Competitive Programmer's Handbook

Antti Laaksonen

January 10, 2017

Contents

Preface vii

I Basic techniques 1

1 Introduction 3
- 1.1 Programming languages 3
- 1.2 Input and output 4
- 1.3 Handling numbers 6
- 1.4 Shortening code 8
- 1.5 Mathematics 9

2 Time complexity 15
- 2.1 Calculation rules 15
- 2.2 Complexity classes 18
- 2.3 Estimating efficiency 19
- 2.4 Maximum subarray sum 19

3 Sorting 23
- 3.1 Sorting theory 23
- 3.2 Sorting in C++ 27
- 3.3 Binary search 29

4 Data structures 33
- 4.1 Dynamic array 33
- 4.2 Set structure 35
- 4.3 Map structure 36
- 4.4 Iterators and ranges 37
- 4.5 Other structures 39
- 4.6 Comparison to sorting 42

5 Complete search 45
- 5.1 Generating subsets 45
- 5.2 Generating permutations 47
- 5.3 Backtracking 48
- 5.4 Pruning the search 49
- 5.5 Meet in the middle 52

6 Greedy algorithms **55**
6.1 Coin problem 55
6.2 Scheduling 56
6.3 Tasks and deadlines 58
6.4 Minimizing sums 59
6.5 Data compression 60

7 Dynamic programming **63**
7.1 Coin problem 63
7.2 Longest increasing subsequence 68
7.3 Path in a grid 69
7.4 Knapsack 70
7.5 Edit distance 71
7.6 Counting tilings 73

8 Amortized analysis **75**
8.1 Two pointers method 75
8.2 Nearest smaller elements 77
8.3 Sliding window minimum 79

9 Range queries **81**
9.1 Static array queries 81
9.2 Binary indexed tree 84
9.3 Segment tree 86
9.4 Additional techniques 91

10 Bit manipulation **93**
10.1 Bit representation 93
10.2 Bit operations 94
10.3 Bit representation of sets 96
10.4 Dynamic programming 98

II Graph algorithms **101**

11 Basics of graphs **103**
11.1 Terminology 103
11.2 Graph representation 106

12 Graph search **111**
12.1 Depth-first search 111
12.2 Breadth-first search 113
12.3 Applications 115

13 Shortest paths **117**
13.1 Bellman–Ford algorithm 117
13.2 Dijkstra's algorithm 120
13.3 Floyd–Warshall algorithm 123

14 Tree algorithms **127**
14.1 Tree search . . . 128
14.2 Diameter . . . 129
14.3 Distances between nodes . . . 130
14.4 Binary trees . . . 131

15 Spanning trees **133**
15.1 Kruskal's algorithm . . . 134
15.2 Union-find structure . . . 137
15.3 Prim's algorithm . . . 139

16 Directed graphs **141**
16.1 Topological sorting . . . 141
16.2 Dynamic programming . . . 143
16.3 Successor paths . . . 146
16.4 Cycle detection . . . 147

17 Strongly connectivity **149**
17.1 Kosaraju's algorithm . . . 150
17.2 2SAT problem . . . 152

18 Tree queries **155**
18.1 Finding ancestors . . . 155
18.2 Subtrees and paths . . . 156
18.3 Lowest common ancestor . . . 159

19 Paths and circuits **163**
19.1 Eulerian path . . . 163
19.2 Hamiltonian path . . . 167
19.3 De Bruijn sequence . . . 168
19.4 Knight's tour . . . 169

20 Flows and cuts **171**
20.1 Ford–Fulkerson algorithm . . . 172
20.2 Parallel paths . . . 176
20.3 Maximum matching . . . 177
20.4 Path covers . . . 180

III Advanced topics **183**

Preface

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

The book is especially intended for high school students who want to learn algorithms and possibly participate in the International Olympiad in Informatics (IOI). The book is also suitable for university students and 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 learn a great deal about algorithms if you spend time reading the book and solving exercises.

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

Part I

Basic techniques

Chapter 1

Introduction

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

The **design of algorithms** consists of problem solving and mathematical thinking. Skills for analyzing problems and solving them using creativity is 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 for 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 to be correct as well.

Good coding style in contests is straightforward and concise. The solutions should be written quickly, because there is not much time available. Unlike in traditional software engineering, the solutions 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[1]. 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 the benefits of them. For example, if big integers are needed in the problem, Python

[1] `https://www.go-hero.net/jam/16`

can be a good choice because it contains a built-in library for handling big integers. Still, usually the goal is to write the problems so that the use of a specific programming language is not an unfair advantage in the contest.

All examples in this book are written in C++, and the data structures and algorithms in the standard library are often used. The book follows the C++11 standard, that can be used in most contests nowadays. If you can't 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:

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

using namespace std;

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

The `#include` line at the beginning of the code is a feature in the g++ compiler that allows to include the whole 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 is enough 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 obeys 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:

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

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

```
123 456 apina
```

```
123   456
apina
```

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

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

Handling 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`, becauses `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 using the `getline` function:

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

If the amount of data is unknown, the following loop can be handy:

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

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 code reads the input from the file "input.txt" and writes the output to the file "output.txt".

1.3 Handling numbers

Integers

The most popular integer type in competitive programming is `int`. This is a 32-bit type with value range $-2^{31} \ldots 2^{31}-1$, i.e., 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 $-2^{63} \ldots 2^{63}-1$, i.e., 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 typical 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, the problems are written so that the type `long long` is enough. Still, it is good to know that the `g++` compiler also features an 128-bit type `__int128_t` with value range $-2^{127} \ldots 2^{127}-1$, i.e., $-10^{38} \ldots 10^{38}$. 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 for a problem is a very big integer but it is enough to print 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 big, it is enough 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 calculated 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 calculate 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 answer should be always given so that the remainder is between $0\ldots m-1$. However, in C++ and other languages, the remainder of a negative number can be negative. An easy way to make sure that this will not happen 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. The easiest way is to use the `printf` function that can be given the number of decimal places. 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, 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
```

Because of a rounding error, the value of x is a bit less 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 due to rounding errors. 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 having absolute value at most 2^{53}.

1.4 Shortening code

Short code is ideal in competitive programming, because the algorithm should be implemented 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;
```

It is also possible to define a macro with parameters which makes it possible to shorten loops and other structures in the code. 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++) {
    haku(i);
}
```

can be shortened as follows:

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

1.5 Mathematics

Mathematics plays an important role in competitive programming, and it is not possible to become a successful competitive programmer without good skills in mathematics. This section covers 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 sum** is a sum where the difference between any two consecutive numbers is constant. For example,

$$3 + 7 + 11 + 15$$

is an arithmetic sum with constant 4. An arithmetic sum 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 sum** is a sum where the ratio between any two consecutive numbers is constant. For example,

$$3 + 6 + 12 + 24$$

is a geometric sum with constant 2. A geometric sum 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 geometric sum 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 the harmonic sum is $\log_2(n)+1$. The reason for this is that we can change each term $1/k$ so that k becomes a power of two that doesn't 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 sum 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 set S, i.e., the number of elements in the set. For example, in the above set, $|X| = 3$.

If set S contains 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 as follows 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 $P = \{1,2,\ldots,10\}$, 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 $A \subset S$. Set S always has $2^{|S|}$ subsets, including the empty set. For example, the subsets of the set $\{2,4,7\}$ are

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

Often used sets are

- $\mathbb{N}$ (natural numbers),
- $\mathbb{Z}$ (integers),
- $\mathbb{Q}$ (rational numbers) and
- $\mathbb{R}$ (real numbers).

The set $\mathbb{N}$ of natural numbers can be defined in two ways, depending on the situation: either $\mathbb{N} = \{0,1,2,\ldots\}$ or $\mathbb{N} = \{1,2,3,...\}$.

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 $f(n)$ 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 the 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 negation $\neg A$ reverses the value of an expression. 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 elements in 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,

$$\forall x((x > 2 \wedge \neg P(x)) \Rightarrow (\exists a(\exists b(x = ab \wedge a > 1 \wedge b > 1))))$$

means that if a number x is larger than 2 and not a prime number, 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 $\max(x_1, x_2, \ldots, x_n)$ return the smallest and the 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** $n!$ is defined

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

or recursively

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

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

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

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:

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

Logarithm

The **logarithm** of a number x is denoted $\log_k(x)$ where k is the base of the logarithm. The logarithm is defined so that $\log_k(x) = a$ exactly when $k^a = x$.

A useful interpretation in algorithmics 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:

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

Logarithms are often needed in the analysis of algorithms because many efficient algorithms divide in half something at each step. Thus, we can estimate the efficiency of those algorithms using the logarithm.

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

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 an 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 estimate if 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

The typical 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
}
```

Correspondingly, 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 doesn't tell the exact number of times the code inside a loop is executed, but it only tells 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 and the other phases are not important.

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 variables. In this case, the formula for the time complexity 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 the function branches into two parts. Thus, the call g(n) causes the following calls:

call	amount
g(n)	1
g($n-1$)	2
$\cdots$	$\cdots$
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

Typical complexity classes are:

$O(1)$ The running time of a **constant-time** algorithm doesn't 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 reason for this is that the logarithm $\log_2 n$ equals the number of times n must be divided by 2 to produce 1.

$O(\sqrt{n})$ The running time of this kind of algorithm is between $O(\log n)$ and $O(n)$. A special feature of the square root is that $\sqrt{n} = n/\sqrt{n}$, so the square root lies "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 means 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 the time complexity of each operation is $O(\log n)$.

$O(n^2)$ A **quadratic** algorithm often contains two nested loops. It is possible to go through all pairs of 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 input elements in $O(n^3)$ time.

$O(2^n)$ This time complexity often means 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 means that the algorithm iterates trough 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.

2.3 Estimating efficiency

By calculating the time complexity, it is possible to check before the implementation that an algorithm is efficient enough for the problem. The starting point for the estimation 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 some tens of seconds time, 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 desired 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)	desired 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 should be $O(n)$ or $O(n \log n)$. This information makes it easier to design an algorithm because it rules out approaches that would yield an algorithm with a slower time complexity.

Still, it is important to remember that a time complexity doesn't tell everything about the efficiency because it hides the **constant factors**. For example, an algorithm that runs in $O(n)$ time can 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 with different time complexities. 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**, i.e., the largest possible sum of numbers in a contiguous region in the array. The problem is interesting because 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

the following subarray produces the maximum sum 10:

1	2	3	4	5	6	7	8
−1	2	4	−3	5	2	−5	2

Solution 1

A straightforward solution for 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 array x with indices $1\ldots n$. Variables a and b select the first and last number in the subarray, and the sum of the subarray is calculated to variable s. 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.

Solution 2

It is easy to make the first solution more efficient by removing one loop. This is possible by calculating the sum at the same time when the right border 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 $O(n^2)$.

Solution 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 index the maximum subarray sum that ends to that index. After this, the answer for the problem is the maximum of those sums.

Condider the subproblem of finding the maximum subarray for a fixed ending index k. There are two possibilities:

1. The subarray only contains the element at index k.
2. The subarray consists of a subarray that ends to index $k-1$, followed by the element at index k.

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

The following code implements the solution:

```
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 access all array elements at least once.

Efficiency comparison

It is interesting to study how efficient the 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	solution 1	solution 2	solution 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 solution 1 becomes slower when $n = 10^3$, and the $O(n^2)$ time solution 2 becomes slower when $n = 10^4$. Only the $O(n)$ time solution 3 solves even the largest inputs instantly.

Chapter 3

Sorting

Sorting is a fundamental algorithm design problem. In addition, 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 question ”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 question ”what is the most frequent element in the array?” can be solved similarly.

There are many algorithms for sorting, that are also good examples of 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 algorithm

for sorting is **bubble sort** where the elements "bubble" forward in the array according to their values.

Bubble sort consists of $n-1$ rounds. On each round, the algorithm iterates through the elements in the array. Whenever two successive elements are found that are not in correct order, the algorithm swaps them. The algorithm can be implemented as follows for array t[1], t[2], ..., 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 is in the correct place, after the second round the second largest element is in the correct place, etc. Thus, after $n-1$ rounds, all elements 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 successive elements in the array. It turns out that the time complexity of this kind of 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**. It is a pair of elements $(\texttt{t}[a],\texttt{t}[b])$ in the array such that $a < b$ and $\texttt{t}[a] > \texttt{t}[b]$, i.e., they are in 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 indicates how sorted the array is. An array is completely sorted when there are no inversions. On the other hand, if the array elements are in reverse order, the number of inversions is maximum:

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

Swapping successive elements that are in wrong order removes exactly one inversion from the array. Thus, if a sorting algorithm can only swap successive 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 an algorithm that is not limited to swapping successive elements. One such algorithm is **mergesort** that sorts an array recursively by dividing it into smaller subarrays.

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

1. If $a = b$, don't do anything because the subarray is already sorted.
2. Calculate the index of the middle element: $k = \lfloor (a+b)/2 \rfloor$.
3. Recursively sort the subarray $[a,k]$.
4. Recursively sort the subarray $[k+1,b]$.
5. *Merge* the sorted subarrays $[a,k]$ and $[k+1,b]$ into a sorted subarray $[a,b]$.

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 $[a,k]$ and $[k+1,b]$ 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

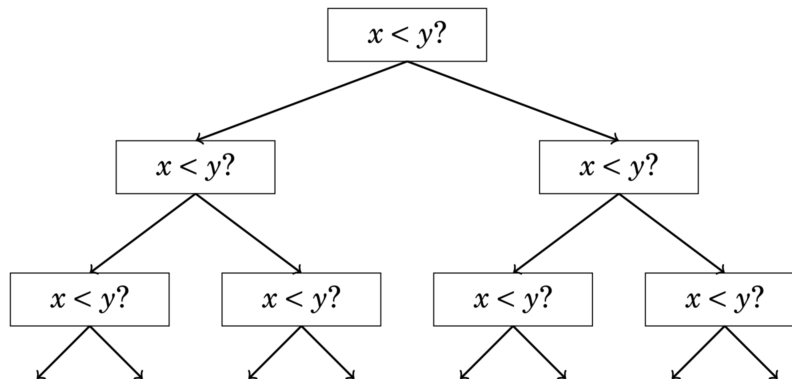
Finally, the algorithm merges the sorted subarrays and creates the final sorted array:

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 examining the 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 order 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 an sorting algorithm in the worst case is at least $n \log n$.

Counting sort

The lower bound $n \log n$ doesn't 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 $0 \dots c$ 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

produces 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 of element 3 in the bookkeeping array is 2, because the element 3 appears two times in the original array (indices 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 amount 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 an own implementation of a 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 home-made sorting function would be better.

In this section we will see how to use the C++ `sort` function. The following code sorts the numbers in vector `t` 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 in increasing, but a reverse order is possible as follows:

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

A regular 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 operator

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 according to alphabetical order.

Pairs (`pair`) are sorted primarily by the first element (`first`). However, if the first elements of two pairs are equal, they are sorted by the second element (`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 $(2,3)$.

Correspondingly, 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 function

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 all elements in the array. For example, the following code searches for an element x in 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, we have to check all elements in the array. If the array can 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 us. 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 desired element is found, or it turns out that there is no such element.

First, the search checks the middle element in the array. If the middle element is the desired element, the search terminates. Otherwise, the search recursively continues to the left half or to the 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 $a \dots b$ that corresponds to the active region in the array. Initially, the range is $1 \dots n$, 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 a more 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 desired element.

The search goes through the array from the left to the 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 desired element has been found or we know that it doesn't exist 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
```

Variable k is the position in the array, and variable b is the jump length. If the array contains the element x, the index of the element will be in 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 array search, because we can use the standard library instead. 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 a 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 ok(x) that returns `true` if x is a valid solution and `false` otherwise. In addition, we know that 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	$k+1$	$\cdots$
ok(x)	false	false	$\cdots$	false	true	true	$\cdots$

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 ok(x) is `false`. Thus, the next value $k = x+1$ is the smallest possible value for which ok(k) is `true`. The initial jump length z has to be large enough, for example some value for which we know beforehand that 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 becomes $O(n \log z)$.

Finding the maximum value

Binary search can also be used for finding 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 $x >= k$.

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 regular binary search, here it is not allowed that successive 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 a suitable 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 array

A **dynamic array** is an array whose size can be changed during the execution of the code. The most popular dynamic array in C++ is the **vector** structure (`vector`), that can be used almost like a regular 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 a regular 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 a regular array. If the size of the vector increases and the array becomes too small, a new array is allocated and all the elements are copied to the new array. However, this doesn't happen often and the time complexity of push_back is $O(1)$ on average.

Also the **string** structure (string) is 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 index k and has length x. The function find(t) finds the position where a substring t appears in the string.

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 structure

A **set** is a data structure that contains 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 how to use it. The function insert adds an element to the set, the function count returns how many times an element appears in the set, 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 a set is 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 in the set. 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 copies of an element. For example, in the following code all copies of the number 5 are added to the set:

```
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 structure

A **map** is a generalized array that consists of key-value-pairs. While the keys in a regular array are always the successive 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 don't have to be successive values.

C++ contains two map implementations that correspond to the set implementations: the structure map is based on a balanced binary tree and accessing an element takes $O(\log n)$ time, while the structure unordered_map uses a hash map and accessing an element 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 a value of a key is requested but the map doesn't 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 determines if a key exists in the map:

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

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

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

4.4 Iterators and ranges

Many functions in the C++ standard library are given iterators to data structures, and iterators often correspond to ranges. 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*.

Handling ranges

Iterators are used in C++ standard library functions that work with ranges of data structures. 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 a regular 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 when accessing elements in 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 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 `find`(x) returns an iterator that points to an element whose value is x. However, if the set doesn't 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 in the set whose value is at least x. Correspondingly, the function `upper_bound`(x) returns an iterator to the smallest element in the set 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 doesn't 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 << "joukko on tyhjä\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 is true, the element nearest to x is either the element that corresponds to a or the previous element.

4.5 Other structures

Bitset

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

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
```

Pakka

A **deque** (deque) is a dynamic array whose size can be changed at both ends of the array. Like a vector, a deque contains functions push_back and pop_back, but it also contains additional 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 the implementation 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.

Pino

A **stack** (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
```

Queue

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

A **priority queue** (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 element or the 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 shows an example:

```
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();
```

The following definition creates a priority queue that supports finding and removing the minimum element:

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

4.6 Comparison to sorting

Often it's 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 for the problem is to go through all pairs of numbers in $O(n^2)$ time, but next we will concentrate on more efficient solutions.

Solution 1

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

Solution 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 data structure that the algorithm uses. The time complexity of the new algorithm is $O(n)$.

Solution 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 in the lists are random integers between $1 \dots 10^9$:

n	solution 1	solution 2	solution 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

Solutions 1 and 2 are equal except that solution 1 uses the `set` structure and solution 2 uses the `unordered_set` structure. In this case, this choice has a big effect on the running time becase solution 2 is 4–5 times faster than solution 1.

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

Chapter 5

Complete search

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

Complete search is a good technique if it is feasible 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, greedy algorithms or dynamic programming, presented in the next chapters, may be used.

5.1 Generating subsets

We first consider the case where the possible solutions for the problem are the subsets of a set of n elements. In this case, a complete search algorithm has to generate all 2^n subsets of the set.

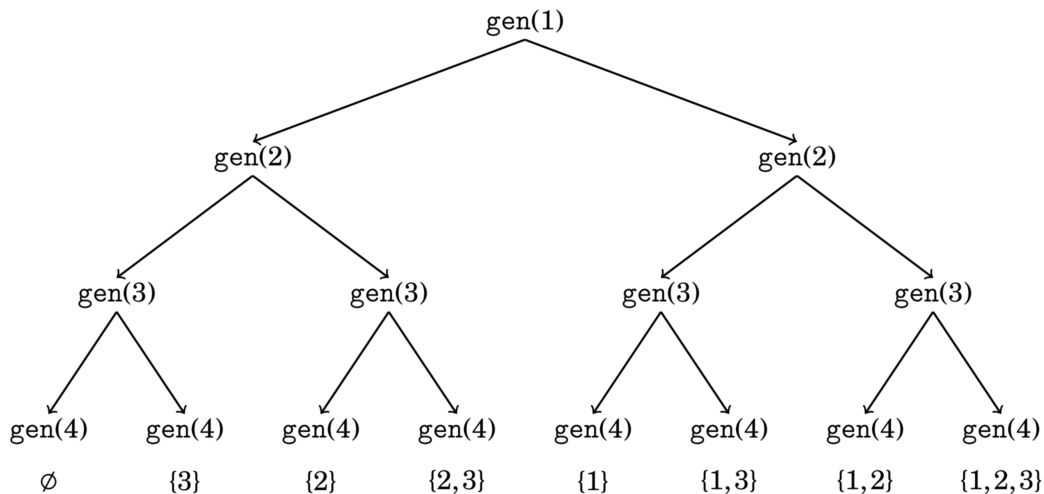
Method 1

An elegant way to go through all subsets of a set is to use recursion. The following function gen generates the subsets of the set $\{1,2,\ldots,n\}$. The function maintains a vector v that will contain the elements in the subset. The generation of the subsets 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 number that is the next candidate to be included in the subset. The function branches to two cases: either k is included or it is not included in the subset. Finally, when $k = n+1$, a decision has been made for all the numbers and one subset has been generated.

For example, when $n = 3$, the function calls create a tree illustrated below. At each call, the left branch doesn't include the number and the right branch includes the number in the subset.



Method 2

Another way to generate the 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 representation indicate which elements of the set are included in the subset.

The usual interpretation is that element k is included in the subset if kth bit from the end of the bit sequence is one. For example, the bit representation of 25 is 11001 that corresponds to the subset $\{1,4,5\}$.

The following iterates through all subsets of a set of n elements

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

The following code converts each bit representation to a vector v that contains the elements in the subset. This can be done by checking which bits are one in the bit representation.

