x_2, y_2) is

$$|x_1 - x_2| + |y_1 - y_2|.$$

For example, consider the following picture:

$$\text{Euclidean distance} \qquad \text{Manhattan distance}$$
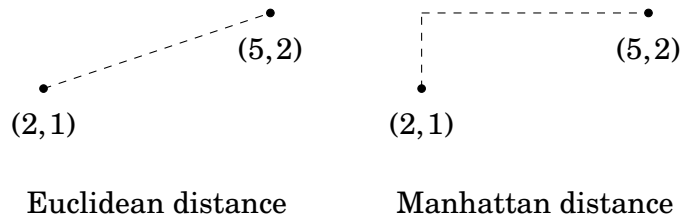
The Euclidean distance between the points is

$$\sqrt{(5-2)^2 + (2-1)^2} = \sqrt{10}$$

and the Manhattan distance is

$$|5-2| + |2-1| = 4.$$

The following picture shows regions that are within a distance of 1 from the center point, using the Euclidean and Manhattan distances:

$$\text{Euclidean distance} \qquad \text{Manhattan distance}$$
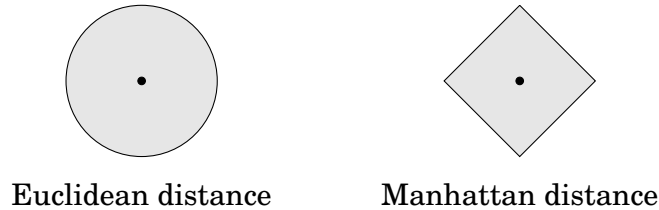
Rotating coordinates

Some problems are easier to solve if the Manhattan distance is used instead of the Euclidean distance. As an example, consider a problem where we are given n points in the two-dimensional plane and our task is to calculate the maximum Manhattan distance between any two points.

For example, consider the following set of points:
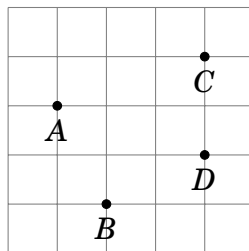
The maximum Manhattan distance is 5 between points B and C:
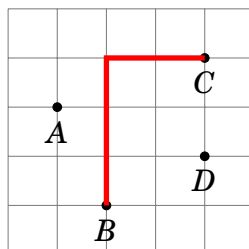
A useful technique related to Manhattan distances is to rotate all coordinates 45 degrees so that a point (x, y) becomes $(x + y, y - x)$. For example, after rotating the above points, the result is:
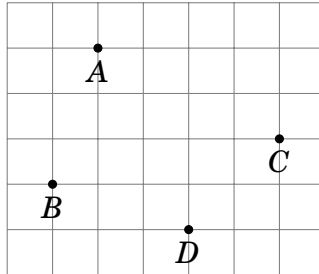
And the maximum distance is as follows:
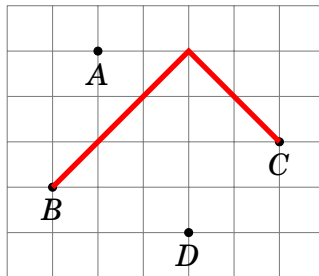
Consider two points $p_1 = (x_1, y_1)$ and $p_2 = (x_1, y_1)$ whose rotated coordinates are $p_1' = (x_1', y_1')$ and $p_2' = (x_2', y_2')$. The Manhattan distance is

$$|x_1 - x_2| + |y_1 - y_2| = \max(|x_1' - x_2'|, |y_1' - y_2'|)$$

For example, if $p_1 = (1, 0)$ and $p_2 = (3, 3)$, the rotated coordinates are $p_1' = (1, -1)$ and $p_2' = (6, 0)$ and the Manhattan distance is

$$|1 - 3| + |0 - 3| = \max(|1 - 6|, |-1 - 0|) = 5.$$

The rotated coordinates provide a simple way to operate with Manhattan distances, because we can consider x and y coordinates separately. To maximize the Manhattan distance between two points, we should find two points whose rotated coordinates maximize the value of

$$\max(|x_1' - x_2'|, |y_1' - y_2'|).$$

This is easy, because either the horizontal or vertical difference of the rotated coordinates has to be maximum.

Chapter 30

Sweep line algorithms

Many geometric problems can be solved using **sweep line** algorithms. The idea in such algorithms is to represent the problem as a set of events that correspond to points in the plane. The events are processed in increasing order according to their x or y coordinate.

As an example, let us consider a problem where there is a company that has n employees, and we know for each employee their arrival and leaving times on a certain day. Our task is to calculate the maximum number of employees that were in the office at the same time.