```
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

Another common situation is that the solutions for the problem are permutations of a set of n elements. In this case, a complete search algorithm has to generate $n!$ possible permutations.

Method 1

Like subsets, permutations can be generated using recursion. The following function gen iterates through the permutations of the set $\{1,2,\ldots,n\}$. The function uses the vector v for storing the permutations, and the generation begins by calling the function without parameters.

```
void haku() {
    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);
            haku();
            p[i] = 0;
            v.pop_back();
        }
    }
}
```

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

Method 2

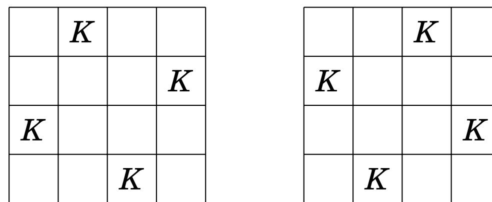
Another method is to begin from permutation $\{1,2,\ldots,n\}$ and at each step generate the next permutation in increasing order. The C++ standard library contains the function `next_permutation` that can be used for this. The following code generates the permutations of the set $\{1,2,\ldots,n\}$ using the function:

```
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

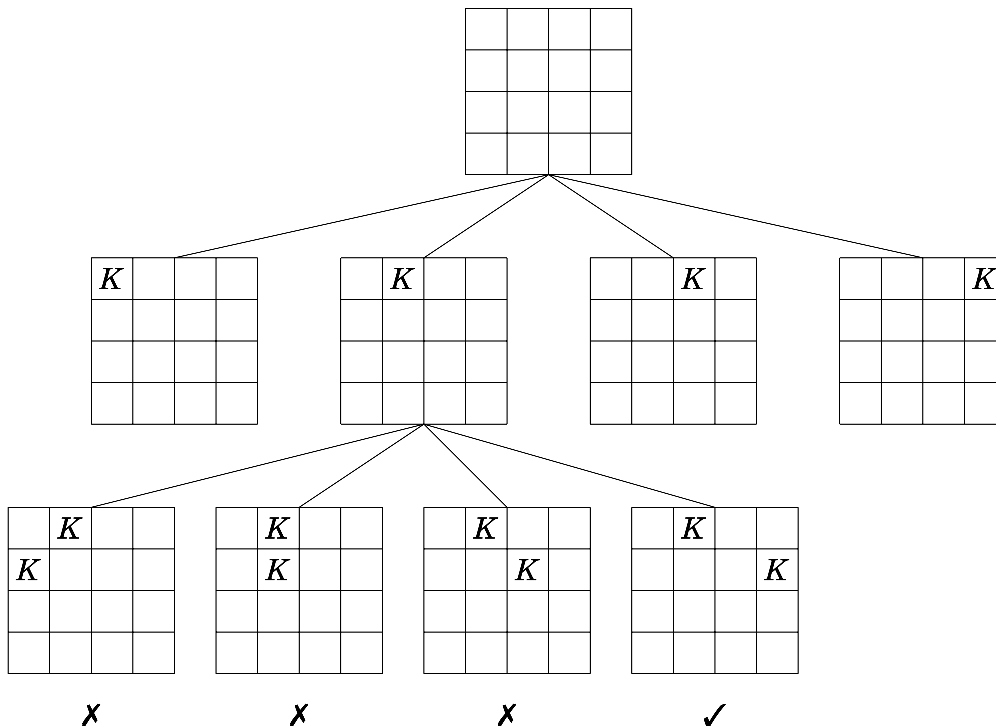
A **backtracking** algorithm begins from an empty solution and extends the solution step by step. At each step, the search branches to all possible directions how the solution can be extended. After processing one branch, the search continues to other possible directions.

As an example, consider the **queen problem** where our 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 for the problem:



The problem can be solved using backtracking by placing queens to the board row by row. More precisely, we should place exactly one queen to each row so that no queen attacks any of the queens placed before. A solution is ready when we have placed all n queens to the board.

For example, when $n = 4$, the tree produced by the backtracking algorithm begins like this:



At the bottom level, the three first subsolutions are not valid because the queens attack each other. However, the fourth subsolution is valid and it can be extended to a full 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 when $0 \le y < n$. Finally, if $y = n$, one 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. Similarly, the arrays `r2` and `r3` keep track of the diagonals. It is not allowed to add another queen to a column or to a diagonal. 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

`r1`

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

`r2`

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

`r3`

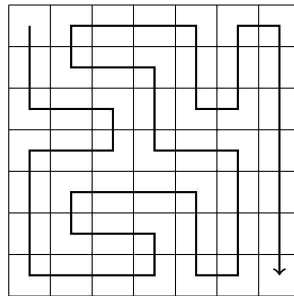
Using the presented backtracking algorithm, we can calculate that, for example, there are 92 ways to place 8 queens to an 8×8 chessboard. When n increases, the search quickly becomes slow because the number of the solutions increases exponentially. For example, calculating the ways to place 16 queens to the 16×16 chessboard already takes about a minute (there are 14772512 solutions).

5.4 Pruning the search

A backtracking algorithm can often be optimized by pruning the search tree. The idea is to add ”intelligence” to the algorithm so that it will notice as soon as possible if is not possible to extend a subsolution into a full solution. This kind of optimization can have a tremendous effect on the efficiency of the search.

Let us consider a problem where our task is to calculate 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 the 7×7 grid, there are

111712 possible paths from the lower-right corner to the upper-right corner. One of the paths is as follows:



We will concentrate on the 7×7 case because it is computationally suitable difficult. We begin with a straightforward backtracking algorithm, and then optimize it step by step using observations how the search tree 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.

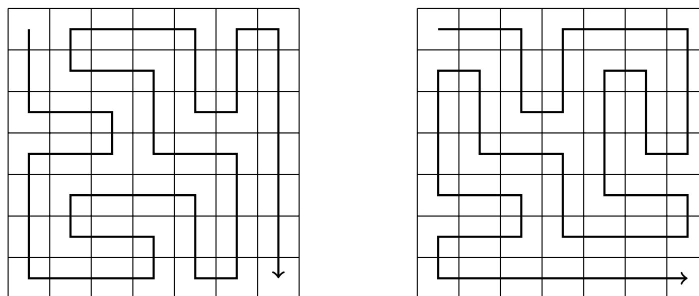
Basic algorithm

The first version of the algorithm doesn't contain any optimizations. We simply use backtracking to generate all possible paths from the upper-left corner to the lower-right corner.

- running time: 483 seconds
- recursive calls: 76 billions

Optimization 1

The first step in a solution is either downward or to the 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:

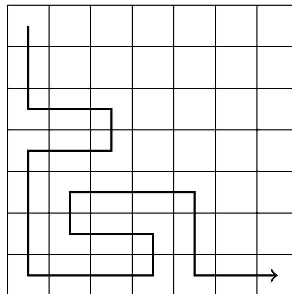


Thus, we can decide that the first step in the solution is always downward, and finally multiply the number of the solutions by two.

- 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 case:

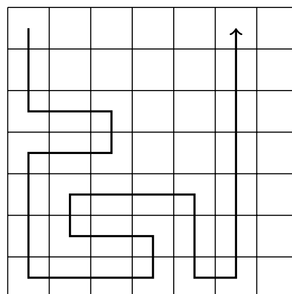


Using this observation, we can terminate the search branch immediately if we reach the lower-right square too early.

- running time: 119 seconds
- recursive calls: 20 billions

Optimization 3

If the path touches the wall so that there is an unvisited square at both sides, the grid splits into two parts. For example, in the following case both the left and the right squares are unvisited:



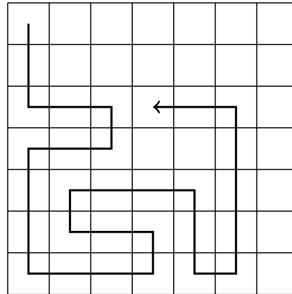
Now it will not be possible to visit every square, so we can terminate the search branch. This optimization is very useful:

- 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 case the top and bottom neighbors are unvisited, so the path cannot visit all squares in the grid anymore:



The search becomes even faster when we terminate the search branch in all such cases:

- running time: 0.6 seconds
- recursive calls: 69 millions

Now it's a good moment to stop optimization and remember our starting point. 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 after the optimizations.

This is a usual phenomenon in backtracking because the search tree is usually large and even simple optimizations can prune a lot of branches in the tree. Especially useful are optimizations that occur at the top of the search tree because they can prune the search very efficiently.

5.5 Meet in the middle

Meet in the middle is a technique where the search space is divided into two equally large parts. A separate search is performed for each of the parts, and finally the results of the searches are combined.

The meet in the middle technique can be used if there is an efficient way to combine the results of the searches. In this case, 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 an integer x. Our task is to find out if it is possible to choose some numbers from the list so that the sum of the numbers 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 the list remains the same but $x = 10$, 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 time complexity of this solution is $O(2^n)$ because there are 2^n possible subsets. However, using the meet in the middle technique, we can create 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 given as input to two lists A and B that each contain about half of the numbers. The first search generates all subsets of the

numbers in the list A and stores their sums to list S_A. Correspondingly, the second search creates the list S_B from the list B. After this, it suffices to check if it is possible to choose one number from S_A and another number 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, assume 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 the lists $S_A = [0,2,4,6]$ and $S_B = [0,5,9,14]$. 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. This choice 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 created using the numbers in S_A and S_B.

Chapter 6

Greedy algorithms

A **greedy algorithm** constructs a solution for a 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 a greedy algorithm is to invent a greedy strategy that always produces an optimal solution for the problem. The locally optimal choices in a greedy algorithm should also be globally optimal. It's often difficult to argue why a greedy algorithm works.

6.1 Coin problem

As the 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 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 natural greedy algorithm for 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 argued as follows:

Each coin 1, 5, 10, 50 and 100 appears at most once in the 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, both coins 2 and 20 can appear at most twice in the optimal solution because, we could replace coins $2+2+2$ by coins $5+1$ and coins $20+20+20$ by coins $50+10$. Moreover, the optimal solution can't contain coins $2+2+1$ or $20+20+10$ because we would 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 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 $5+20+20+5+2+2 = 99$. Thus, the greedy algorithm that always selects the largest coin produces the optimal solution.

This example shows that it can be difficult to argue why 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 *not* necessarily produces an optimal solution.

We can prove that a greedy algorithm doesn't work by showing a counterexample where the algorithm gives a wrong answer. In this problem it's easy to find a counterexample: if the coins are $\{1,3,4\}$ and the sum of money is 6, the greedy algorithm produces the solution $4+1+1$, while the optimal solution is $3+3$.

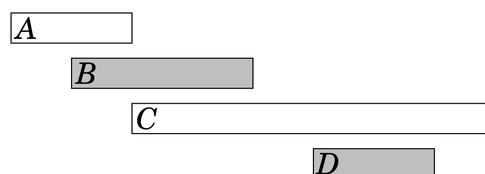
We don't know if the general coin problem can be solved using any greedy algorithm. However, we will revisit the problem in the next chapter because the general problem can be solved using a dynamic programming algorithm that always gives the correct answer.

6.2 Scheduling

Many scheduling problems can be solved using a greedy strategy. A classic problem is as follows: Given n events with their starting and ending times, our task is to plan a schedule so that we can join as many events as possible. It's not possible to join 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

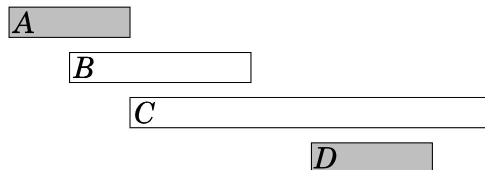
In this case the maximum number of events is two. For example, we can join events B and D as follows:



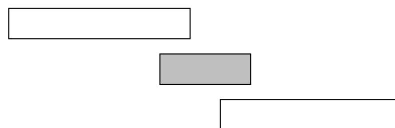
It is possible to invent several greedy algorithms for the problem, but which of them works in every case?

Algorithm 1

The first idea is to select as *short* events as possible. In the example case this algorithm selects the following events:



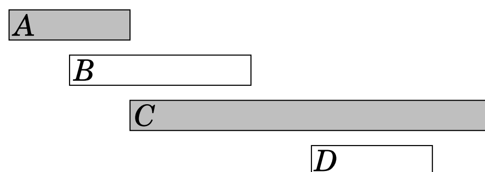
However, choosing short events is not always a correct strategy but the algorithm fails, for example, in the following case:



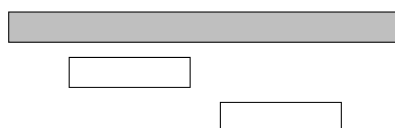
If we select the short event, we can only select one event. However, it would be possible to select both the long events.

Algorithm 2

Another idea is to always select the next possible event that *begins* as *early* as possible. This algorithm selects the following events:



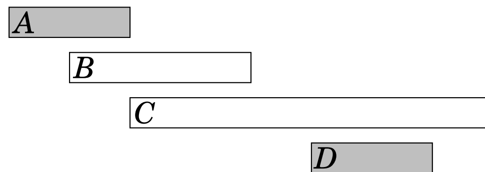
However, we can find a counterexample for this algorithm, too. For example, in the following case, the algorithm selects only one event:



If we select the first event, it is not possible to select any other events. However, it would be possible to join 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. The algorithm works because regarding the final solution, it is optimal to select an event that ends as soon as possible. Then it is optimal to select the next event using the same strategy, etc.

One way to justify the choice is to think what happens if we first select some event that ends later than the event that ends as soon as possible. This can never be a better choice because after an event that ends later, we will have at most an equal number of possibilities to select for the next events, compared to the strategy that we select the event that ends as soon as possible.

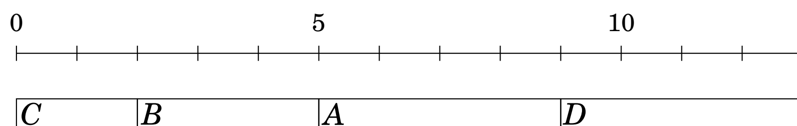
6.3 Tasks and deadlines

We are given n tasks with duration and deadline. Our task is to choose an order to perform the tasks. For each task, we get $d - x$ points where d is the deadline of the task and x is the moment when we finished the task. What is the largest possible total score we can obtain?

For example, if the tasks are

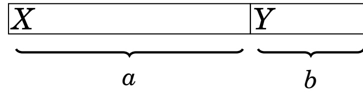
task	duration	deadline
A	4	2
B	3	5
C	2	7
D	4	5

then the optimal solution is to perform the tasks as follows:

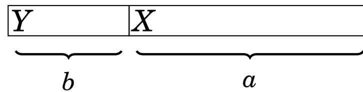


In this solution, C yields 5 points, B yields 0 points, A yields -7 points and D yields -8 points, so the total score is -10.

Surprisingly, the optimal solution for the problem doesn't depend on the dedalines 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 successive tasks such that the first task takes longer than the second task, we can obtain a better solution if we swap the tasks. For example, if the successive tasks are



and $a > b$, the swapped order of the tasks



gives b points less to X and a points more to Y, so the total score increases by $a - b > 0$. In an optimal solution, for each two successive 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 such that the sum

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

becomes as small as possible. 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 the 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 Thus, we should move x as near the median as possible, so the optimal solution 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 doesn't 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

We are given a string, and our task is to *compress* it so that it requires less space. We will do this using a **binary code** that determines for each character a **codeword** that consists of bits. After this, we can compress the string by replacing each character by the corresponding codeword. For example, the following binary code determines 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, the compressed form of the string `AABACDACA` is

000001001011001000,

so 18 bits are needed. However, we can compress the string better by using 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 the **optimal** code for the aforementioned string is as follows:

character	codeword
`A`	0
`B`	110
`C`	10
`D`	111

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

001100101110100,

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

Note that it is required 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 also 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:

merkki	koodisana
A	10
B	11
C	1011
D	111

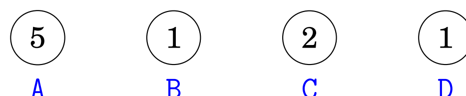
Using this code, it would not be possible to know if the compressed string 1011 means the string AB or the string C.

Huffman coding

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

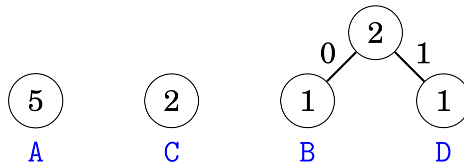
Initially, each character of the string is represented by a node whose weight is the number of times the character appears in the string. Then at each step two nodes with minimum weights are selected and they are combined by creating a new node whose weight is the sum of the weights of the original nodes.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

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:

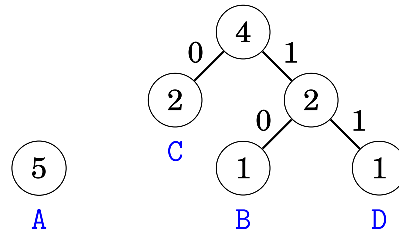


The node that represents character A has weight 5 because character A appears 5 times in the string. The other weights have been calculated in the same way.

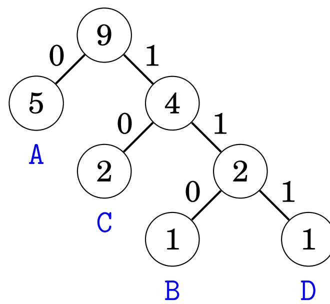
The first step is to combine the nodes that correspond to characters B and D, both with weight 1. The result is:



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 used if the problem can be divided into subproblems that can be calculated independently.

There are two uses for dynamic programming:

- **Findind an optimal solution**: We want to find a solution that is as large as possible or as small as possible.
- **Couting the number of solutions**: We want to calculate the total number of possible solutions.

We will first see how dynamic programming can be used for finding 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 for different problems. This chapter introduces a set of classic problems that are a good starting point.

7.1 Coin problem

We first consider a problem that we have already seen: 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 6.1, we solved the problem using a greedy algorithm that always selects the largest possible coin for the sum. The greedy algorithm works, for example, when the coins are the euro coins, but in the general case the greedy algorithm doesn't necessarily produce an optimal solution.

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

Recursive formulation

The idea in dynamic programming is to formulate the problem recursively so that the answer for the problem can be calculated from the answers for the smaller subproblems. In this case, a natural 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 for 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 values are $\{1,3,4\}$, the first values of the function are as follows:

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

First, $f(0) = 0$ because no coins are needed for 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 the value $f(x)$ can be calculated recursively from the 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 returns 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's a good idea to define

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

This means that an infinite number of coins is needed to create a negative sum of money. This prevents the situation that the recursive function would form a solution where the initial sum of money is negative.

Now it's possible to implement the function in C++ directly using the recursive definition:

```
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 $\texttt{c}[1], \texttt{c}[2], \ldots, \texttt{c}[k]$, and the value 10^9 means 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 becomes efficient by using memoization.

Memoization

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

In this problem, we can use the array

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

where $\texttt{d}[x]$ will contain the value $f(x)$. The constant N should be chosen so that there is space for all needed values of the function.