The problem can be solved by modeling the situation so that each employee is assigned two events that corresponds to their arrival and leaving times. After sorting the events, we can go through them and keep track of the number of people in the office. For example, the table

person	arrival time	leaving time
Uolevi	10	15
Maija	6	12
Kaaleppi	14	16
Liisa	5	13

corresponds to the following events:
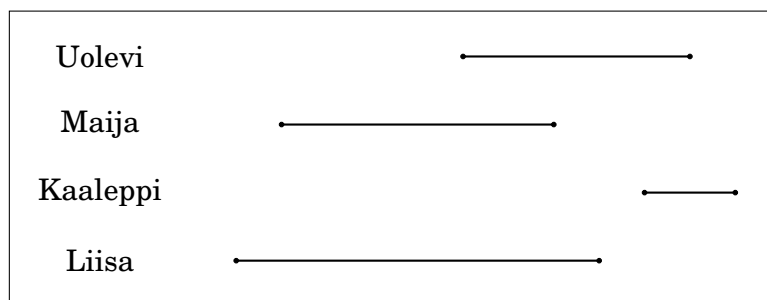
We go through the events from left to right and maintain a counter. Always when a person arrives, we increase the value of the counter by one, and when a person leaves, we decrease the value of the counter by one. The answer to the problem is the maximum value of the counter during the algorithm.

In the example, the events are processed as follows:
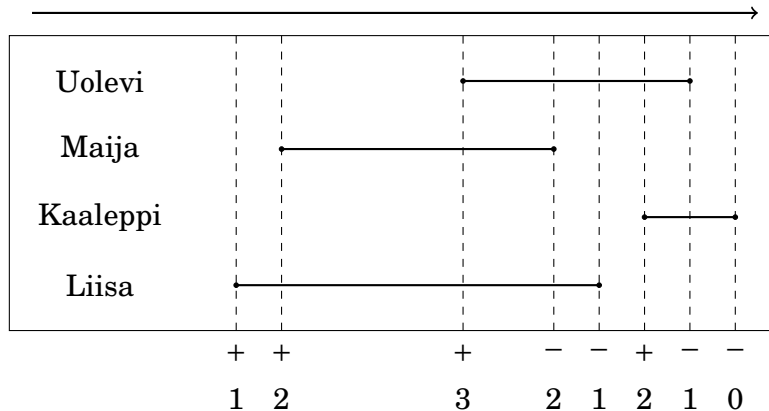
The symbols + and − indicate whether the value of the counter increases or decreases, and the value of the counter is shown below. The maximum value of the counter is 3 between Uolevi's arrival time and Maija's leaving time.

The running time of the algorithm is $O(n \log n)$, because sorting the events takes $O(n \log n)$ time and the rest of the algorithm takes $O(n)$ time.

30.1 Intersection points

Given a set of n line segments, each of them being either horizontal or vertical, consider the problem of counting the total number of intersection points. For example, when the line segments are
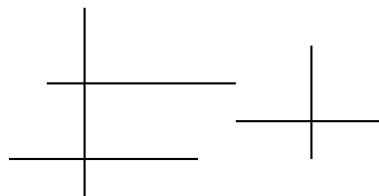
there are three intersection points:
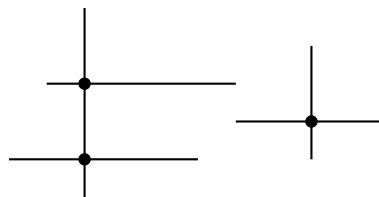
It is easy to solve the problem in $O(n^2)$ time, because we can go through all possible pairs of segments and check if they intersect. However, we can solve the problem more efficiently in $O(n \log n)$ time using a sweep line algorithm.

The idea is to generate three types of events:

  (1) horizontal segment begins
  (2) horizontal segment ends
  (3) vertical segment

The following events correspond to the example:

266
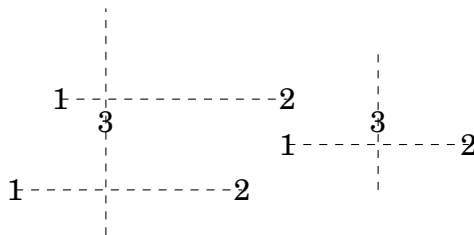
We go through the events from left to right and use a data structure that maintains a set of y coordinates where there is an active horizontal segment. At event 1, we add the y coordinate of the segment to the set, and at event 2, we remove the y coordinate from the set.

Intersection points are calculated at event 3. When there is a vertical segment between points y_1 and y_2, we count the number of active horizontal segments whose y coordinate is between y_1 and y_2, and add this number to the total number of intersection points.