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

```
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 and stored to $\texttt{d}[x]$. If $f(x)$ can be found in the array, the function directly returns it. Otherwise the function calculates the value recursively and stores it to $\texttt{d}[x]$.

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

The time complexity of the resulting algorithm is $O(xk)$ when the sum is x and the number of coins is k. In practice, the algorithm is usable 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 implement dynamic programming solutions using loops. Still, the underlying idea is the same as in the recursive function.

Constructing the solution

Sometimes it is not enough to find out the value of the optimal solution, but we should also give an example how such a solution can be constructed. In this 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 variation of the problem that it's 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 required 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 sum together all possible alternatives to construct a solution.

In this case, we can define a function $f(x)$ that returns 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 function $f(x)$ can be recursively calculated 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 should first choose some coin c_i and after this 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's not possible to form a negative sum of money.

In the above example the function becomes

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

and the first values of the function are:

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

The following code calculates the value $f(x)$ 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 the 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 will be done in modulo m. In this case, it is enough to add the line

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

after the line

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

Now we have covered 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 how dynamic programming can be used.

7.2 Longest increasing subsequence

Given an array that contains n numbers $x_1, x_2, \ldots, x_n$, our task is find the **longest increasing subsequence** in the array. This is a sequence of array elements that goes from the left to the 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 to index k. Thus, the answer for the problem is the largest of values $f(1), f(2), \ldots, f(n)$.

For example, in the above array the values for the function are as follows:

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

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

1. The subsequence only contains the element x_k, so $f(k) = 1$.
2. We choose some index i for which $i < k$ and $x_i < x_k$. We extend the longest increasing subsequence that ends to index i by adding the element x_k to it. In this case $f(k) = f(i)+1$.

Consider calculating the value $f(7)$. The best solution is to extend the longest increasing subsequence that ends to index 5, i.e., the sequence $[2,5,7]$, by adding the element $x_7 = 8$. The result is $[2,5,7,8]$, and $f(7) = f(5)+1 = 4$.

A straightforward way to calculate the value $f(k)$ is to go through all indices $i = 1,2,\ldots,k-1$ that can contain the previous element in the subsequence. 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, but this is more difficult.

7.3 Path 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. 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. In addition, it is only allowed to move downwards and to the right.

In the followig grid, the best path is marked with gray background:

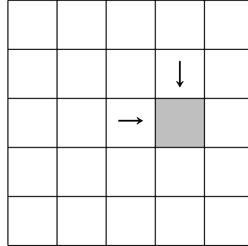
3	7	9	2	7
9	8	3	5	5
1	7	9	8	5
3	8	6	4	10
6	3	9	7	8

The sum of numbers is $3+9+8+7+9+8+5+10+8 = 67$ that is the largest possible sum in a path from the upper-left corner to the lower-right corner.

A good approach for the problem is to calculate for each square (y,x) the largest possible sum in a path from the upper-left corner to the square (y,x). We

denote this sum $f(y,x)$, so $f(n,n)$ is the largest sum in 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 to square(y,x) can either come from square $(y,x-1)$ or from 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:

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

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 $p_1, p_2, \dots, p_n$ and values $a_1, a_2, \dots, a_n$. 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 total weight is 12, the optimal solution is to select objects B and D. Their total weight $6+5=11$ doesn't exceed 12, and their total value $3+3=6$ is as large as possible.

This task is possible to solve 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 for 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 $u \neq 0$. 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 for 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 $u \neq 0$. 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 features of the input affect on 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**, also known as the **Levenshtein distance**, indicates how similar two strings are. It is the minimum number of editing operations needed for transforming the first string into the second 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 operation `LOVE` → `MOVE` (change) and then operation `MOVE` → `MOVIE` (insertion). This is the smallest possible number of operations because it is clear that one operation is not enough.

Suppose we are given strings `x` of n characters and `y` of m characters, and we want to calculate the edit distance between them. This can be efficiently 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` is $f(n,m)$, and the function also determines the editing operations needed.

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 covers all ways to shorten the strings:

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

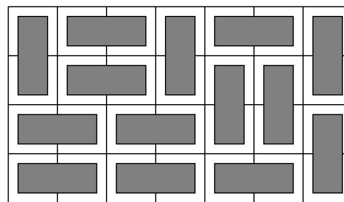
The lower-right corner of the table indicates that the edit distance between `LOVE` and `MOVIE` is 2. The table also shows how to construct the shortest sequence of editing operations. In this case the path is as follows:

		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 dynamic programming state is more complex than a fixed combination of numbers. As an example, we consider a problem where our task is to calculate the number of different 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 $\{\sqcap, \sqcup, \sqsubset, \sqsupset\}$. For example, the above solution consists of four rows that correspond to the following strings:

- $\sqcap\sqsubset\sqsupset\sqcap\sqsubset\sqsupset\sqcap$
- $\sqcup\sqsubset\sqsupset\sqcup\sqcap\sqcap\sqcup$
- $\sqsubset\sqsupset\sqsubset\sqsupset\sqcup\sqcup\sqcap$
- $\sqsubset\sqsupset\sqsubset\sqsupset\sqsubset\sqsupset\sqcup$

Let $f(k,x)$ denote the number of ways to construct a solution for the rows $1 \dots k$ 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 be the state of the previous row.

A solution is valid if row 1 doesn't contain the character $\sqcup$, row n doesn't contain the character $\sqcap$, and all successive rows are *compatible*. For example, the rows $\sqcup\sqsubset\sqsupset\sqcup\sqcap\sqcap\sqcup$ and $\sqsubset\sqsupset\sqsubset\sqsupset\sqcup\sqcup\sqcap$ are compatible, while the rows $\sqcap\sqsubset\sqsupset\sqcap\sqsubset\sqsupset\sqcap$ and $\sqsubset\sqsupset\sqsubset\sqsupset\sqsubset\sqsupset\sqcup$ are not compatible.

Since a row consists of m characters and there are four choices for each character, the number of different rows is at most 4^m. Thus, the time complexity of the solution is $O(n4^{2m})$ because we can check 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's 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 as strings. It turns out that it is sufficient to know the columns

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

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

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

This formula is very efficient because it calculates the number of tilings on $O(nm)$ 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.

Chapter 8

Amortized analysis

Often the time complexity of an algorithm is easy to analyze by looking at the structure of the algorithm: what loops there are and how many times they are performed. However, sometimes a straightforward analysis doesn't give a true picture of the efficiency of the algorithm.

Amortized analysis can be used for analyzing an algorithm that contains an operation whose time complexity varies. The idea is to consider all such operations during the execution of the algorithm instead of a single operation, and estimate the total time complexity of the operations.

8.1 Two pointers method

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

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

Subarray sum

Given an array that contains n positive integers, our task is to find out if there is a subarray where the sum of the elements is x. For example, the array

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

contains a subarray with sum 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 using the two pointers method. The idea is to iterate through the array using two pointers that define a

range in the array. On each turn, the left pointer moves one step forward, and the right pointer moves forward as long as the sum is at most x. If the sum of the range becomes exactly x, we have found a solution.

As an example, we 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

First, the pointers define a range with sum $1+3+2=6$. The range can't be larger because the next number 5 would make the sum larger than x.

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

After this, the left pointer moves one step forward. The right pointer doesn't 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 the right pointer move $O(n)$ steps during the algorithm, the time complexity is $O(n)$.

Sum of two numbers

Given an array of n integers and an integer x, our task is to find two numbers in array whose sum is x or report that there are no such numbers. This problem is known as the **2SUM** problem, and it can be solved efficiently using the two pointers method.

First, we sort the numbers in the array in increasing order. After this, we iterate through the array using two pointers that begin at both ends of the array. The left pointer begins from the first element and moves one step forward on each

turn. The right pointer begins from the last element and always moves backward until the sum of the range defined by the pointers is at most x. If the sum is exactly x, we have found a solution.

For example, consider the following array when our task is to find two elements whose sum is $x = 12$:

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 doesn't move, and the solution $5 + 7 = 12$ has been found.

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

At the beginning of the algorithm, the sorting 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 in another way in $O(n \log n)$ time using binary search. In this 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 somewhat more difficult problem is the **3SUM** problem where our task is to find *three* numbers whose sum is x. This problem can be solved in $O(n^2)$ time. Can you see how it is possible?

8.2 Nearest smaller elements

Amortized analysis is often used for estimating the number of operations performed for a data structure. The operations may be distributed unevenly so that

the most operations appear during a certain phase in the algorithm, but the total number of the operations is limited.

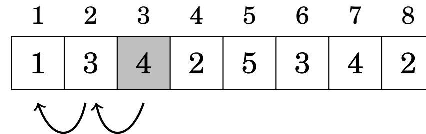
As an example, let us consider a problem where our task is to find for each element in an array the **nearest smaller element**, i.e., the nearest smaller element that precedes the element in the array. It is possible that no such element exists, and the algorithm should notice this. It turns out that the problem can be efficiently solved in $O(n)$ time using a suitable data structure.

An efficient solution for the problem is to iterate through the array from the left to the right, and maintain a chain of elements where the first element is the active element in the array and each following element is the nearest smaller element of the previous element. If the chain only contains one element, the active element doesn't have a nearest smaller element. At each step, we remove elements from the chain until the first element is smaller than the active element, or the chain is empty. After this, the active element becomes the first element in the chain.

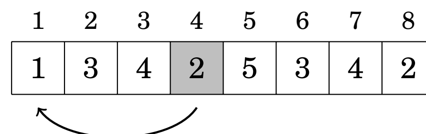
As an example, consider the following array:

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

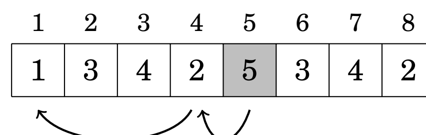
First, numbers 1, 3 and 4 are added to the chain because each element is larger than the previous element. This means that the nearest smaller element of number 4 is number 3 whose nearest smaller element is number 1.



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



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



Algorithm continues in a similar way and finds out 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 in the chain, it

will only be inserted to the beginning of 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. Thus, each element causes $O(1)$ operations to the chain, and the total time complexity of the algorithm is $O(n)$.

8.3 Sliding window minimum

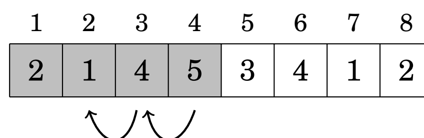
A **sliding window** is an active subarray that moves through the array whose size is constant. At each position of the window, we typically want to calculate some information about the elements inside the window. An interesting problem is to maintain the **sliding window minimum**. This means that at each position of the window, we should report the smallest element inside the window.

The sliding window minima can be calculated using the same idea that we used for calculating the nearest smaller elements. The idea is to maintain a chain whose first element is the last element in the window, and each element 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 all elements from the chain that are larger than the new element. After this, we add the new number to the chain. In addition, if the last element in the chain doesn't belong to the window anymore, it is removed from the chain.

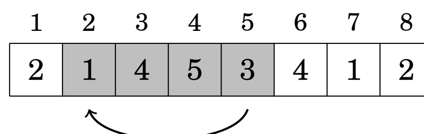
As an example, consider the following array when the window size is $k = 4$:

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

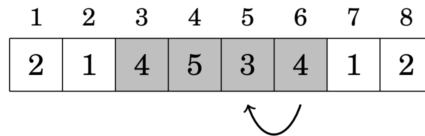
The sliding window begins from the left border of the array. At the first window position, the smallest element is 1:



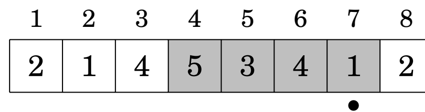
Then the window moves one step forward. The new number 3 is smaller than the numbers 5 and 4 in the chain, so the numbers 5 and 4 are removed and the number 3 is added to the chain. The smallest element is 1 as before.



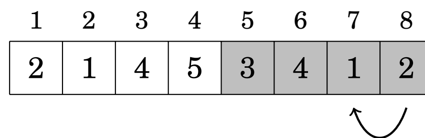
After this, the window moves again and the smallest element 1 doesn't belong to the window anymore. Thus, it is removed from the chain and the smallest element is now 3. In addition, the new number 4 is added to the chain.



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:



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.



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 a **range query**, a range of an array is given and we should calculate some value from the elements in the range. Typical range queries are:

- **sum query**: calculate the sum of elements in range $[a,b]$
- **minimum query**: find the smallest element in range $[a,b]$
- **maximum query**: find the largest element in range $[a,b]$

For example, in range $[4,7]$ of the following array, the sum is $4+6+1+3 = 14$, the minimum is 1 and the maximum is 6:

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

An easy way to answer a range query is to iterate through all the elements in the range. For example, we can answer a sum query as follows:

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

The above function handles a sum query in $O(n)$ time, which is slow if the array is large and there are a lot of queries. In this chapter we will learn how range queries can be answered much more efficiently.

9.1 Static array queries

We will first focus on a simple case where the array is **static**, i.e., the elements never change between the queries. In this case, it suffices to process the contents of the array beforehand and construct a data structure that can be used for answering any possible range query efficiently.

Sum query

Sum queries can be answered efficiently by constructing a **sum array** that contains the sum of the range $[1,k]$ for each $k = 1,2,\ldots,n$. After this, the sum of any range $[a,b]$ of the original array can be calculated in $O(1)$ time using the precalculated sum array.

For example, for the 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

The following code constructs a prefix sum array s from array t in $O(n)$ time:

```
for (int i = 1; i <= n; i++) {
    s[i] = s[i-1]+t[i];
}
```

After this, the following function answers a sum query in $O(1)$ time:

```
int sum(int a, int b) {
    return s[b]-s[a-1];
}
```

The function calculates the sum of range $[a,b]$ by subtracting the sum of range $[1,a-1]$ from the sum of range $[1,b]$. Thus, only two values from the sum array are needed, and the query takes $O(1)$ time. Note that thanks to the one-based indexing, the function also works when $a = 1$ if $s[0] = 0$.

As an example, consider the range $[4,7]$:

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

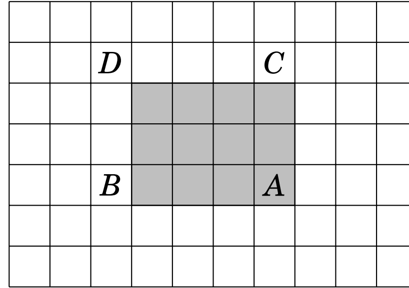
The sum of the range $[4,7]$ is $8+6+1+4 = 19$. This can be calculated from the sum array using the sums $[1,3]$ and $[1,7]$:

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

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

We can also generalize the idea of a sum array for a two-dimensional array. In this case, it will be possible to calculate the sum of any rectangular subarray in $O(1)$ time. The sum array will contain sums for all subarrays that begin from the upper-left corner.

The following picture illustrates the idea:



The sum inside the gray subarray can be calculated using the formula

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

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

Minimum query

It is also possible to answer a minimum query in $O(1)$ time after preprocessing, though it is more difficult than answer a sum query. Note that minimum and maximum queries can always be implemented using same techniques, so it suffices to focus on the minimum query.

The idea is to find the minimum element for each range of size 2^k in the array. For example, in the array

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

the following minima will be calculated:

range	size	min
[1,1]	1	1
[2,2]	1	3
[3,3]	1	4
[4,4]	1	8
[5,5]	1	6
[6,6]	1	1
[7,7]	1	4
[8,8]	1	2

range	size	min
[1,2]	2	1
[2,3]	2	3
[3,4]	2	4
[4,5]	2	6
[5,6]	2	1
[6,7]	2	1
[7,8]	2	2

range	size	min
[1,4]	4	1
[2,5]	4	3
[3,6]	4	1
[4,7]	4	1
[5,8]	4	1
[1,8]	8	1

The number of 2^k ranges in an array is $O(n \log n)$ because there are $O(\log n)$ ranges that begin from each array index. The minima for all 2^k ranges can be calculated in $O(n \log n)$ time because each 2^k range consists of two 2^{k-1} ranges, so the minima can be calculated recursively.

After this, the minimum of any range $[a,b]$c can be calculated in $O(1)$ time as a minimum of two 2^k ranges where $k = \lfloor \log_2(b-a+1) \rfloor$. The first range begins from index a, and the second range ends to index b. The parameter k is so chosen that two 2^k ranges cover the range $[a,b]$ entirely.

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 [2,7] is 6, and $\lfloor \log_2(6) \rfloor = 2$. Thus, the minimum can be calculated from two ranges of length 4. The ranges are [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

The minimum of the range [2,5] is 3, and the minimum of the range [4,7] is 1. Thus, the minimum of the range [2,7] is 1.

9.2 Binary indexed tree

A **binary indexed tree** or a **Fenwick tree** is a data structure that resembles a sum array. The supported operations are answering a sum query for range $[a,b]$, and updating the element at index k. The time complexity for both of the operations is $O(\log n)$.

Unlike a sum array, a binary indexed tree can be efficiently updated between the sum queries. This would not be possible using a sum array because we should build the whole sum array again in $O(n)$ time after each update.

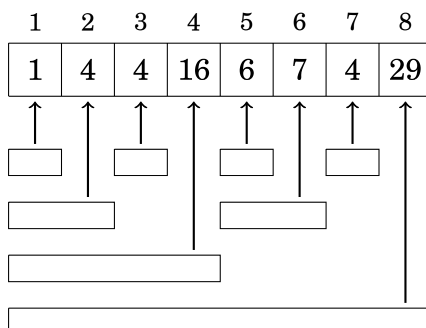
Structure

A binary indexed tree can be represented as an array where index k contains the sum of a range in the original array that ends to index k. The length of the range is the largest power of two that divides k. For example, if $k = 6$, the length of the range is 2 because 2 divides 6 but 4 doesn't divide 6.

For example, for the 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:

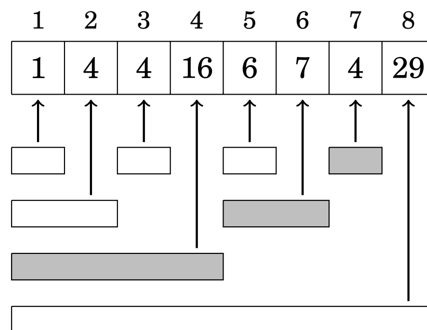


For example, the binary indexed tree contains the value 7 at index 6 because the sum of the elements in the range $[5,6]$ of the original array is $6+1=7$.

Sum query

The basic operation in a binary indexed tree is calculating the sum of a range $[1,k]$ where k is any index in the array. The sum of any range can be constructed by combining sums of subranges in the tree.

For example, the range $[1,7]$ will be divided into three subranges:

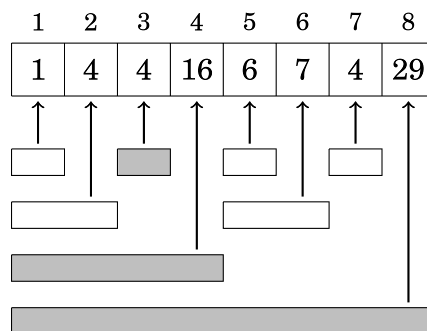


Thus, the sum of the range $[1,7]$ is $16+7+4=27$. Because of the structure of the binary indexed tree, the length of each subrange inside a range is distinct, so the sum of a range always consists of sums of $O(\log n)$ subranges.

Using the same technique that we previously used with a sum array, we can efficiently calculate the sum of any range $[a,b]$ by substracting the sum of the range $[1,a-1]$ from the sum of the range $[1,b]$. The time complexity remains $O(\log n)$ because it suffices to calculate two sums of $[1,k]$ ranges.

Array update

When an element in the original array changes, several sums in the binary indexed tree change. For example, if the value at index 3 changes, the sums of the following ranges change:



Also in this case, the length of each range is distinct, so $O(\log n)$ ranges will be updated in the binary indexed tree.

Implementation

The operations of a binary indexed tree can be implemented in an elegant and efficient way using bit manipulation. The bit operation needed is $k\&-k$ that returns the last bit one from number k. For example, $6\&-6=2$ because the number 6 corresponds to 110 and the number 2 corresponds to 10.

It turns out that when calculating a range sum, the index k in the binary indexed tree should be decreased by $k\&-k$ at every step. Correspondingly, when updating the array, the index k should be increased by $k\&-k$ at every step.

The following functions assume that the binary indexed tree is stored to array b and it consists of indices $1\dots n$.

The function sum calculates the sum of the range $[1,k]$:

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

The function add increases the value of element k by x:

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

The time complexity of both above functions is $O(\log n)$ because the functions change $O(\log n)$ values in the binary indexed tree and each move to the next index takes $O(1)$ time using the bit operation.

9.3 Segment tree

A **segment tree** is a data structure whose supported operations are handling a range query for range $[a,b]$ and updating the element at index k. Using a segment tree, we can implement sum queries, minimum 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.

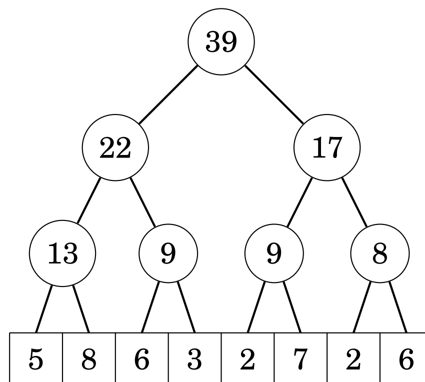
Structure

A segment tree contains $2n-1$ nodes so that the bottom n nodes correspond to the original array and the other nodes contain information needed for range queries. The values in a segment tree depend on the supported query type. We will first assume that the supported query is the sum query.

For example, the array

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

corresponds to the following segment tree:



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.

It is convenient to build a segment tree when the size of the array is a power of two and the tree is a complete binary tree. In the sequel, we will assume that the tree is built like this. If the size of the array is not a power of two, we can always extend it using zero elements.

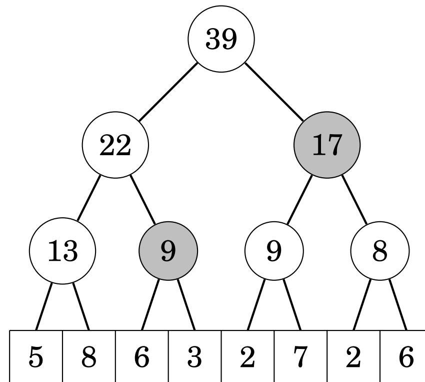
Range query

In a segment tree, the answer for a range query is calculated from nodes that belong to the range and are as high as possible in the tree. Each node gives the answer for a subrange, and the answer for the entire range can be calculated by combining these values.

For example, consider the following range:

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

The sum of elements in the range $[3,8]$ is $6+3+2+7+2+6=26$. The sum can be calculated from the segment tree using the following subranges:



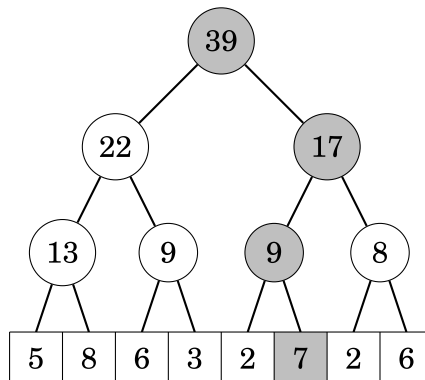
Thus, the sum of the range is $9+17=26$.

When the answer for a range query is calculated using as high nodes as possible, at most two nodes on each level of the segment tree are needed. Because of this, the total number of nodes examined is only $O(\log n)$.

Array update

When an element in the array changes, we should update all nodes in the segment tree whose value depends on the changed element. This can be done by travelling from the bottom to the top in the tree 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.



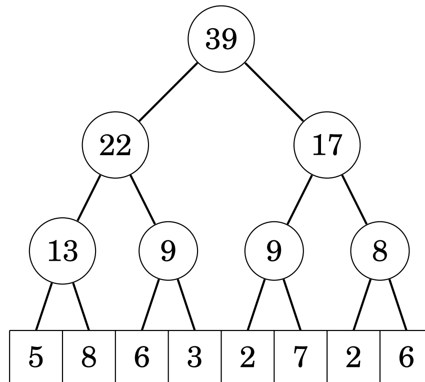
The path from the bottom of the segment tree to the top always consists of $O(\log n)$ nodes, so updating the array affects $O(\log n)$ nodes in the tree.

Storing the tree

A segment tree can be stored as an array of $2N$ elements where N is a power of two. From now on, we will assume that the indices of the original array are between 0 and $N-1$.

The element at index 1 in the segment tree array contains the top node of the tree, the elements at indices 2 and 3 correspond to the second level of the tree, and so on. Finally, the elements beginning from index N contain the bottom level of the tree, i.e., the actual content 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 index k,

- the parent node is at index $\lfloor k/2 \rfloor$,
- the left child node is at index $2k$, and
- the right child node is at index $2k+1$.

Note that this implies that the index of a node is even if it is a left child and odd if it is a right child.

Functions

We assume that the segment tree is stored in the array p. The following function calculates the sum of 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 begins from the bottom of the tree and moves step by step upwards in the tree. The function calculates the range sum to the variable s by combining the sums in the tree nodes. The value of a node is added to the sum if the parent node doesn't belong to the range.

The function add increases the value of element 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];
    }
}
```

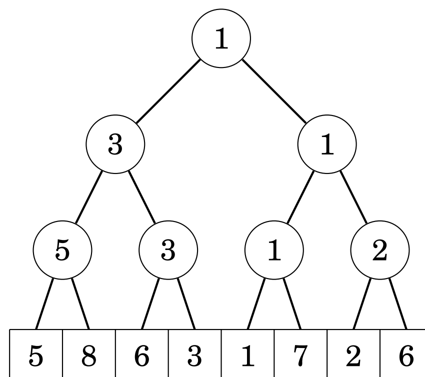
First the function updates the bottom level of the tree that corresponds to the original array. After this, the function updates the values of all internal nodes in the tree, until it reaches the root node of the tree.

Both operations in the segment tree 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 at each step.

Other queries

Besides the sum query, the segment tree can support any range query where the answer for range $[a,b]$ can be efficiently calculated from ranges $[a,c]$ and $[c+1,b]$ where c is some element between a and b. Such queries are, for example, minimum and maximum, greatest common divisor, and bit operations.

For example, the following segment tree supports minimum queries:

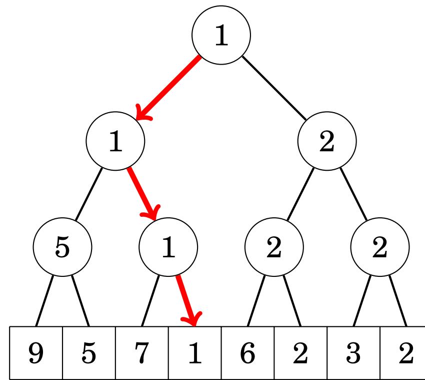


In this segment tree, every node in the tree contains the smallest element in the corresponding range of the original array. The top node of the tree contains the smallest element in the array. The tree can be implemented like previously, but instead of sums, minima are calculated.

Binary search in tree

The structure of the segment tree makes it possible to use binary search. For example, if the tree supports the minimum query, we can find the index of the smallest element in $O(\log n)$ time.

For example, in the following tree the smallest element is 1 that can be found by following a path downwards from the top node:



9.4 Additional techniques

Index compression

A limitation in data structures that have been built upon an array is that the elements are indexed using integers $1,2,3$, etc. Difficulties arise when the indices needed are large. For example, using the index 10^9 would require that the array would contain 10^9 elements which is not realistic.

However, we can often bypass this limitation by using **index compression** where the indices are redistributed so that they are integers $1,2,3$, 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 index $p(x)$ where p is a function that redistributes the indices. We require that the order of the indices doesn't change, so if $a < b$, then $p(a) < p(b)$. Thanks to this, we can conviently perform queries despite the fact that the indices are compressed.

For example, if the original indices are 555, 10^9 and 8, the new indices are:

$$
\begin{array}{lcl}
p(8) & = & 1 \\
p(555) & = & 2 \\
p(10^9) & = & 3
\end{array}
$$

Range update

So far, we have implemented data structures that support range queries and modifications of single values. Let us now consider a reverse situation where we should update ranges and retrieve single values. We focus on an operation that increases all elements in range $[a,b]$ by x.

Surprisingly, we can use the data structures presented in this chapter also in this situation. This requires that we change the array so that each element indicates the *change* with respect to the previous element. For example, the array

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

becomes as follows:

1	2	3	4	5	6	7	8
3	0	−2	0	0	4	−3	0

The original array is the sum array of the new array. Thus, any value in the original array corresponds to a sum of elements in the new array. For example, the value 6 at index 5 in the original array corresponds to the sum $3-2+4=5$.

The benefit in using the new array is that we can update a range by changing just two elements in the new array. For example, if we want to increase the range $2\ldots5$ by 5, it suffices to increase the element at index 2 by 5 and decrease the element at index 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 range $a\ldots b$ by x, we increase the element at index a by x and decrease the element at index $b+1$ by x. The required operations are calculating the sum in a range and updating a value, 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 this is possible as well.

Chapter 10

Bit manipulation

A computer internally manipulates data 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 for manipulating them. It turns out that there are many uses for bit operations in the implementation of algorithms.

10.1 Bit representation

The **bit representation** of a number indicates which powers of two form the number. For example, the bit representation of the number 43 is 101011 because $43 = 2^5 + 2^3 + 2^1 + 2^0$ where bits 0, 1, 3 and 5 from the right are ones, and all other bits are zeros.

The length of a bit representation of a number in a computer is static, and depends on the data type chosen. For example, the `int` type in C++ is usually a 32-bit type, and an `int` number consists of 32 bits. In this case, the bit representation of 43 as an `int` number is as follows:

$$00000000000000000000000000101011$$

The bit representation of a number is either **signed** or **unsigned**. The first bit of a signed number is the sign (+ or −), and we can represent numbers $-2^{n-1} \dots 2^{n-1} - 1$ using n bits. In an unsigned number, in turn, all bits belong to the number and we can represent numbers $0 \dots 2^n - 1$ using n bits.

In an signed bit representation, the first bit of a nonnegative number is 0, and the first bit of a negative number is 1. **Two's complement** is used which means that the opposite number of a number can be calculated by first inversing all the bits in the number, and then increasing the number by one.

For example, the representation of -43 as an `int` number is as follows:

$$11111111111111111111111111010101$$

The connection between signed and unsigned numbers is that the representations of a signed number $-x$ and an unsigned number $2^n - x$ are equal. Thus, the above representation corresponds to the unsigned number $2^{32} - 43$.

In C++, the numbers are signed as default, but we can create unsigned numbers by using the keyword `unsigned`. For example, in the code

```
int x = -43;
unsigned int y = x;
cout << x << "\n"; // -43
cout << y << "\n"; // 4294967253
```

the signed number $x = -43$ becomes the unsigned number $y = 2^{32} - 43$.

If a number becomes too large or too small for the bit representation chosen, it will overflow. In practice, 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, in the code

```
int x = 2147483647
cout << x << "\n"; // 2147483647
x++;
cout << x << "\n"; // -2147483648
```

we increase $2^{31} - 1$ by one to get -2^{31}.

10.2 Bit operations

And operation

The **and** operation x & y produces a number that has bit 1 in positions where both the numbers x and y have bit 1. For example, 22 & 26 = 18 because

	10110	(22)
&	11010	(26)
=	10010	(18)

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.

Or operation

The **or** operation x | y produces a number that has bit 1 in positions where at least one of the numbers x and y have bit 1. For example, 22 | 26 = 30 because

	10110	(22)
		11010	(26)
=	11110	(30)

Xor operation

The **xor** operation x ^ y produces a number that has bit 1 in positions where exactly one of the numbers x and y have bit 1. For example, 22 ^ 26 = 12 because

$$\begin{array}{rrr} & 10110 & (22) \\ \wedge & 11010 & (26) \\ \hline = & 01100 & (12) \end{array}$$

Not operation

The **not** operation $\sim x$ produces a number where all the bits of x have been inversed. 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:

$$\begin{array}{rcrl} x & = & 29 & 00000000000000000000000000011101 \\ \sim x & = & -30 & 11111111111111111111111111100010 \end{array}$$

Bit shifts

The left bit shift $x << k$ produces a number where the bits of x have been moved k steps to the left by adding k zero bits to the number. The right bit shift $x >> k$ produces a number where the bits of x have been moved k steps to the right by removing k last bits from the number.

For example, $14 << 2 = 56$ because 14 equals 1110, and it becomes 56 that equals 111000. Correspondingly, $49 >> 3 = 6$ because 49 equals 110001, and it becomes 6 that equals 110.

Note that the left bit shift $x << k$ corresponds to multiplying x by 2^k, and the right bit shift $x >> k$ corresponds to dividing x by 2^k rounding downwards.

Bit manipulation

The bits in a number are indexed from the right to the left beginning from zero. A number of the form $1 << k$ contains a one bit in position k, and all other bits are zero, so we can manipulate single bits of numbers using these numbers.

The kth bit in x is one if $x \,\&\, (1 << k) = (1 << k)$. The formula $x \mid (1 << k)$ sets the kth bit of x to one, the formula $x \,\&\, \sim(1 << k)$ sets the kth bit of x to zero, and the formula $x \wedge (1 << k)$ inverses 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 $x \mid (x-1)$, in turn, inverses 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 contains the following functions for bit manipulation:

- `__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 following code shows how to use the functions:

```
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 functions support `int` numbers, but there are also `long long` versions of the functions available with the prefix `ll`.

10.3 Bit representation of sets

Each subset of a set $\{0,1,2,\ldots,n-1\}$ corresponds to a n bit number where the one bits indicate which elements are included in the subset. For example, the bit representation for $\{1,3,4,8\}$ is 100011010 that equals $2^8+2^4+2^3+2^1=282$.

The bit representation of a set uses little memory because only one bit is needed for the information whether an element belongs to the set. In addition, we can efficiently manipulate sets that are stored as bits.

Set operations

In the following code, the variable x contains a subset of $\{0,1,2,\ldots,31\}$. The code adds elements 1, 3, 4 and 8 to the set and then prints the elements in the set.

```
// x is an empty set
int x = 0;
// add numbers 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
```

Using the bit representation of a set, we can efficiently implement set operations using bit operations:

- a & b is the intersection $a \cap b$ of a and b (this contains the elements that are in both the sets)
- a | b is the union $a \cup b$ of a and b (this contains the elements that are at least in one of the sets)
- a & (~b) is the difference $a \setminus b$ of a and b (this contains the elements that are in a but not in b)

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 iterates 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 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 iteration over permutations into iteration over subsets. In this case, the dynamic programming state contains a subset of a set and possibly some additional information.

The benefit in this technique is that $n!$, the number of permutations of an n element set, is much larger than 2^n, the number of subsets. For example, if $n = 20$, then $n! = 2432902008176640000$ and $2^n = 1048576$. Thus, for certain values of n, we can go through subsets but not through permutations.

As an example, let's calculate the number of permutations of set $\{0,1,\ldots,n-1\}$ where the difference between any two successive elements is larger than one. For example, there are two solutions for $n = 4$:

- $(1,3,0,2)$
- $(2,0,3,1)$

Let $f(x,k)$ denote the number of permutations for a subset x where the last number is k and the difference between any two successive 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 can't be next to each other.

Using f, the solution for the problem is the sum

$$\sum_{i=0}^{n-1} f(\{0,1,\ldots,n-1\},i).$$

The dynamic programming states can be stored as follows:

```
long long 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;
```

After this, 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];
            }
        }
    }
}
```

The variable b contains the bit representation of the subset, and the corresponding permutation is of the form $(\ldots, j, i)$. It is required that the difference between i and j is larger than 1, and the numbers belong to subset b.

Finally, the number of solutions can be calculated as follows to s:

```
long long s = 0;
for (int i = 0; i < n; i++) {
    s += d[(1<<n)-1][i];
}
```

Sums of subsets

Let's assume that every subset x of $\{0,1,\ldots,n-1\}$ is assigned a value $c(x)$, and our task is to calculate for each subset x the sum

$$s(x) = \sum_{y \subset x} c(y)$$

that corresponds to the sum

$$s(x) = \sum_{y \& x = y} c(y)$$

using bit operations. The following table gives an example of the values of the functions when $n = 3$:

x	$c(x)$	$s(x)$
000	2	2
001	0	2
010	1	3
011	3	6
100	0	2
101	4	6
110	2	5
111	0	12

For example, $s(110) = c(000) + c(010) + c(100) + c(110) = 5$.

The problem can be solved in $O(2^n n)$ time by defining a function $f(x,k)$ that calculates the sum of values $c(y)$ where x can be converted into y by changing any one bits in positions $0,1,\ldots,k$ to zero bits. Using this function, the solution for the problem is $s(x) = f(x, n-1)$.

The base cases for the function are:

$$f(x,0) = \begin{cases} c(x) & \text{if bit 0 in } x \text{ is 0} \\ c(x) + c(x \wedge 1) & \text{if bit 0 in } x \text{ is 1} \end{cases}$$

For larger values of k, the following recursion holds:

$$f(x,k) = \begin{cases} f(x,k-1) & \text{if bit } k \text{ in } x \text{ is 0} \\ f(x,k-1) + f(x \wedge (1 << k), k-1) & \text{if bit } k \text{ in } x \text{ is 1} \end{cases}$$

Thus, we can calculate the values for the function as follows using dynamic programming. The code assumes that the array c contains the values for c, and it constructs an array s that contains the values for s.

```
for (int x = 0; x < (1<<n); x++) {
    f[x][0] = c[x];
    if (x&1) f[x][0] += c[x^1];
}
for (int k = 1; k < n; k++) {
    for (int x = 0; x < (1<<n); x++) {
        f[x][k] = f[x][k-1];
        if (b&(1<<k)) f[x][k] += f[x^(1<<k)][k-1];
    }
    if (k == n-1) s[x] = f[x][k];
}
```

Actually, a much shorter implementation is possible because we can calculate the results directly to array s:

```
for (int x = 0; x < (1<<n); x++) s[x] = c[x];
for (int k = 0; k < n; k++) {
    for (int x = 0; x < (1<<n); x++) {
        if (x&(1<<k)) s[x] += s[x^(1<<k)];
    }
}
```

Part II

Graph algorithms

Chapter 11

Basics of graphs

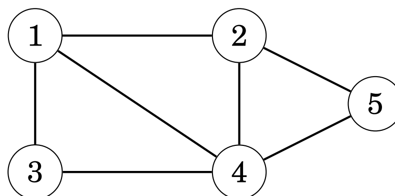
Many programming problems can be solved by interpreting the problem as a graph problem and using a suitable graph algorithm. A typical example of a graph is a network of roads and cities in a country. Sometimes, though, the graph is hidden in the problem and it can be difficult to detect it.

This part of the book discusses techniques and algorithms involving graphs that are important in competitive programming. We will first go through graph terminology and different ways to store graphs in algorithms.

11.1 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. In addition, the nodes are numbered using integers $1,2,\dots,n$.

For example, the following graph contains 5 nodes and 7 edges:

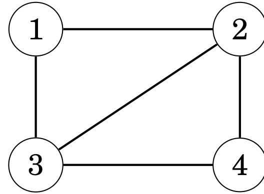


A **path** is a route from node a to node b that goes through the edges in the graph. The **length** of a path is the number of edges in the path. For example, in the above graph, paths from node 1 to node 5 are:

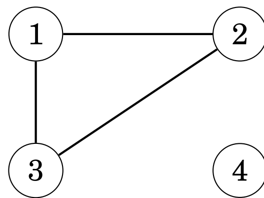
- $1 \to 2 \to 5$ (length 2)
- $1 \to 4 \to 5$ (length 2)
- $1 \to 2 \to 4 \to 5$ (length 3)
- $1 \to 3 \to 4 \to 5$ (length 3)
- $1 \to 4 \to 2 \to 5$ (length 3)
- $1 \to 3 \to 4 \to 2 \to 5$ (length 4)

Connectivity

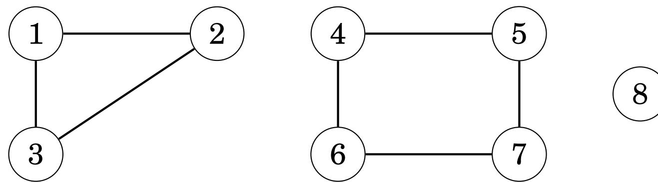
A graph is **connected**, if there is path between any two nodes. For example, the following graph is connected:



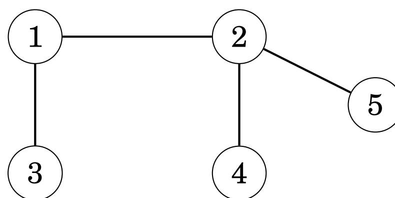
The following graph is not connected because it is not possible to get to other nodes from node 4.



The connected parts of a graph are its **components**. For example, the following graph contains three components: $\{1,2,3\}$, $\{4,5,6,7\}$ and $\{8\}$.

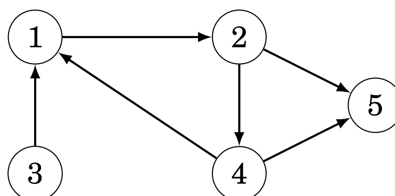


A **tree** is a connected graph that contains n nodes and $n-1$ edges. In a tree, there is a unique path between any two nodes. For example, the following graph is a tree:



Edge directions

A graph is **directed** if the edges can be travelled only in one direction. For example, the following graph is directed:

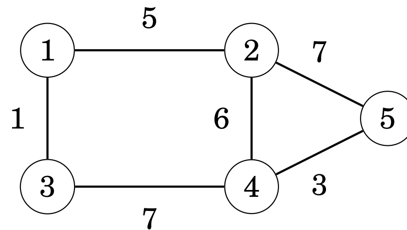


The above graph contains a path from node 3 to 5 using edges $3 \to 1 \to 2 \to 5$. However, the graph doesn't contain a path from node 5 to 3.

A **cycle** is a path whose first and last node is the same. For example, the above graph contains a cycle $1 \to 2 \to 4 \to 1$. If a graph doesn't contain any cycles, it is called **acyclic**.

Edge weights

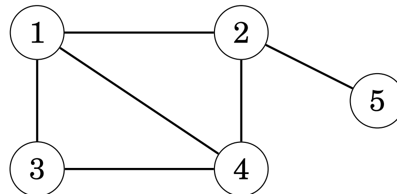
In a **weighted** graph, each edge is assigned a **weight**. Often, the weights are interpreted as edge lengths. For example, the following graph is weighted:



Now the length of a path is the sum of edge weights. For example, in the above graph the length of path $1 \to 2 \to 5$ is 12, and the length of path $1 \to 3 \to 4 \to 5$ is 11. The latter is the shortest path from node 1 to node 5.

Neighbors and degrees

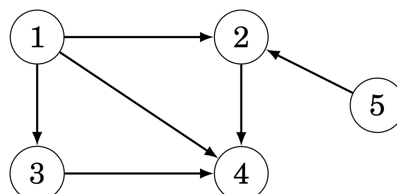
Two nodes are **neighbors** or **adjacent** if there is a edge between them. The **degree** of a node is the number of its neighbors. For example, in the following graph, the neighbors of node 2 are 1, 4 and 5, so its degree is 3.



The sum of degrees in a graph is always $2m$ where m is the number of edges. The reason for this is that each edge increases the degree of two nodes by one. Thus, 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.

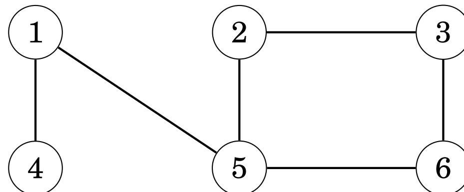
In a directed graph, the **indegree** and **outdegree** of a node is the number of edges that end and begin at the node, respectively. For example, in the following graph, node 2 has indegree 2 and outdegree 1.



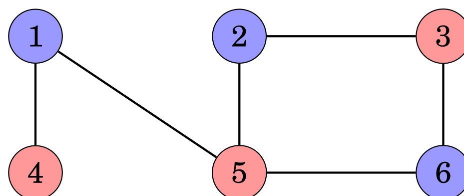
Colorings

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 doesn't contain a cycle with odd number of edges. For example, the graph

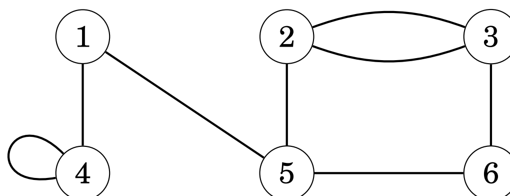


is bipartite because we can color it as follows:



Simplicity

A graph is **simple** if no edge begins and ends at the same node, and there are no multiple edges between two nodes. Often we will assume that the graph is simple. For example, the graph



is *not* simple because there is an edge that begins and ends at node 4, and there are two edges between nodes 2 and 3.

11.2 Graph representation

There are several ways how to represent graphs in memory in an algorithm. The choice of a data structure depends on the size of the graph and how the algorithm manipulates it. Next we will go through three representations.

Adjacency list representation

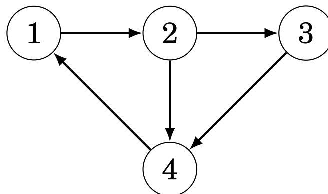
A usual way to represent a graph is to create an **adjacency list** for each node. An adjacency list contains contains all nodes that can be reached from the node

using a single edge. The adjacency list representation is the most popular way to store a graph, and most algorithms can be efficiently implemented using it.

A good way to store the adjacency lists is to allocate an array whose each element is a vector:

```
vector<int> v[N];
```

The adjacency list for node s is in position v[s] in the array. The constant N is so chosen that all adjacency lists can be stored. For example, the graph



can be stored as follows:

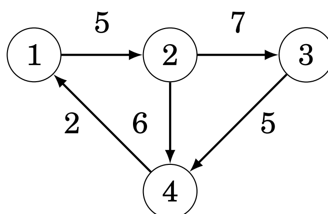
```
v[1].push_back(2);
v[2].push_back(3);
v[2].push_back(4);
v[3].push_back(4);
v[4].push_back(1);
```

If the graph is undirected, it can be stored in a similar way, but each edge each is store in both directions.