An appropriate data structure for storing y coordinates of horizontal segments is either a binary indexed tree or a segment tree, possibly with index compression. Using such structures, processing each event takes $O(\log n)$ time, so the total running time of the algorithm is $O(n \log n)$.

30.2   Closest pair problem

Given a set of n points, our next problem is to find two points whose distance is minimum. For example, if the points are
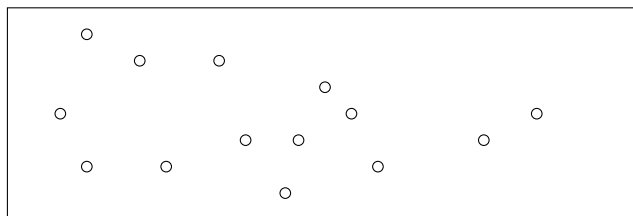
we should find the following points:
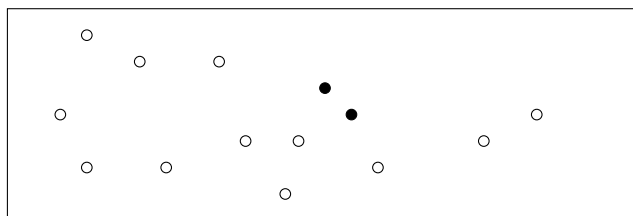
This is another example of a problem that can be solved in $O(n \log n)$ time using a sweep line algorithm. We go through the points from left to right and maintain a value d: the minimum distance between two points seen so far. At each point, we find the nearest point to the left. If the distance is less than d, it is the new minimum distance and we update the value of d.

If the current point is (x, y) and there is a point to the left within a distance of less than d, the x coordinate of such a point must be between $[x - d, x]$ and the y

coordinate must be between $[y-d, y+d]$. Thus, it suffices to only consider points that are located in those ranges, which makes the algorithm efficient.

For example, in the following picture the region marked with dashed lines contains the points that can be within a distance of d from the active point:
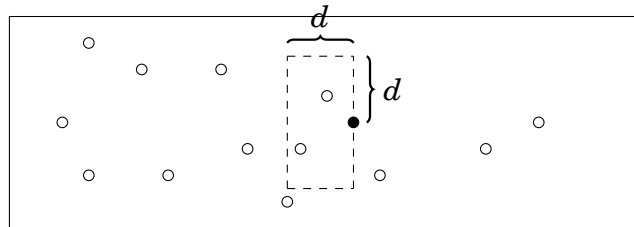
The efficiency of the algorithm is based on the fact that such a region always contains only $O(1)$ points. We can go through those points in $O(\log n)$ time by maintaining a set of points whose x coordinate is between $[x-d, x]$, in increasing order according to their y coordinates.

The time complexity of the algorithm is $O(n \log n)$, because we go through n points and find for each point the nearest point to the left in $O(\log n)$ time.

30.3 Convex hull problem

A **convex hull** is the smallest convex polygon that contains all points of a given set. Convexity means that a line segment between any two vertices of the polygon is completely inside the polygon.

For example, for the points
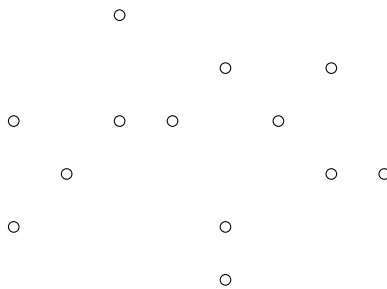
the convex hull is as follows:
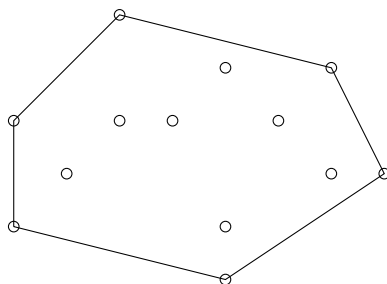
Andrew's algorithm [3] provides an easy way to construct the convex hull for a set of points in $O(n \log n)$ time. The algorithm constructs the convex hull in

two steps: first the upper hull and then the lower hull. Both steps are similar, so we can focus on constructing the upper hull.

We sort the points primarily according to x coordinates and secondarily according to y coordinates. After this, we go through the points and always add the new point to the hull. After adding a point we check using cross products whether the tree last point in the hull turn left. If this holds, we remove the middle point from the hull. After this we keep checking the three last points and removing points, until the three last points do not turn left.