For an weighted graph, the structure can be extended as follows:

```
vector<pair<int,int>> v[N];
```

Now each adjacency list contains pairs whose first element is the target node, and the second element is the edge weight. 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 the adjacency list representation is that we can efficiently find the nodes that can be reached from a certain node. For example, the following loop goes trough all nodes that can be reached from node s:

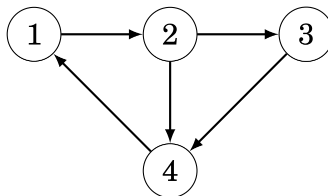
```
for (auto u : v[s]) {
    // process node u
}
```

Adjacency matrix representation

An **adjacency matrix** is a two-dimensional array that indicates for each possible edge if it is included in the graph. Using an adjacency matrix, we can efficiently check if there is an edge between two nodes. On the other hand, the matrix takes a lot of memory if the graph is large. We can store the matrix as an array

```
int v[N][N];
```

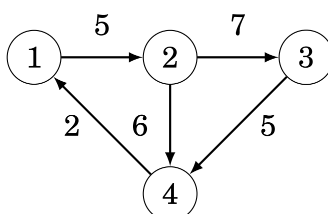
where the value v[a][b] indicates whether the graph contains an edge from node a to node b. If the edge is included in the graph, then v[a][b] $= 1$, and otherwise v[a][b] $= 0$. For example, the graph



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

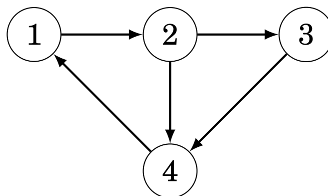
Edge list representation

An **edge list** contains all edges of a graph. This is a convenient way to represent a graph, if the algorithm will go trough all edges of the graph, and it is not needed to find edges that begin at a given node.

The edge list can be stored in a vector

```
vector<pair<int,int>> v;
```

where each element contains the starting and ending node of an edge. Thus, the graph



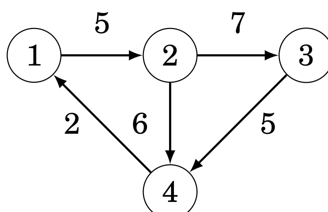
can be represented as follows:

```
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, we can extend the structure as follows:

```
vector<pair<pair<int,int>,int>> v;
```

Now the list contains pairs whose first element contains the starting and ending node of an edge, and the second element corresponds to the edge weight. For example, the graph



can be represented as follows:

```
v.push_back({{1,2},5});
v.push_back({{2,3},7});
v.push_back({{2,4},6});
v.push_back({{3,4},5});
v.push_back({{4,1},2});
```

Chapter 12

Graph search

This chapter introduces 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.

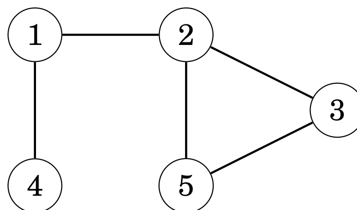
12.1 Depth-first search

Depth-first search (DFS) is a straightforward graph search 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 back 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.

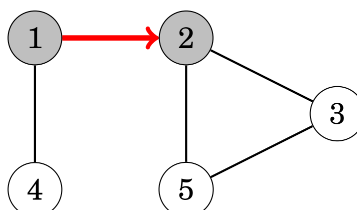
Example

Let's consider how depth-first search processes the following graph:

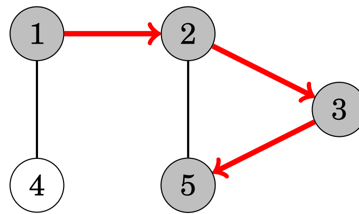


The algorithm can begin at any node in the graph, but we will now assume that it begins at node 1.

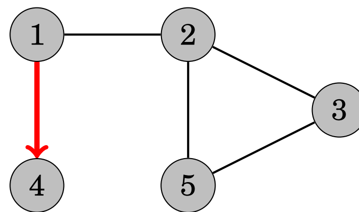
The search first proceeds to node 2:



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's time to return back. Also the neighbors of nodes 3 and 2 have been visited, so we'll next proceed 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 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);
    }
}
```

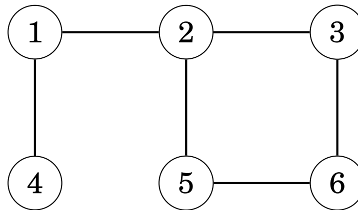
12.2 Breadth-first search

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.

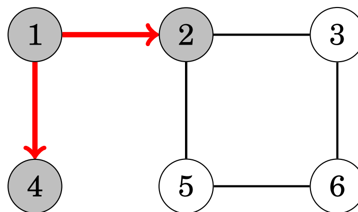
Breadth-first search goes through the nodes one level after another. First the search explores the nodes whose distance from the starting node is 1, then the nodes whose distance is 2, and so on. This process continues until all nodes have been visited.

Example

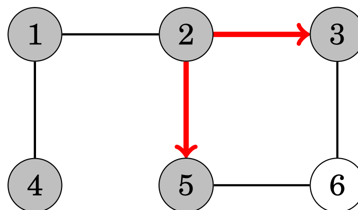
Let's consider how the algorithm processes the following graph:



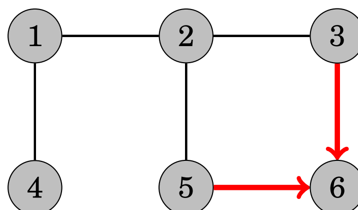
Assume again that the search begins at node 1. First, we process all nodes that can be reached from node 1 using a single edge:



After this, we procees 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 in the graph. A typical implementation is to maintain a queue of nodes to be processed. 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 array z indicates which nodes the search already has visited and array e will contain the minimum distance 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);
    }
}
```

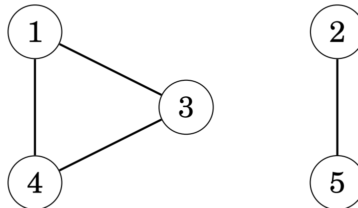
12.3 Applications

Using the graph search 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.

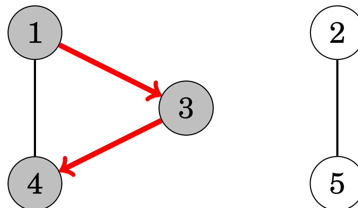
Connectivity check

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 selecting an arbitrary node and finding out if we can reach all other nodes.

For example, in the graph



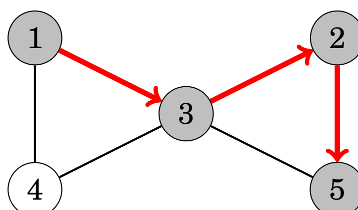
a depth-first search from node 1 visits the following nodes:



Since the search didn't visit all the nodes, we can conclude that the graph is not connected. In a similar way, we can also find all components in a graph by iterating trough the nodes and always starting a new depth-first search if the node doesn't belong to a component.

Finding cycles

A graph contains a cycle if during a graph search, we find a node whose neighbor (other than the previous node in the current path) has already been visited. For example, the graph



contains a cycle because when we move from node 2 to node 5 it turns out that the neighbor node 3 has already been visited. Thus, the graph contains a cycle that goes through node 3, for example, $3 \rightarrow 2 \rightarrow 5 \rightarrow 3$.

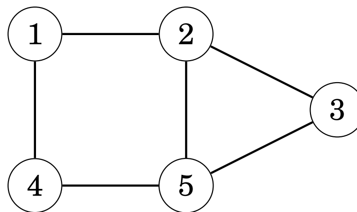
Another way to find out whether a graph contains a cycle is to simply calculate the number of nodes and edges in every component. If a component contains c nodes and no cycle, it must contain exactly $c-1$ edges. If there are c or more edges, the component always contains a cycle.

Bipartiteness check

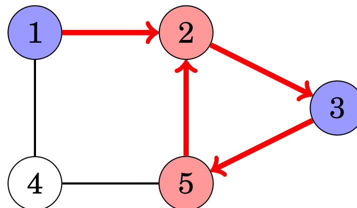
A graph is bipartite if its nodes can be colored using two colors so that there are no adjacent nodes with same color. It is suprisingly easy to check if a graph is bipartite using graph search algorithms.

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 produces the following situation:



We notice that the color or both node 2 and node 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 doesn't 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.

Chapter 13

Shortest paths

Finding the shortest path between two nodes is an important graph 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 for finding the shortest path. However, in this chapter we concentrate on weighted graphs. In this case we need more sophisticated algorithms for finding shortest paths.

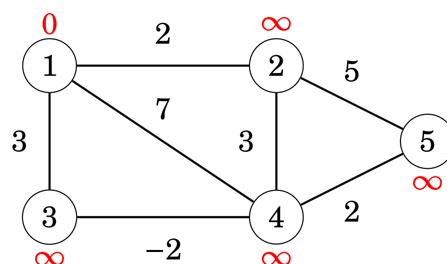
13.1 Bellman–Ford algorithm

The **Bellman–Fordin algoritmi** finds the shortest path from a starting node to all other nodes in the graph. The algorithm works in all kinds of graphs, provided that the graph doesn't contain a cycle with negative length. If the graph contains a negative cycle, the algorithm can detect this.

The algorithm keeps track of estimated distances from the starting node to other nodes. Initially, the estimated distance is 0 to the starting node and infinite to all other nodes. The algorithm improves the estimates by finding edges that shorten the paths until it is not possible to improve any estimate.

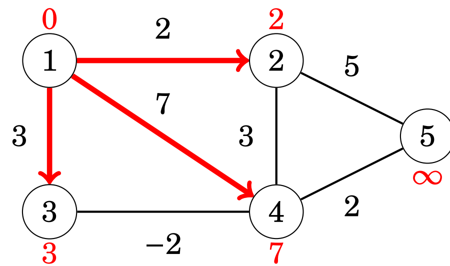
Example

Let's consider how the Bellman–Ford algorithm works in the following graph:

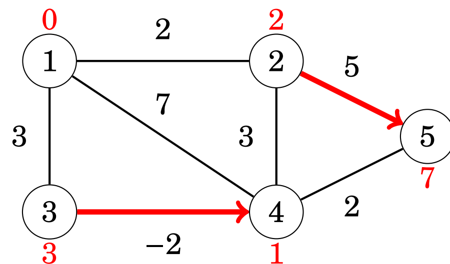


Each node in the graph is assigned an estimated distance. Initially, the distance is 0 to the starting node and infinite to all other nodes.

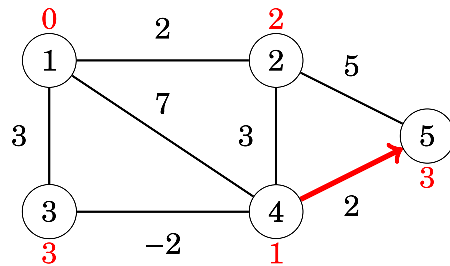
The algorithm searches for edges that improve the estimated distances. First, all edges from node 1 improve the estimates:



After this, edges $2 \to 5$ and $3 \to 4$ improve the estimates:

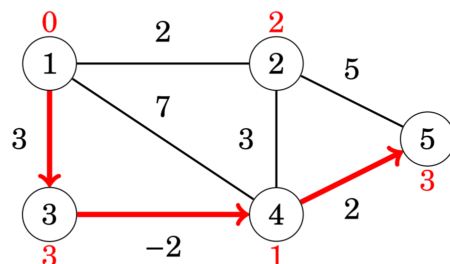


Finally, there is one more improvment:



After this, no edge improves the estimates. 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 smallest 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 paths from a node x to all other nodes in the graph. The code assumes that the graph is stored as adjacency lists in array

```
vector<pair<int,int>> v[N];
```

so that each pair contains the target node and the edge weight.

The algorithm consists of $n-1$ rounds, and on each round the algorithm goes through all nodes in the graph and tries to improve the estimated distances. The algorithm builds an array e that will contain the distance from x to all nodes in the graph. The initial value 10^9 means infinity.

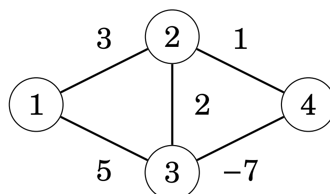
```
for (int i = 1; i <= n; i++) e[i] = 1e9;
e[x] = 0;
for (int i = 1; i <= n-1; i++) {
    for (int a = 1; a <= n; a++) {
        for (auto b : v[a]) {
            e[b.first] = min(e[b.first],e[a]+b.second);
        }
    }
}
```

The time complexity of the algorithm is $O(nm)$ because it consists of $n-1$ rounds and iterates through all m nodes 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 much faster than in $n-1$ rounds. Thus, a possible way to make the algorithm more efficient is to stop the algorithm if we can't improve any distance during a round.

Negative cycle

Using the Bellman–Ford algorithm we can also check if the graph contains a cycle with negative length. For example, the graph



contains a negative cycle $2 \to 3 \to 4 \to 2$ 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 here.

A negative cycle can be detected using the Bellman–Ford algorithm by running the algorithm for n rounds. If the last round improves any distance, the graph

contains a negative cycle. Note that this algorithm searches for a negative cycle in the whole graph regardless of the starting node.

SPFA algorithm

The **SPFA algoritmi** (”Shortest Path Faster Algorithm”) is a variation for the Bellman–Ford algorithm, that is often more efficient than the original algorithm. It doesn’t 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 improving 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 $a \to b$ improves a distance, node b is added to the end of the queue.

The following implementation uses a queue structure q. In addition, array z indicates if a node is already in the queue, in which case the algorithm doesn’t 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 usually very 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 standard Bellman–Ford algorithm.

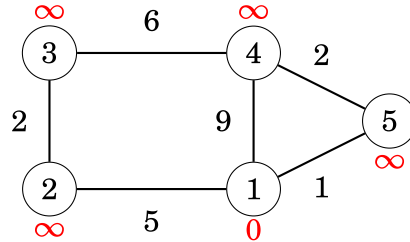
13.2 Dijkstra’s algorithm

Dijkstra’s algorithm 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 estimated distances for the nodes and improves them during the algorithm. 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

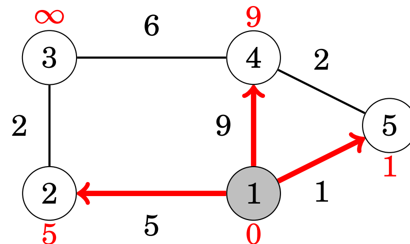
Let's consider how Dijkstra's algorithm works in the following graph when the starting node is node 1:



Like in the Bellman–Ford algorithm, the estimated distance is 0 to the starting node and infinite to all other nodes.

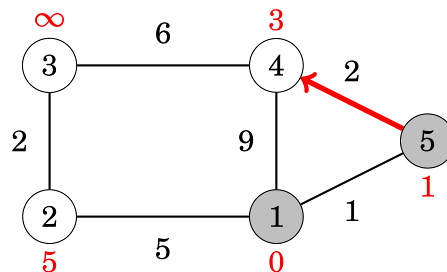
At each step, Dijkstra's algorithm selects a node that has not been processed yet and whose estimated distance is as small as possible. The first such node is node 1 with distance 0.

When a node is selected, the algorithm goes through all edges that begin from the node and improves the distances using them:

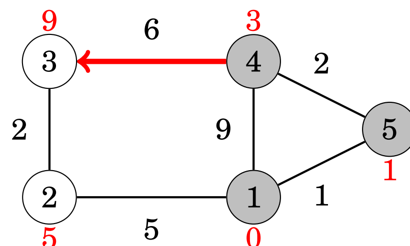


The edges from node 1 improved distances to nodes 2, 4 and 5 whose now distances are now 5, 9 and 1.

The next node to be processed is node 5 with distance 1:

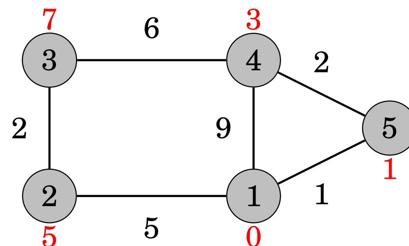


After this, the next node is node 4:



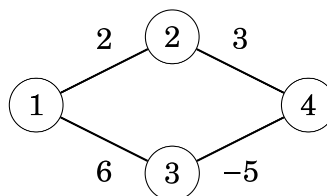
A nice property in Dijkstra's algorithm is that whenever a node is selected, its distance is final. For example, at this point of the algorithm, the distances 0, 1 and 3 are the final distances to nodes 1, 5 and 4.

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 doesn't 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 $1 \to 3 \to 4$, and its length is 1. However, Dijkstra's algorithm finds the path $1 \to 2 \to 4$ by following the lightest edges. The algorithm cannot recognize that in the lower path, the weight -5 compensates the previous large weight 6.

Implementation

The following implementation of Dijkstra's algorithm calculates the minimum distance from a node x to all other nodes. The graph is stored in an array `v` as adjacency lists that contain target nodes and weights for each edge.

An efficient implementation of Dijkstra's algorithm requires that it is possible to quickly find the smallest node that has not been processed. A suitable data structure for this is a priority queue that contains the nodes ordered by the estimated 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 estimated distance and second element is the identifier of the corresponding node.

```
priority_queue<pair<int,int>> q;
```

A small difficulty is that in Dijkstra's algorithm, we should find the node with *minimum* distance, while the C++ priority queue finds the *maximum* element as default. An easy solution is to use *negative* distances, so we can directly use the C++ priority queue.

The code keeps track of processed nodes in array z, and maintains estimated distances in array e. Initially, the distance to the starting node is 0, and the distance to all other nodes is 10^9 that corresponds to infinity.

```
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 estimated distance to the priority queue.

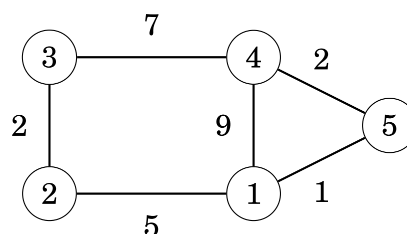
13.3 Floyd–Warshall algorithm

The **Floyd–Warshall algorithm** is an alternative way to approach the problem of finding shortest paths. Unlike 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 updates the distances by allowing to use intermediate nodes in the paths.

Example

Let's 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 successive rounds. On each round, one new node is selected that can act as intermediate node in paths, and the algorithm improves the distances in the array using this node.

On the first round, node 1 is the intermediate node. Now there is a new path between nodes 2 and 4 with length 14 because node 1 connects them. Correspondingly, there is 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 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

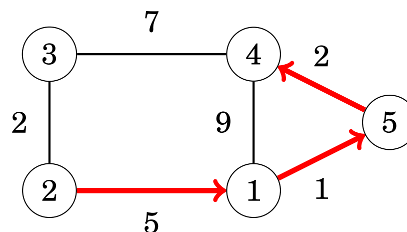
On the third round, node 3 is the intermediate round. There is a new path between nodes 2 and 4:

	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 intermediate nodes. After the algorithm has finished, the array contains the minimum distance 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 indicates that the shortest path between nodes 2 and 4 has length 8. This corresponds to the following path:



Implementation

The benefit in the Floyd–Warshall algorithm that it is easy to implement. The following code constructs a distance matrix `d` where `d`[a][b] is the smallest distance in a path between nodes a and b. First, the algorithm initializes `d` using the adjacency matrix `v` of the graph (value 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 paths 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.

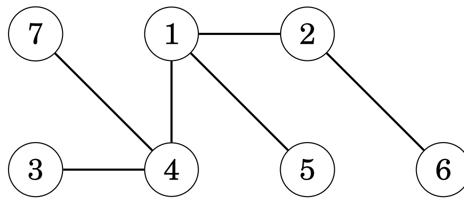
Since the implementation of the Floyd–Warshall algorithm is simple, the algorithm can be a good choice even if we need to find only a single shortest path in the graph. However, this is only possible when the graph is so small that a cubic time complexity is enough.

Chapter 14

Tree algorithms

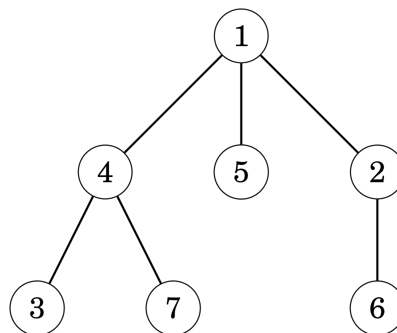
A **tree** is a connected, acyclic graph that contains n nodes and $n-1$ edges. Removing any edge from a tree divides it into two components, and adding any edge to a tree creates a cycle. Moreover, there is always a unique path between any two nodes in a tree.

For example, the following tree contains 7 nodes and 6 edges:



The **leaves** of a tree are nodes with degree 1, i.e., with only one neighbor. For example, the leaves in the above tree are nodes 3, 5, 6 and 7.

In a **rooted** tree, one of the nodes is chosen to be a **root**, 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 **childern** of a node are its lower neighbors, and the **parent** of a node is its upper neighbor. Each node has exactly one parent, except that the root doesn't have a parent. For example, in the above tree, the childern 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 travelling downwards in the tree. For example, in the above tree, the subtree of node 4 contains nodes 4, 3 and 7.

14.1 Tree search

Depth-first search and breadth-first search can be used for going through the nodes in a tree. However, the search is easier to implement than for a general graph, because there are no cycles in the tree, and it is not possible that the search would visit a node several times.

Often, we start a depth-first search from a chosen root node. The following recursive function implements it:

```
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 idea of the parameter e is to ensure that the search only proceeds downwards in the tree towards 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

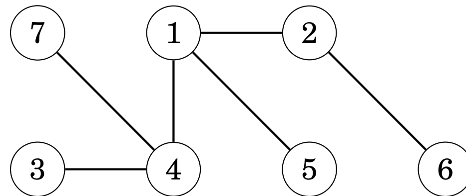
We can also use dynamic programming to calculate some information from the tree during the search. Using dynamic programming, we can, for example, calculate in $O(n)$ time for each node the number of nodes in its subtree, or the length of the longest path downwards that begins at the node.

As an example, let's 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 haku(int s, int e) {
    c[s] = 1;
    for (auto u : v[s]) {
        if (u == e) continue;
        haku(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



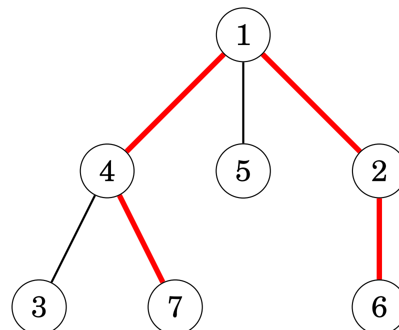
the diameter is 4, and it corresponds to two paths: the path between nodes 3 and 6, and the path between nodes 7 and 6.

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, one of the nodes is chosen to be the root. After this, the algorithm calculates for each node the length of the longest path that begins at some leaf, ascends to the node and then descends to another leaf. The length of the longest such path equals the diameter of the tree.

In the example case, the longest path begins at node 7, ascends to node 1, and then descends to node 6:



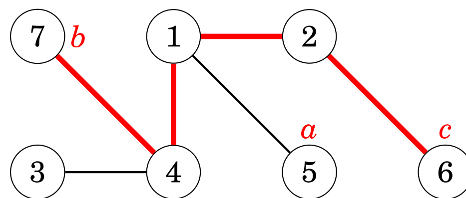
The algorithm first calculates using dynamic programming for each node the length of the longest path that goes downwards from the node. For example, in the above tree, the longest path from node 1 downwards has length 2 (the path can be $1 \to 4 \to 3$, $1 \to 4 \to 7$ or $1 \to 2 \to 6$).

After this, the algorithm calculates for each node the length of the longest path where the node is the turning point of the path. The longest such path can be found by selecting two children with longest paths downwards. For example, in the above graph, nodes 2 and 4 are chosen for node 1.

Algorithm 2

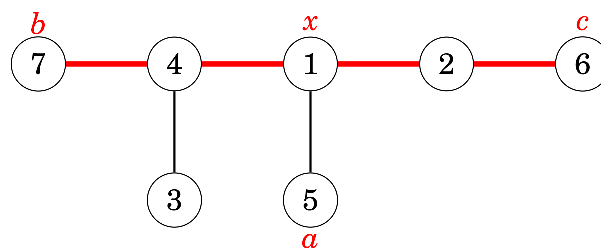
Another efficient way to calculate the diameter of a tree is based on two depth-first searches. First, we choose an arbitrary node a in the tree and find a node b with maximum distance to a. Then, we find a node c with maximum distance to b. The diameter is the distance between nodes b and c.

In the example case, 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:

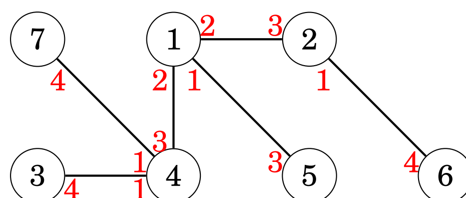


Node x indicates the place where the path from node a joins the path that corresponds to the diameter. The farthest node from a is node b, node c or some other node that is at least as far from node x. Thus, this node can always be chosen for a starting node of a path that corresponds to the diameter.

14.3 Distances between nodes

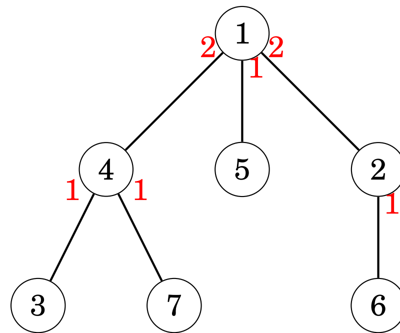
A more difficult problem is to calculate for each node in the tree and for each direction, the maximum distance to a node in that direction. It turns out that this can be calculated in $O(n)$ time as well using dynamic programming.

In the example case, the distances are as follows:



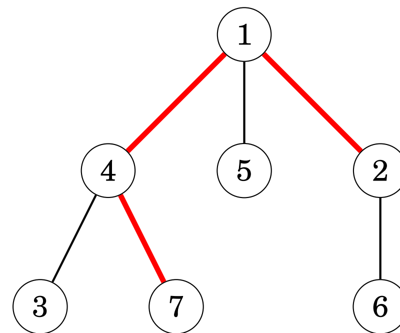
For example, the furthest node from node 4 upwards is node 6, and the distance to this node is 3 using the path $4 \rightarrow 1 \rightarrow 2 \rightarrow 6$.

Also in this problem, a good starting point is to root the tree. After this, all distances downwards can be calculated using dynamic programming:

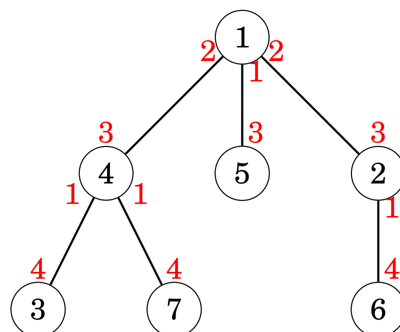


The remaining task is to calculate the distances upwards. This can be done by going through the nodes once again and keeping track of the largest distance from the parent of the current node to some other node in another direction.

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:



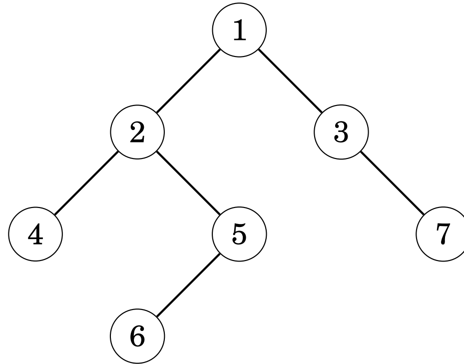
Finally, we can calculate the distances for all nodes and all directions:



14.4 Binary trees

A **binary tree** is a rooted tree where each node has a left subtree and a right subtree. It is possible that a subtree of a node is empty. Thus, every node in a binary tree has 0, 1 or 2 children.

For example, the following tree is a binary tree:



The nodes in a binary tree have three natural orders that correspond to different ways to recursively traverse the nodes:

- **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 the in-order of a tree, we can find out the exact structure of the tree. For example, the tree above is the only possible tree with pre-order $[1,2,4,5,6,3,7]$ and in-order $[4,2,6,5,1,3,7]$. Correspondingly, the post-order and the in-order also determine the structure of a tree.

However, the situation is different if we only know the pre-order and the post-order of a tree. In this case, there may be more than one tree that match the orders. For example, in both of the trees

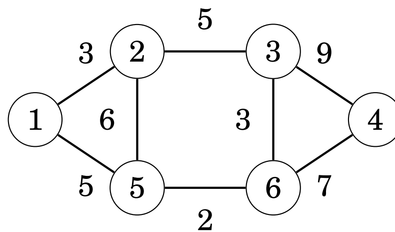
the pre-order is $[1,2]$ and the post-order is $[2,1]$ but the trees have different structures.

Chapter 15

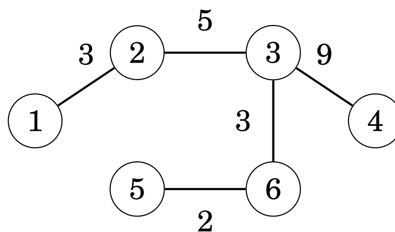
Spanning trees

A **spanning tree** is a set of edges of a graph such that there is a path between any two nodes in the graph using only the edges in the spanning tree. Like trees in general, a spanning tree is connected and acyclic. Usually, there are many ways to construct a spanning tree.

For example, in the graph

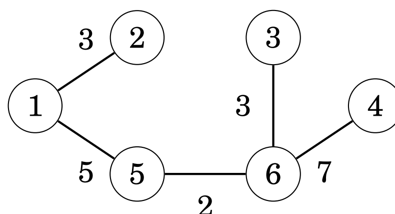


one possible spanning tree is as follows:

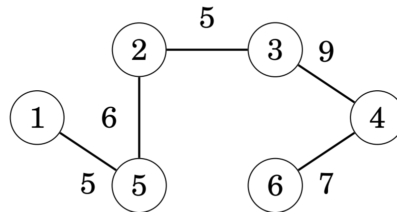


The weight of a spanning tree is the sum of the edge weights. For example, the weight of the above spanning tree is $3+5+9+3+2=22$.

A **minimum spanning tree** is a spanning tree whose weight is as small as possible. The weight of a minimum spanning tree for the above graph is 20, and a tree can be constructed as follows:



Correspondingly, a **maximum spanning tree** is a spanning tree whose weight is as large as possible. The weight of a maximum spanning tree for the above graph is 32:



Note that there may be several different ways for constructing a minimum or maximum spanning tree, so the trees are not unique.

This chapter discusses algorithms that construct a minimum or maximum spanning tree for a graph. It turns out that it is easy to find such spanning trees because many greedy methods produce an optimal solution.

We will learn two algorithms that both construct the tree by choosing edges ordered by weights. We will focus on finding a minimum spanning tree, but the same algorithms can be used for finding a maximum spanning tree by processing the edges in reverse order.

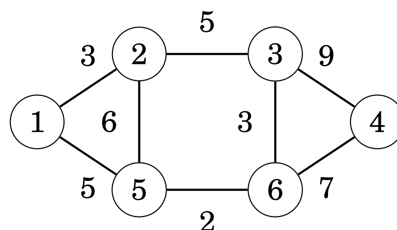
15.1 Kruskal's algorithm

In **Kruskal's algorithm**, the initial spanning tree is empty and doesn't contain any edges. Then the algorithm adds edges to the tree one at a time in increasing order of their weights. At each step, the algorithm includes an edge in the tree if it doesn't create a cycle.

Kruskal's algorithm maintains the components in the tree. Initially, each node of the graph is in its own component, and each edge added to the tree joins two components. Finally, all nodes will be in the same component, and a minimum spanning tree has been found.

Example

Let's consider how Kruskal's algorithm processes the following graph:

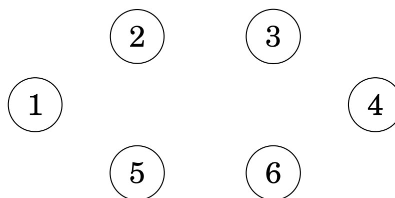


The first step in the algorithm is to sort the edges in increasing order of their weights. The result is the following list:

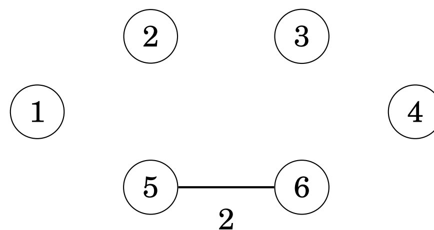
edge	weight
5–6	2
1–2	3
3–6	3
1–5	5
2–3	5
2–5	6
4–6	7
3–4	9

After this, the algorithm goes through the list and adds an edge to the tree if it joins two separate components.

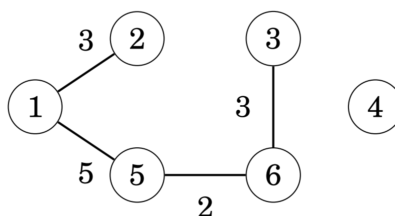
Initially, each node is in its own component:



The first edge to be added to the tree is edge 5–6 that joins components {5} and {6} into component {5,6}:



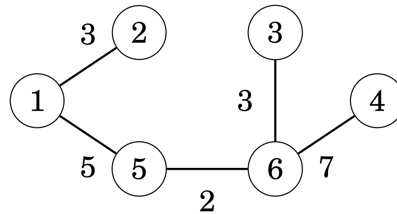
After this, edges 1–2, 3–6 and 1–5 are added in a similar way:



After those steps, many 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 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, edge 2–5 will not be added to the tree.

Finally, edge 4–6 will be included in the tree:

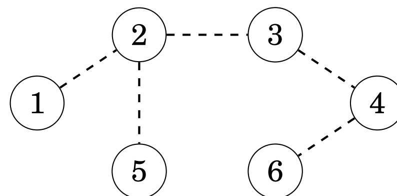


After this, the algorithm terminates because there is a path between any two nodes and the graph is connected. The resulting graph is a minimum spanning tree with weight $2+3+3+5+7=20$.

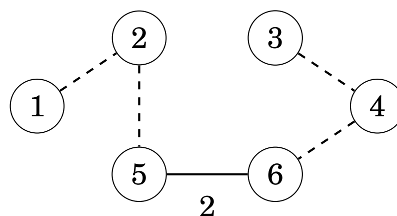
Why does this work?

It's a good question why Kruskal's algorithm works. Why does the greedy strategy guarantee that we will find a minimum spanning tree?

Let's see what happens if the lightest edge in the graph is not included in the minimum spanning tree. For example, assume that a minimum spanning tree for the above graph would not contain the edge between nodes 5 and 6 with weight 2. We don't know exactly how the new minimum spanning tree would look like, but still it has to contain some edges. Assume that the tree would be as follows:



However, it's not possible that the above tree would be a real minimum spanning tree for the graph. The reason for this is that we can remove an edge from it and replace it with the edge with weight 2. This produces a spanning tree whose weight is *smaller*:



For this reason, it is always optimal to include the lightest edge in the minimum spanning tree. Using a similar argument, we can show that we can also add the second lightest edge to the tree, and so on. Thus, Kruskal's algorithm works correctly and always produces a minimum spanning tree.

Implementation

Kruskal's algorithm can be conveniently implemented using an edge list. The first phase of the algorithm sorts the edges in $O(m \log m)$ time. After this, the second phase of the algorithm builds the minimum spanning tree.

The second phase of the algorithm looks 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. The code uses two functions: the function `same` determines if the nodes are in the same component, and the function `unite` joins two components into a single component.

The problem is how to efficiently implement the functions `same` and `unite`. One possibility is to maintain the graph in a usual way and implement the function `same` as graph traversal. However, using this technique, the running time of the function `same` would be $O(n+m)$, and this would be slow because the function will be called for each edge in the graph.

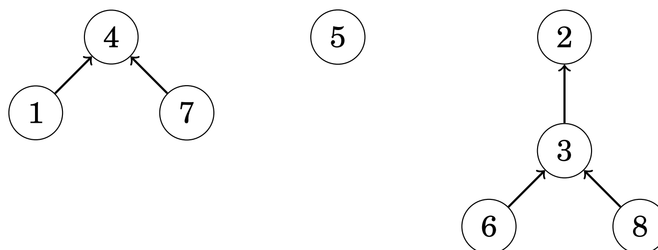
We will solve the problem using a union-find structure that implements both the functions in $O(\log n)$ time. Thus, the time complexity of Kruskal's algorithm will be $O(m \log n)$ after sorting the edge list.

15.2 Union-find structure

The **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 first operation checks if two elements belong to the same set, and the second operation joins two sets into a single set.

Structure

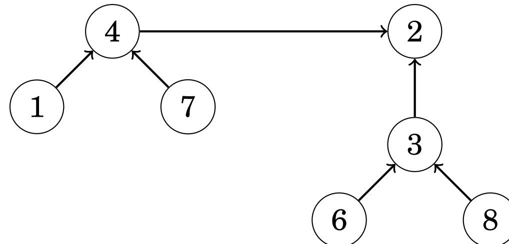
In the union-find structure, one element in each set is the representative of the set. All other elements in the set point to the representative directly or through other elements in the set. For example, in the following picture there are three sets: $\{1,4,7\}$, $\{5\}$ and $\{2,3,6,8\}$.



In this case the representatives of the sets are 4, 5 and 2. For each element, we can find the representative for the corresponding set by following the path that

begins at the element. For example, element 2 is the representative for the set that contains element 6 because the path is $6 \to 3 \to 2$. Thus, two elements belong to the same set exactly when they point to the same representative.

Two sets can be combined by connecting the representative of one set to the representative of another set. For example, sets $\{1,4,7\}$ and $\{2,3,6,8\}$ can be combined as follows into set $\{1,2,3,4,6,7,8\}$:



In this case, element 2 becomes the representative for the whole set and the old representative 4 points to it.

The efficiency of the operations depends on the way the sets are combined. It turns out that we can follow a simple strategy and always connect the representative of the smaller set to the representative of the larger set (or, if the sets are of the same size, both choices are fine). Using this strategy, the length of a path from a element in a set to a representative is always $O(\log n)$ because each step forward in the path doubles the size of the corresponding set.

Implementation

We can implement the union-find structure using arrays. In the following implementation, array `k` contains for each element the next element in the path, or the element itself if it is a representative, and array `s` indicates for each representative the size of the corresponding set.

Initially, each element has an own set with size 1:

```
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 element x. The representative can be found by following the path that begins at element x.

```
int find(int x) {
    while (x != k[x]) x = k[x];
    return x;
}
```

The function `same` finds out 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 combines the sets that contain elements a and b into a single set. 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[b] > s[a]) 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 the path is $O(\log n)$. Thus, the functions same and union also work in $O(\log n)$ time. The function union ensures that the length of each path is $O(\log n)$ by connecting the smaller set to the larger set.

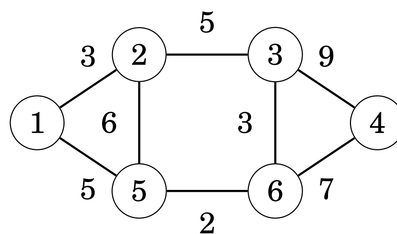
15.3 Prim's algorithm

Prim's algorithm is an alternative method for finding a minimum spanning tree. The algorithm first adds an arbitrary node to the tree, and then always selects an edge whose weight is as small as possible and 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.

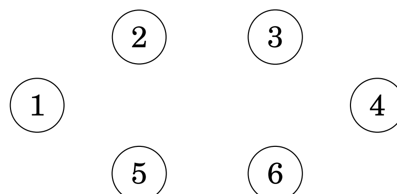
Prim's algorithm resembles Dijkstra's algorithm. The difference is that Dijkstra's algorithm always selects an edge that creates a shortest path from the starting node to another node, but Prim's algorithm simply selects the lightest edge that adds a new node to the tree.

Example

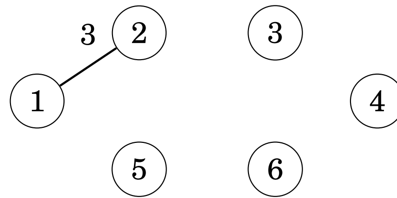
Let's consider how Prim's algorithm works in the following graph:



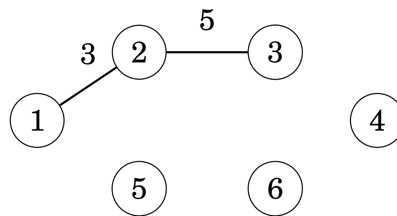
Initially, there are no edges between the nodes:



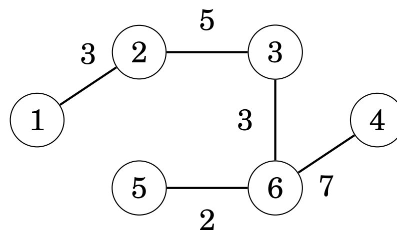
We can select an arbitrary node as a starting node, so let's select node 1. First, an edge with weight 3 connects nodes 1 and 2:



After this, there are two edges with weight 5, so we can add either node 3 or node 5 to the tree. Let's 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. In this case, the priority queue contains 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 algorithm and Kruskal's algorithm 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

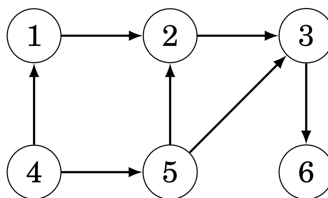
In this chapter, we focus on two classes of directed graphs:

- **Acyclic graph**: There are no cycles in the graph, so there is no path from any node to itself.
- **Successor graph**: The outdegree of each node is 1, so each node has a unique successor.

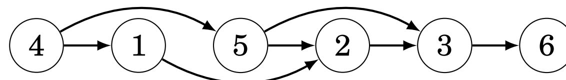
It turns out that in both cases, we can design efficient algorithms that are based on the special properties of the graphs.

16.1 Topological sorting

A **topological sort** is a ordering of the nodes of a directed graph where node a is always before node b if there is a path from node a to node b. For example, for the graph



a possible topological sort is $[4,1,5,2,3,6]$:



A topological sort always exists if the graph is acyclic. However, if the graph contains a cycle, it is not possible to find a topological sort because no node in the cycle can appear before other nodes in the cycle in the ordering. It turns out that we can use depth-first search to both construct a topological sort or find out that it is not possible because the graph contains a cycle.

Algorithm

The idea is to go through the nodes in the graph and always begin a depth-first search if the node has not been processed yet. During each search, 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)

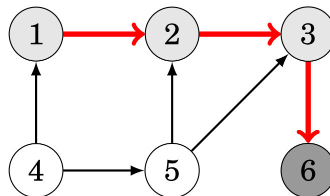
Initially, the state of each node is 0. When a search reaches a node for the first time, the state of the node becomes 1. Finally, after all neighbors of a node have been processed, the state of the node becomes 2.

If the graph contains a cycle, we will realize 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.

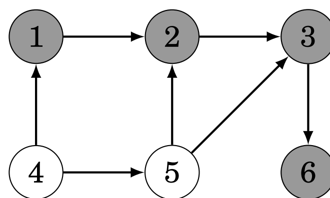
If the graph doesn't contain a cycle, we can construct a topological sort by adding each node to the end of a list when its state becomes 2. This list in reverse order is a topological sort.