The following pictures show how Andrew's algorithm works:

269
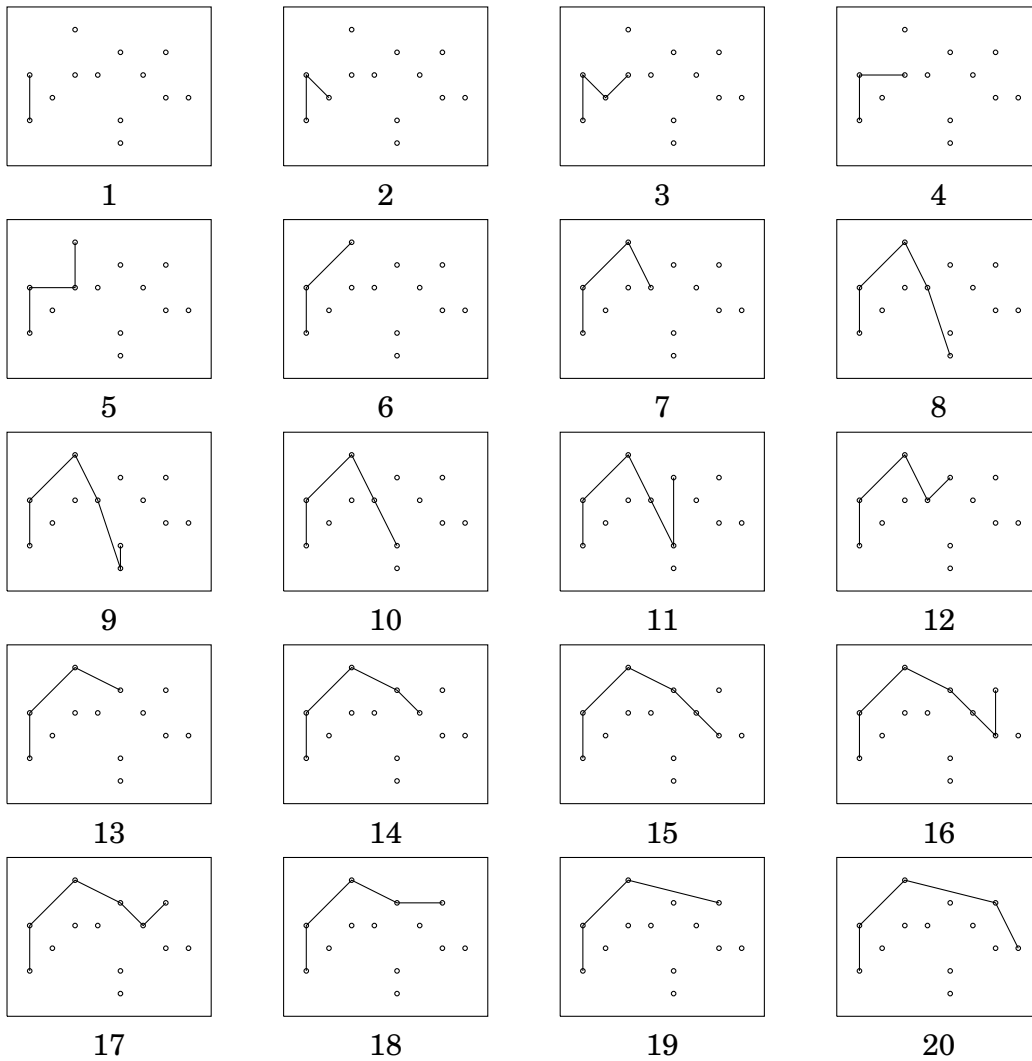
Bibliography

[1] A. V. Aho, J. E. Hopcroft and J. Ullman. *Data Structures and Algorithms*, Addison-Wesley, 1983.

[2] R. K. Ahuja and J. B. Orlin. Distance directed augmenting path algorithms for maximum flow and parametric maximum flow problems. *Naval Research Logistics*, 38(3):413–430, 1991.

[3] A. M. Andrew. Another efficient algorithm for convex hulls in two dimensions. *Information Processing Letters*, 9(5):216–219, 1979.

[4] B. Aspvall, M. F. Plass and R. E. Tarjan. A linear-time algorithm for testing the truth of certain quantified boolean formulas. *Information Processing Letters*, 8(3):121–123, 1979.

[5] R. Bellman. On a routing problem. *Quarterly of Applied Mathematics*, 16(1):87–90, 1958.

[6] M. A. Bender and M. Farach-Colton. The LCA problem revisited. In *Latin American Symposium on Theoretical Informatics*, 88–94, 2000.

[7] J. Bentley. *Programming Pearls*. Addison-Wesley, 1986.