Example 1

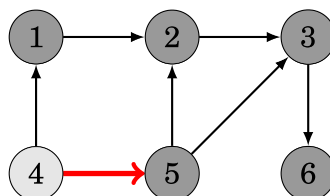
In the example graph, the search first proceeds from node 1 to node 6:



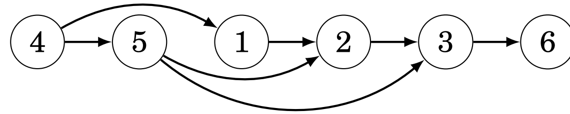
Now node 6 has been processed, so it is added to the list. After this, the search returns back:



At this point, the list contains values $[6,3,2,1]$. The next search begins at node 4:



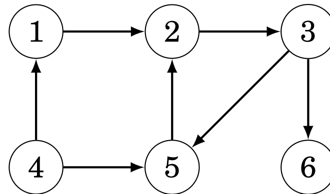
Thus, the final list is $[6,3,2,1,5,4]$. We have processed all nodes, so a topological sort has been found. The topological sort is the reverse list $[4,5,1,2,3,6]$:



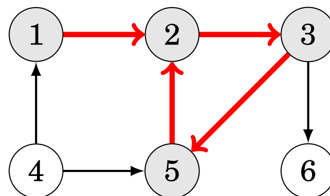
Note that a topological sort is not unique, but there can be several topological sorts for a graph.

Example 2

Let's consider another example where we can't construct a topological sort because there is a cycle in the graph:



Now the search proceeds as follows:



The search reaches node 2 whose state is 1 which means the graph contains a cycle. In this case, the cycle is $2 \rightarrow 3 \rightarrow 5 \rightarrow 2$.

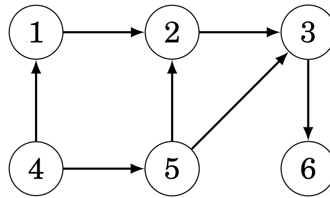
16.2 Dynamic programming

If a directed graph is acyclic, dynamic programming can be applied to it. For example, we can solve the following problems concerning paths from a starting node to an ending node efficiently in $O(n+m)$ time:

- 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 the path?

Counting the number of paths

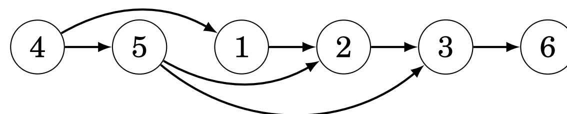
As an example, let's calculate the number of paths from a starting node to an ending node in a directed, acyclic graph. For example, in the graph



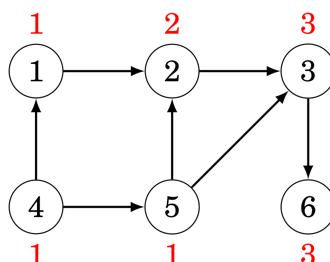
there are 3 paths from node 4 to node 6:

- $4 \to 1 \to 2 \to 3 \to 6$
- $4 \to 5 \to 2 \to 3 \to 6$
- $4 \to 5 \to 3 \to 6$

The idea is to go through the nodes in a topological sort, and calculate for each node the total number of paths that reach the node from different directions. In this case, the topological sort is as follows:



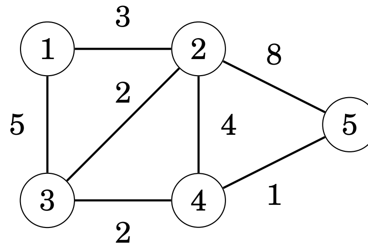
The numbers of paths are as follows:



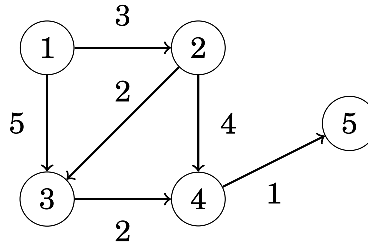
For example, there is an edge to node 2 from nodes 1 and 5. There is one path from node 4 to both node 1 and 5, so there are two paths from node 4 to node 2. Correspondingly, there is an edge to node 3 from nodes 2 and 5 that correspond to two and one paths from node 4.

Extending Dijkstra's algorithm

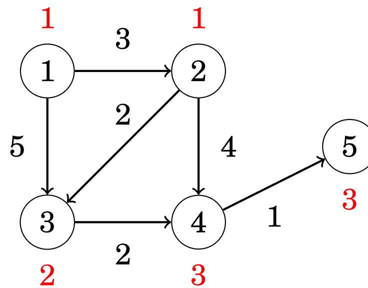
A by-product of Dijkstra's algorithm is a directed, acyclic graph that shows for each node in the original graph the possible ways to reach the node using a shortest path from the starting node. Dynamic programming can be applied also to this graph. For example, in the graph



the following edges can belong to the shortest paths from node 1:



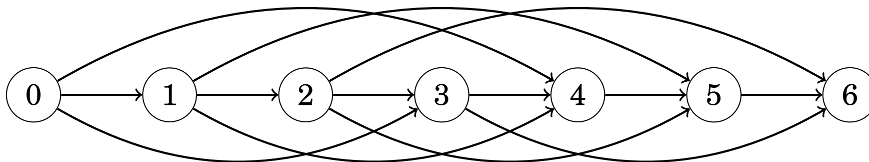
Now we can, for example, calculate the number of shortest paths from node 1 to node 5 using dynamic programming:



Representing problems as graphs

Actually, any dynamic programming problem can be represented as a directed, acyclic graph. In such a graph, each node is a dynamic programming state, and the edges indicate how the states depend on each other.

As an example, consider the problem where our task is to form a sum of money x using coins $\{c_1, c_2, \ldots, c_k\}$. In this case, we can construct a graph where each node corresponds to a sum of money, and the edges show how we can choose coins. 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

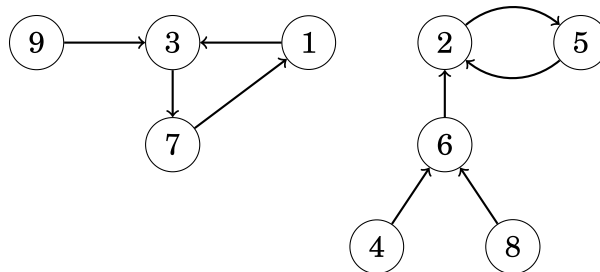
For the rest of the chapter, we concentrate on **successor graphs** where the outdegree of each node is 1, i.e., exactly one edge begins at the node. Thus, the graph consists of one or more components, and each component contains one cycle and some paths that lead to it.

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 returns the successor of the node.

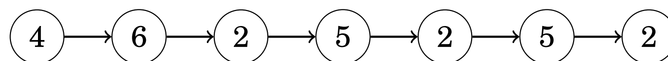
For example, the function

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 by walking 6 steps from node 4, we will reach node 2:



A straightforward way to calculate a value $f(x,k)$ is to walk through the path step by step which takes $O(k)$ time. However, using preprocessing, we can calculate any value $f(x,k)$ 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 done efficiently 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 we calculate $O(\log u)$ values 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
...									

After this, any value $f(x,k)$ can be calculated by presenting the value k as a sum of powers of two. For example, if we want to calculate the value $f(x,11)$, we first form the representation $11 = 8+2+1$. Using this,

$$f(x,11) = f(f(f(x,8),2),1).$$

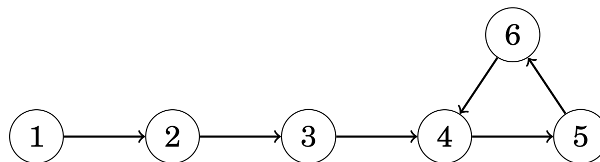
For example, in the above graph

$$f(4,11) = f(f(f(4,8),2),1) = 5.$$

Such a representation always consists of $O(\log k)$ parts so calculating a value $f(x,k)$ takes $O(\log k)$ time.

16.4 Cycle detection

Interesting questions in a successor graph are which node is the first node in the cycle if we begin our walk at node x, and what is the size of the cycle. For example, in the graph



if we begin 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 a cycle is to walk in the graph beginning from node x and keep track of all visited nodes. Once we will visit a node for the second time, the first node in the cycle has been found. 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 those algorithms is still $O(n)$, but they only use $O(1)$ memory. This is an important improvement if n is large. Next we will learn 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 the starting node of the graph. Then, on each turn, pointer a walks one step forward, while pointer b walks two steps forward. The search continues like that until the pointers will meet each other:

```
a = f(x);
b = f(f(x));
while (a != b) {
    a = f(a);
    b = f(f(b));
}
```

At this point, pointer a has walked k steps, and pointer b has walked $2k$ steps where the length of the cycle divides k. Thus, the first node that belongs to the cycle can be found by moving pointer a to the starting node and advancing the pointers step by step until they will 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:

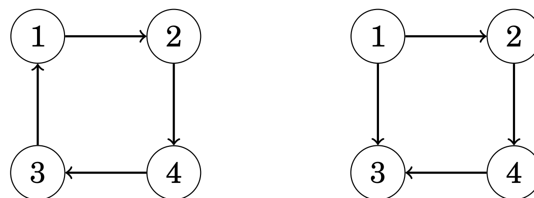
```
b = f(a);
c = 1;
while (a != b) {
    b = f(b);
    c++;
}
```

Chapter 17

Strongly connectivity

In a directed graph, the directions of the edges restrict possible paths in the graph, so even if the graph is connected, this doesn't guarantee that there would be a path between any two nodes. Thus, it is meaningful to define a new concept for directed graphs that requires more than connectivity.

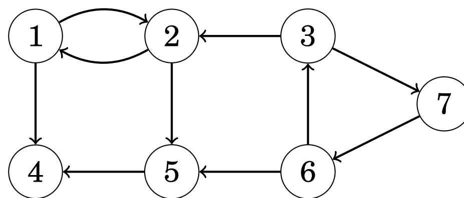
A graph is **strongly connected** if there is a path from any node to all other nodes in the graph. For example, in the following picture, the left graph is strongly connected, while the right graph is not.



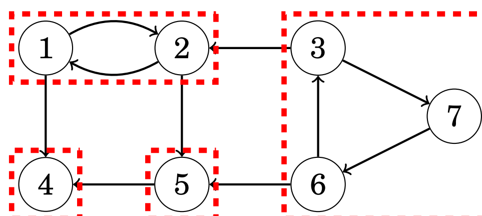
The right graph is not strongly connected because, for example, there is no path from node 2 to node 1.

The **strongly connected components** of a graph divide the graph into strongly connected subgraphs that are as large as possible. The strongly connected components form an acyclic **component graph** that represents the deep structure of the original graph.

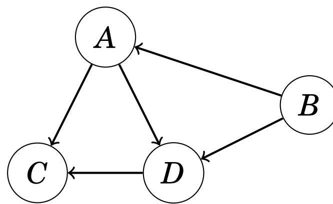
For example, for the graph



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 because it doesn't contain cycles. Thus, as in Chapter 16, it is possible to construct a topological sort for a component graph and also use dynamic programming algorithms.

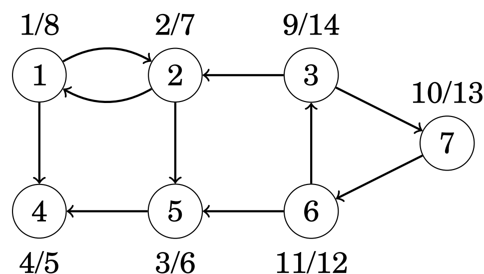
17.1 Kosaraju's algorithm

Kosaraju's algorithm is an efficient method for finding the strongly connected components of a directed graph. It 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 the 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 moment x and ended at moment y. When the nodes are sorted according to ending times, the result is the following list:

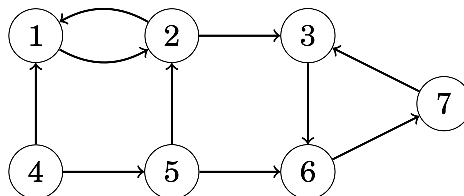
node	ending time
4	5
5	6
2	7
1	8
6	12
7	13
3	14

In the second phase of the algorithm, the nodes will be processed in reverse order: $[3,7,6,1,2,5,4]$.

Search 2

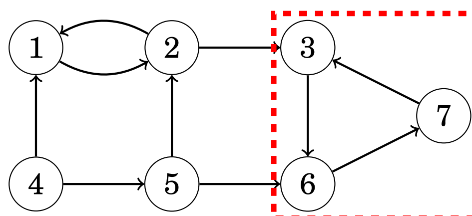
The second phase of the algorithm forms the strongly connected components of the graph. First, the algorithm reverses every edge in the graph. This ensures that during the second search, we will always find a strongly connected component without extra nodes.

The example graph becomes as follows after reversing the edges:



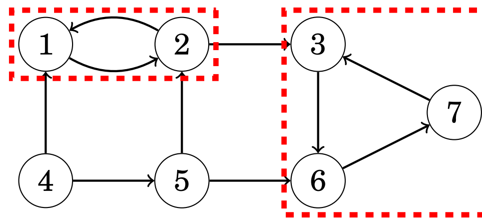
After this, the algorithm goes through the nodes in the order defined by the first search. If a node doesn't belong to a component, the algorithm creates a new component and begins a depth-first search where all new nodes found during the search are added to the new component.

In the example graph, the first component begins at node 3:

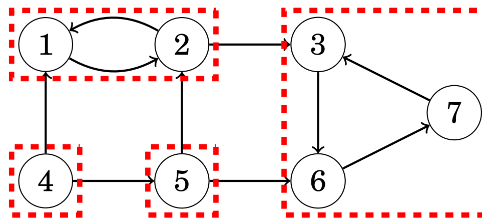


Note that since we reversed all edges in the graph, the component doesn't "leak" to other parts in the graph.

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 5 that create the remaining strongy connected components:



The time complexity of the algorithm is $O(n+m)$ where n is the number of nodes and m is the number of edges. The reason for this is that the algorithm performs two depth-first searches and each search takes $O(n+m)$ time.

17.2 2SAT problem

Strongly connectivity is also linked with the **2SAT problem**. 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 it 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 x_1 and x_2 are false and x_3 and x_4 are true. However, the formula

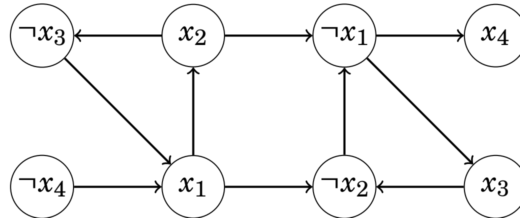
$$L_2 = (x_1 \vee x_2) \wedge (x_1 \vee \neg x_2) \wedge (\neg x_1 \vee x_3) \wedge (\neg x_1 \vee \neg x_3)$$

is always false. The reason for this is that we can't choose a value for variable x_1 without creating a contradiction. If x_1 is false, both x_2 and $\neg x_2$ should hold which is impossible, and if x_1 is true, both x_3 and $\neg x_3$ should hold which is impossible as well.

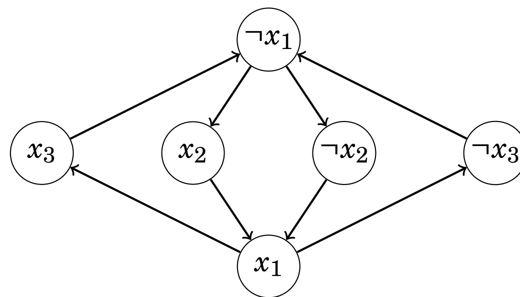
The 2SAT problem can be represented as a graph where the nodes correspond to variables x_i and negations $\neg x_i$, and the edges determine the connections

between the variables. Each pair $(a_i \lor b_i)$ generates two edges: $\neg a_i \to b_i$ and $\neg b_i \to a_i$. This means that if a_i doesn't hold, b_i must hold, and vice versa.

The graph for formula L_1 is:



And the graph for formula L_2 is:

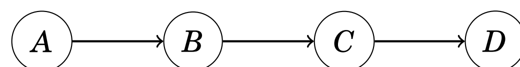


The structure of the graph indicates whether the corresponding 2SAT problem can be solved. If there is a variable x_i such that both x_i and $\neg x_i$ belong to the same strongly connected component, then there are no solutions. In this case, the graph contains a path from x_i to $\neg x_i$, and also a path from $\neg x_i$ to x_i, so both x_i and $\neg x_i$ should hold which is not possible. However, if the graph doesn't contain such a variable, then there is always a solution.

In the graph of formula L_1 no nodes x_i and $\neg x_i$ belong to the same strongly connected component, so there is a solution. In the graph of 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 processing the nodes of the component graph in a reverse topological sort order. At each step, we process and remove a component that doesn't contain edges that lead to the remaining components. If the variables in the component don't have values, their values will be determined according to the component, and if they already have values, they are not changed. The process continues until all variables have been assigned a value.

The component graph for 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 component D where x_4 becomes true. After this, we process 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 don't change the variables anymore.

Note that this method works because the structure of the graph is special. 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 $(a_i \vee b_i \vee c_i)$. No efficient algorithm for solving this problem is known, but it is a NP-hard problem.

Chapter 18

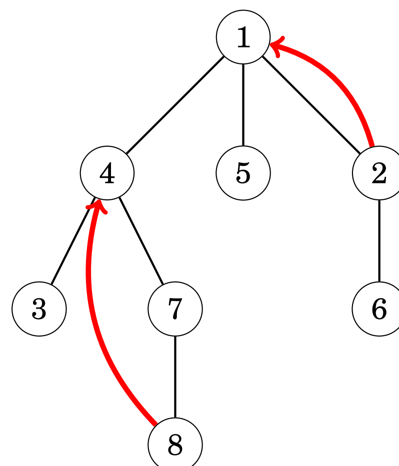
Tree queries

This chapter discusses techniques for efficiently performing queries for a rooted tree. The queries are related to subtrees and paths in the tree. For example, possible queries are:

- what is the kth ancestor of node x?
- what is the sum of values in the subtree of node x?
- what is the sum of values in a path between nodes a and b?
- what is the lowest common ancestor of nodes a and b?

18.1 Finding ancestors

The kth ancestor of node x in the tree is found when we ascend k steps in the tree beginning at node x. Let $f(x,k)$ denote the kth ancestor of node x. For example, in the following tree, $f(2,1) = 1$ and $f(8,2) = 4$.



A straighforward way to calculate $f(x,k)$ is to move k steps upwards in the tree beginning from node x. However, the time complexity of this method is $O(n)$ because it is possible that the tree contains a chain of $O(n)$ nodes.

As in Chapter 16.3, any value of $f(x,k)$ can be efficiently calculated in $O(\log k)$ 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 tree above are as follows:

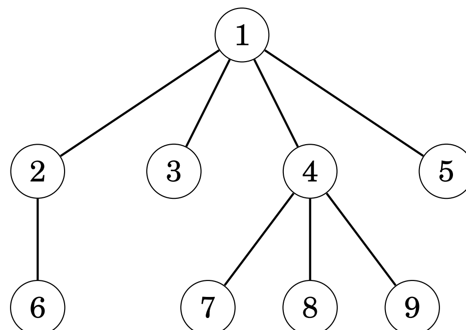
x	1	2	3	4	5	6	7	8
$f(x,1)$	0	1	4	1	1	2	4	7
$f(x,2)$	0	0	1	0	0	1	1	4
$f(x,4)$	0	0	0	0	0	0	0	0
...								

The value 0 means that the kth ancestor of a node doesn't exist.

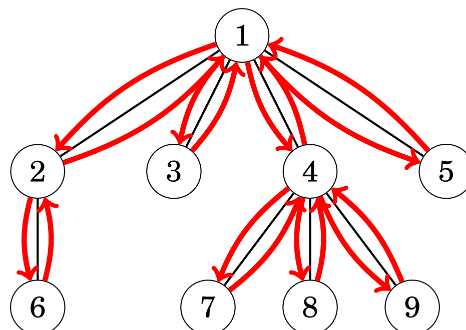
The preprocessing takes $O(n \log n)$ time because each node can have at most n ancestors. After this, any value $f(x,k)$ can be calculated in $O(\log k)$ time by representing the value k as a sum where each term is a power of two.

18.2 Subtrees and paths

A **node 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 node array is as follows:

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

Subtree queries

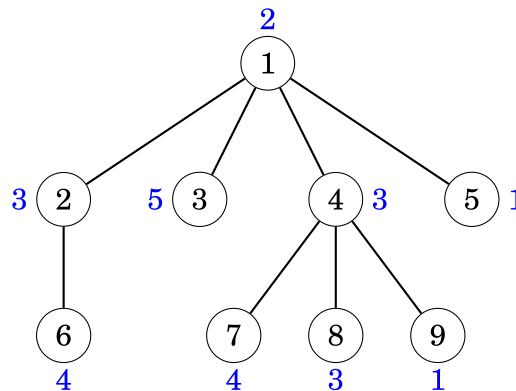
Each subtree of a tree corresponds to a subarray in the node array, where the first element is the root node. For example, the following subarray contains the nodes in the subtree of node 4:

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 the tree. As an example, consider a problem where each node is assigned a value, and our task is to support the following queries:

- change the value of node x
- calculate the sum of values in the subtree of node x

Let us consider the following tree where blue numbers are values of nodes. For example, the sum of values in the subtree of node 4 is $3+4+3+1=11$.



The idea is to construct a node array that contains three values for each node: (1) identifier of the node, (2) size of the subtree, and (3) 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 nodes in a subtree by first reading 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

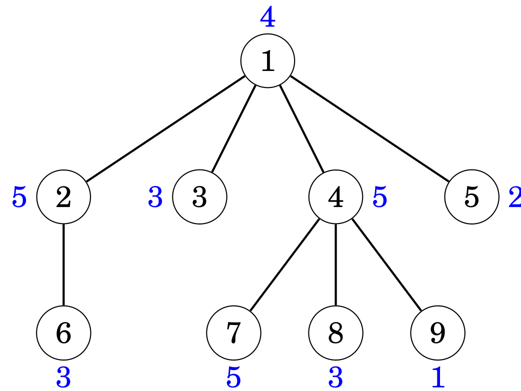
The remaining step is to store the values of the nodes in a binary indexed tree or segment tree. After this, we can both calculate the sum of values and change a value in $O(\log n)$ time, so we can efficiently process the queries.