[8] Codeforces: On "Mo's algorithm", `http://codeforces.com/blog/entry/20032`

[9] E. W. Dijkstra. A note on two problems in connexion with graphs. *Numerische Mathematik*, 1(1):269–271, 1959.

[10] J. Edmonds. Paths, trees, and flowers. *Canadian Journal of Mathematics*, 17(3):449–467, 1965.

[11] J. Edmonds and R. M. Karp. Theoretical improvements in algorithmic efficiency for network flow problems. *Journal of the ACM*, 19(2):248–264, 1972.

[12] D. Fanding. A faster algorithm for shortest-path – SPFA. *Journal of Southwest Jiaotong University*, 2, 1994.

[13] P. M. Fenwick. A new data structure for cumulative frequency tables. *Software: Practice and Experience*, 24(3):327–336, 1994.

[14] J. Fischer and V. Heun. Theoretical and practical improvements on the RMQ-problem, with applications to LCA and LCE. In *Annual Symposium on Combinatorial Pattern Matching*, 36–48, 2006.

[15] J. Fischer and V. Heun. Space-efficient preprocessing schemes for range minimum queries on static arrays. *SIAM Journal on Computing*, 40(2):465–492, 2011.

[16] R. W. Floyd Algorithm 97: shortest path. *Communications of the ACM*, 5(6):345, 1962.

[17] L. R. Ford and D. R. Fulkerson. Maximal flow through a network. *Canadian Journal of Mathematics*, 8(3):399–404, 1956.

[18] F. Le Gall. Powers of tensors and fast matrix multiplication. In *Proceedings of the 39th International Symposium on Symbolic and Algebraic Computation*, 296–303.

[19] M. R. Garey and D. S. Johnson. *Computers and Intractability: A Guide to the Theory of NP-Completeness*, W. H. Freeman and Company, 1979.

[20] Google Code Jam Statistics (2016), `https://www.go-hero.net/jam/16`

[21] A. Grønlund and S. Pettie. Threesomes, degenerates, and love triangles. *2014 IEEE 55th Annual Symposium on Foundations of Computer Science*, 621–630, 2014.

[22] D. Gusfield. *Algorithms on Strings, Trees and Sequences: Computer Science and Computational Biology*, Cambridge University Press, 1997.

[23] A method for the construction of minimum-redundancy codes. *Proceedings of the IRE*, 40(9):1098–1101, 1952.

[24] The ACM-ICPC International Collegiate Programming Contest, `https://icpc.baylor.edu/`

[25] International Olympiad in Informatics – Official site, `http://www.ioinformatics.org/`

[26] International Olympiad in Informatics – Statistics, `http://stats.ioinformatics.org/`

[27] MisoF's IOI Syllabus page, `https://people.ksp.sk/~misof/ioi-syllabus/`

[28] R. M. Karp and M. O. Rabin. Efficient randomized pattern-matching algorithms. *IBM Journal of Research and Development*, 31(2):249–260, 1987.

[29] P. W. Kasteleyn. The statistics of dimers on a lattice: I. The number of dimer arrangements on a quadratic lattice. *Physica*, 27(12):1209–1225, 1961.

[30] C. Kent, G. m. Landau and M. Ziv-Ukelson. On the complexity of sparse exon assembly. *Journal of Computational Biology*, 13(5):1013–1027, 2006.

[31] J. B. Kruskal. On the shortest spanning subtree of a graph and the traveling salesman problem. *Proceedings of the American Mathematical Society*, 7(1):48–50, 1956.

[32] M. G. Main and R. J. Lorentz. An $O(n \log n)$ algorithm for finding all repetitions in a string. *Journal of Algorithms*, 5(3):422–432, 1984.

[33] Młodzieżowa Akademia Informatyczna (MAIN), `http://main.edu.pl/en`

[34] J. Pachocki and J. Radoszweski. Where to use and how not to use polynomial string hashing. *Olympiads in Informatics*, 2013.

[35] R. C. Prim. Shortest connection networks and some generalizations. *Bell System Technical Journal*, 36(6):1389–1401, 1957.

[36] M. Sharir. A strong-connectivity algorithm and its applications in data flow analysis. *Computers & Mathematics with Applications*, 7(1):67–72, 1981.

[37] V. Strassen. Gaussian elimination is not optimal. *Numerische Mathematik*, 13(4):354–356, 1969.

[38] H. N. V. Temperley and M. E. Fisher. Dimer problem in statistical mechanics – an exact result. *Philosophical Magazine*, 6(68):1061–1063, 1961.

Index