Path queries

Using a node array, we can also efficiently process paths between the root node and 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 node x
- calculate the sum of values from the root to node x

For example, in the following tree, the sum of values from the root to node 8 is $4+5+3=12$.



To solve this problem, we can use a similar technique as we used for subtree queries, but the values of the nodes are stored in a special way: if the value of a node at index k increases by a, the value at index k increases by a and the value at index $k+c$ decreases by a, where c is the size of the subtree.

For example, the following array corresponds to the above tree:

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

For example, the value of node 3 is -5, because it is the next node after the subtrees of nodes 2 and 6 and its own value is 3. So the value decreases by $5+3$ and increases by 3. Note that the array contains an extra index 10 that only has the opposite number of the value of the root node.

Using this array, the sum of values in a path from the root to node x equals the sum of values in the array from the beginning to node x. For example, the sum from the root to node 8 can be calculated as follows:

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

The sum is

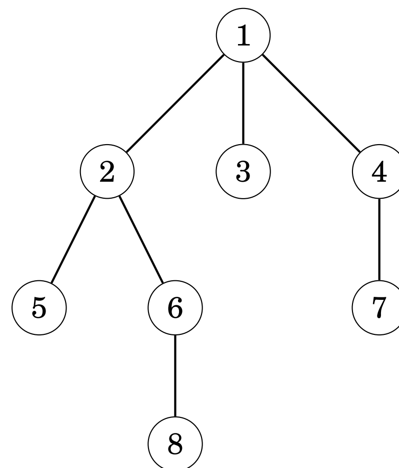
$$4+5+3-5+2+5-2=12,$$

that equals the sum $4+5+3=12$. This method works because the value of each node is added to the sum when the depth-first search visits it for the first time, and correspondingly, the value is removed from the sum when the subtree of the node has been processed.

Once again, we can store the values of the nodes in a binary indexed tree or a segment tree, so it is possible to both calculate the sum of values and change a value efficiently in $O(\log n)$ time.

18.3 Lowest common ancestor

The **lowest common ancestor** of two nodes is a the lowest node in the tree whose subtree contains both the nodes. A typical problem is to efficiently process queries where the task is to find the lowest common ancestor of two nodes. For example, in the tree



the lowest common ancestor of nodes 5 and 8 is node 2, and the lowest common ancestor of nodes 3 and 4 is node 1.

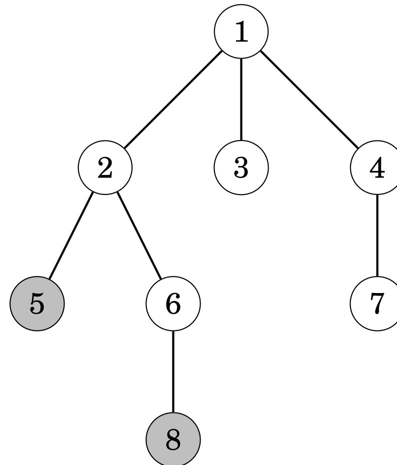
Next we will discuss two efficient techniques for finding the lowest common ancestor of two nodes.

Method 1

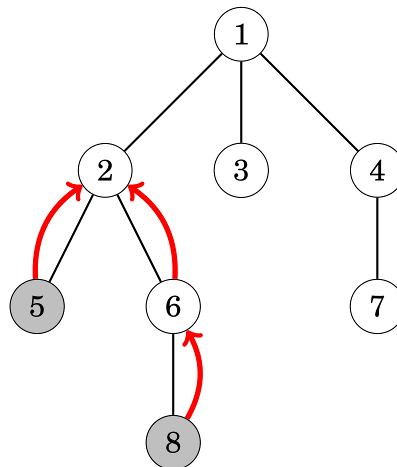
One way to solve the problem is use the fact that we can efficiently find the kth ancestor of any node in the tree. Using this idea, we can first ensure that both

nodes are at the same level in the tree, and then find the smallest value of k where the kth ancestor of both nodes is the same.

As an example, let's find the lowest common ancestor of nodes 5 and 8 in the following tree:



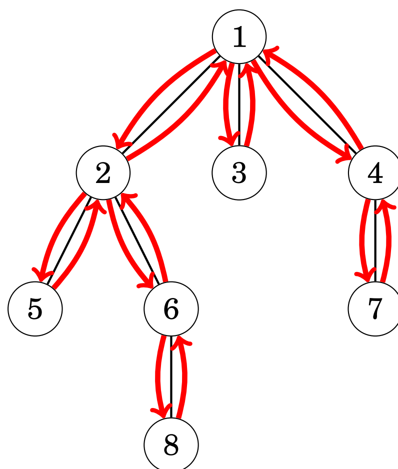
Node 5 is at level 3, while node 8 is at level 4. Thus, we first move one step upwards from node 8 to node 6. After this, it turns out that the parent of both node 5 and node 6 is node 2, so we have found the lowest common ancestor.



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 done in $O(\log n)$ time.

Method 2

Another way to solve the problem is based on a node array. Again, the idea is to traverse the nodes using a depth-first search:



However, we add each node to the node array *always* when the depth-first search visits the node, and not only at the first visit. Thus, a node that has k children appears $k+1$ times in the node array, and there are a total of $2n-1$ nodes in the array.

We store two values in the array: (1) 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 locating 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
			↑											

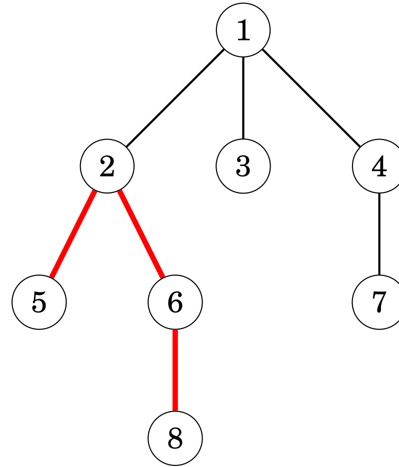
Node 5 is at index 3, node 8 is at index 6, and the node with lowest level between indices $3\ldots6$ is node 2 at index 4 whose level is 2. Thus, the lowest common ancestor of nodes 5 and 8 is node 2.

Using a segment tree, we can find the lowest common ancestor in $O(\log n)$ time. Since the array is static, the time complexity $O(1)$ is also possible, but this is rarely needed. In both cases, preprocessing takes $O(n \log n)$ time.

Distances of nodes

Finally, let's consider a problem where each query asks to find the distance between two nodes in the tree, i.e., the length of the path between them. It turns out that this problem reduces to finding the lowest common ancestor.

First, we choose an arbitrary node for the root of the tree. After this, the distance between nodes a and b is $d(a)+d(b)-2\cdot d(c)$, where c is the lowest common ancestor, and $d(s)$ is 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 goes first upwards from node 5 to node 2, and then downwards 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

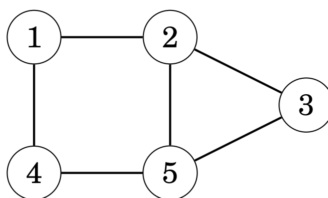
This chapter focuses on two types of paths in a graph:

- 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.

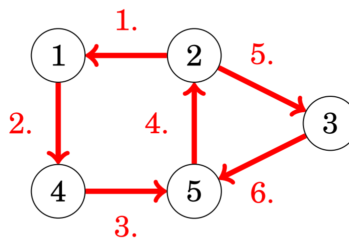
While Eulerian and Hamiltonian paths look like similar concepts at first glance, the computational problems related to them are very different. It turns out that a simple rule based on node degrees determines if a graph contains an Eulerian path, and there is also an efficient algorithm for finding the path. On the contrary, finding a Hamiltonian path is a NP-hard problem and thus no efficient algorithm is known for solving the problem.

19.1 Eulerian path

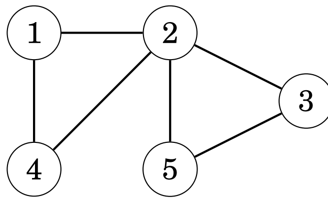
An **Eulerian path** is a path that goes exactly once through each edge in the graph. For example, the graph



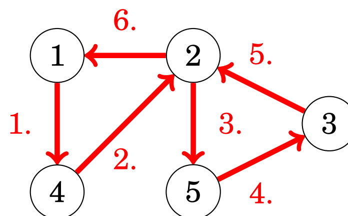
has an Eulerian path from node 2 to node 5:



An **Eulerian circuit** is an Eulerian path that begins and ends at the same node. For example, the graph



has an Eulerian circuit that starts and ends at node 1:



Existence

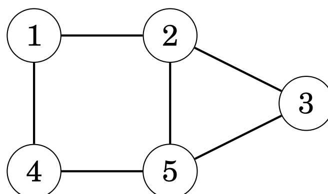
It turns out that the existence of Eulerian paths and circuits depends on the degrees of the nodes in the graph. The degree of a node is the number of its neighbours, i.e., the number of nodes that are connected with a direct edge.

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 is also an Eulerian circuit. In the second case, the odd-degree nodes are the starting and ending nodes of an Eulerian path, and it is not an Eulerian circuit.

For example, in the graph



the degree of nodes 1, 3 and 4 is 2, and the degree of nodes 2 and 5 is 3. Exactly two nodes have an even degree, so there is an Eulerian path between nodes 2 and 5, but the graph doesn't contain an Eulerian circuit.

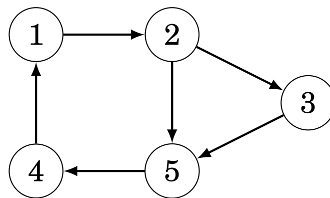
In a directed graph, the situation is a bit more difficult. In this case we should focus on indegree and outdegrees of the nodes in the graph. The indegree of a node is the number of edges that end at the node, and correspondingly, the outdegree is the number of edges that begin at the node.

A directed graph contains an Eulerian path if all the edges belong to the same strongly connected component and

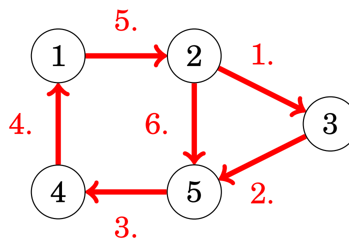
- each node has the same indegree and outdegree *or*
- in one node, indegree is one larger than outdegree, in another node, outdegree is one larger than indegree, and all other nodes have the same indegree and outdegree.

In the first case, each Eulerian path is also an Eulerian circuit, and in the second case, the graph only 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 constructs an Eulerian circuit in an undirected graph. The algorithm assumes that all edges belong to the same connected component, and the degree of each node is even. The algorithm can be implemented in $O(n+m)$ time.

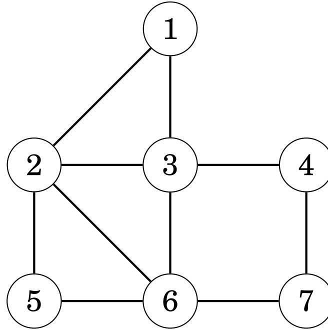
First, the algorithm constructs a circuit that contains some (not necessarily all) of the edges in the graph. After this, the algorithm extends the circuit step by step by adding subcircuits to it. This continues until all edges have been added and the Eulerian circuit is ready.

The algorithm extends the circuit by always choosing a node x that belongs to the circuit but has some edges that are not included in the circuit. The algorith constructs a new path from node x that only contains edges that are not in the circuit. Since the degree of each node is even, sooner or later the path will return to node x which creates a subcircuit.

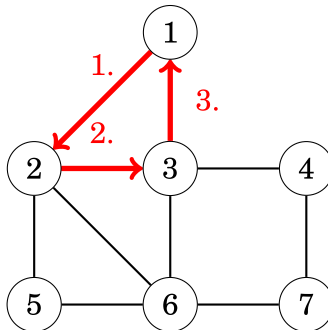
If the graph contains two odd-degree nodes, Hierholzer's algorithm can also be used for constructing an Eulerian path by adding an extra edge between the odd-degree nodes. After this, we can first construct an Eulerian circuit and then remove the extra edge, which produces an Eulerian path in the original graph.

Example

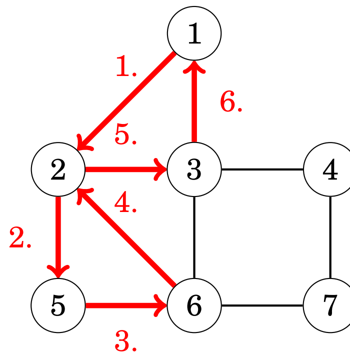
Let's consider the following graph:



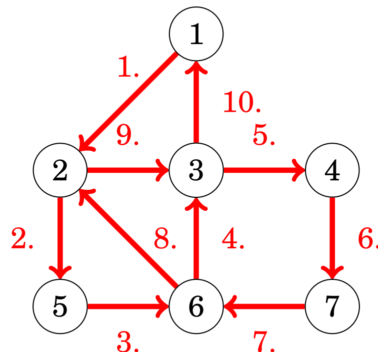
Assume that the algorithm first creates a circuit that begins at node 1. One possible circuit is $1 \to 2 \to 3 \to 1$:



After this, the algorithm adds a subcircuit $2 \to 5 \to 6 \to 2$:



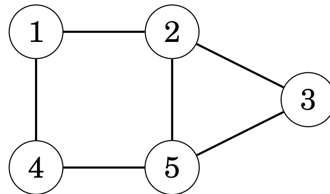
Finally, the algorithm adds a subcircuit $6 \to 3 \to 4 \to 7 \to 6$:



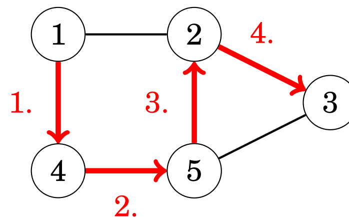
Now all edges are included in the circuit, so we have successfully constructed an Eulerian circuit.

19.2 Hamiltonian path

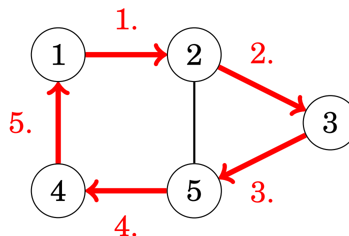
A **Hamiltonian path** is a path that visits each node in the graph exactly once. For example, the graph



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 way is known to check if a graph contains a Hamiltonian path. 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 feature in these theorems and other results is that they guarantee that a Hamiltonian path exists if the graph has *a lot* of edges. This makes sense because the more edges the graph has, the more possibilities we have 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 possibilities how to construct the path. The time complexity of such an algorithm is at least $O(n!)$, because there are $n!$ different ways to form a path from 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.

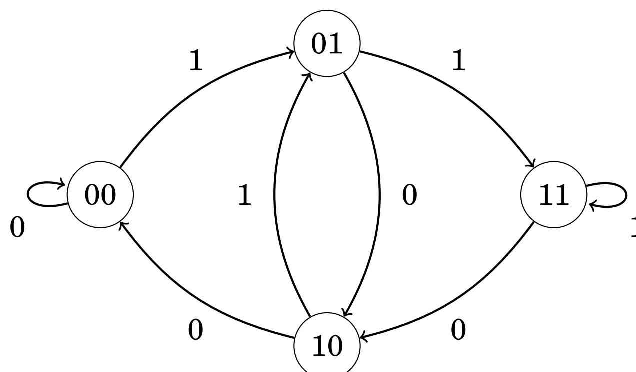
19.3 De Bruijn sequence

A **De Bruijn sequence** is a string that contains every string of length n exactly once as a substring, for a fixed alphabet that consists 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 example case:



An Eulerian path in this graph produces a string that contains all strings of length n. The string contains the characters in the starting node, and all character in the edges. The starting node contains $n-1$ 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 tour

A **knight's tour** is a sequence of moves of a knight on an $n \times n$ chessboard following the rules of chess where 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's a 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 solve the problem is to use backtracking. The search can be made more efficient by using **heuristics** that attempts to guide the knight so that a complete tour will be found quickly.

Warnsdorff's rule

Warnsdorff's rule is a simple and good heuristic for finding a knight's tour. Using the rule, it is possible to efficiently find 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 case there are five possible squares where the knight can move:

1				a
		2		
b				e
	c		d	

In this case, 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 are three moves available.

Chapter 20

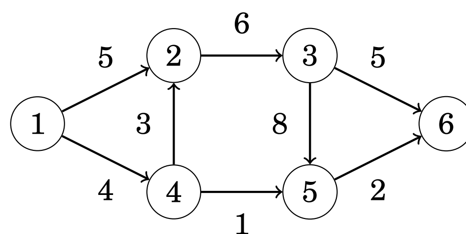
Flows and cuts

In this chapter, we will focus on the following problems in a directed, weighted graph where a starting node and a ending node is given:

- **Finding a maximum flow**: What is the maximum amount of flow we can deliver from the starting node to the ending node?
- **Finding a minimum cut**: What is a minimum-weight set of edges that separates the starting node and the ending node?

It turns out that these problems correspond to each other, and we can solve them simultaneously using the same algorithm.

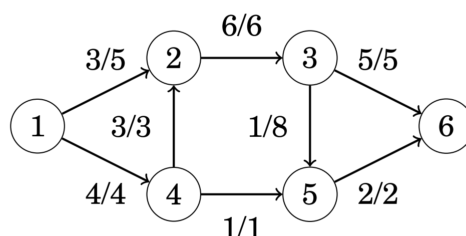
As an example, we will use the following graph where node 1 is the starting node and node 6 is the ending node:



Maximum flow

A **maximum flow** is a flow from the starting node to the ending node whose total amount is as large as possible. The weight of each edge is a capacity that determines the maximum amount of flow that can go through the edge. In all nodes, except for the starting node and the ending node, the amount of incoming and outgoing flow must be the same.

A maximum flow for the example graph is as follows:



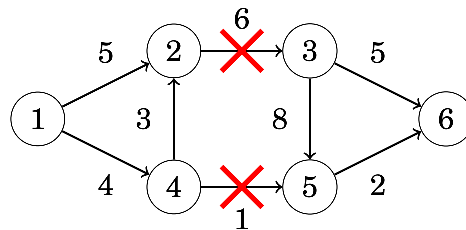
The notation v/k means that amount of the flow through the edge is v and the capacity of the edge is k. For each edge, it is required that $v \le k$. In this graph, the size of a maximum flow is 7 because the outgoing flow from the starting node is $3+4=7$, and the incoming flow to the ending node is $5+2=7$.

Note that in each intermediate node, the incoming flow and the outgoing flow are equally large. For example, in node 2, the incoming flow is $3+3=6$ from nodes 1 and 4, and the outgoing flow is 6 to node 3.

Minimum cut

A **minimum cut** is a set of edges whose removal separates the starting node from the ending node, and whose total weight is as small as possible. A cut divides the graph into two components, one containing the starting node and the other containing the ending node.

A minimum cut for the example graph is as follows:



In this cut, the first component contains nodes $\{1,2,4\}$, and the second component contains nodes $\{3,5,6\}$. The weight of the cut is 7, because it consists of edges $2 \rightarrow 3$ and $4 \rightarrow 5$, and the total weight of the edges is $6+1=7$.

It is not a coincidence that both the size of the maximum flow and the weight of the minimum cut is 7 in the example graph. It turns out that a maximum flow and a minimum cut are *always* of equal size, so the concepts are two sides of the same coin.

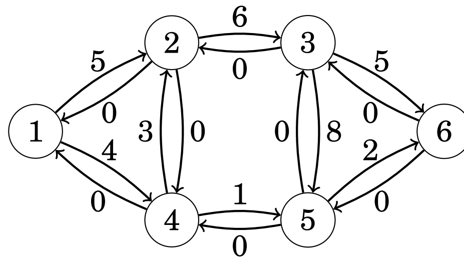
Next we will discuss the Ford–Fulkerson algorithm that can be used for finding a maximum flow and a minimum cut in a graph. The algorithm also helps us to understand *why* they are equally large.

20.1 Ford–Fulkerson algorithm

The **Ford–Fulkerson algorithm** finds a 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 can't extend the flow anymore, it terminates and a maximum flow has been found.

The algorithm uses a special representation for 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. Initially, the weight of each original edge equals the capacity of the edge, and the weight of each reverse edge is zero.

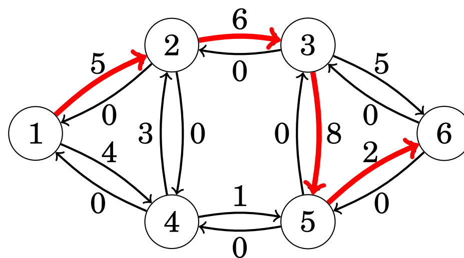
The new representation for the example graph is as follows:



Algoritmin toiminta

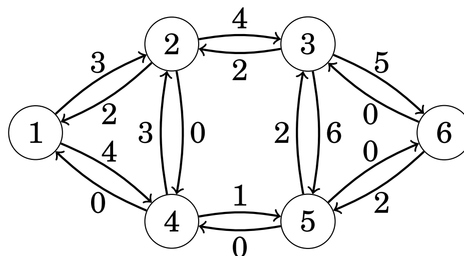
The Ford–Fulkerson algorithm finds at each step a path from the starting node to the ending node where each edge has a positive weight. If there are more than one possible paths, we can choose any of them.

In the example graph, we can choose, say, the following path:



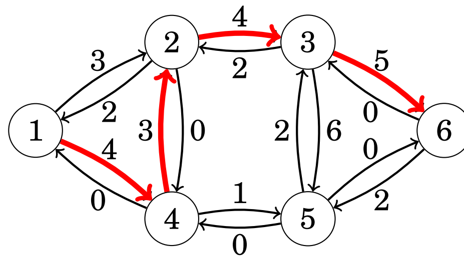
After choosing the path, the flow increases by x units where x is the smallest weight of an edge in the path. In addition, the weight of each edge in the path decreases by x, and the weight of each reverse edge increases by x.

In the above path, the weights of the edges are 5, 6, 8 and 2. The minimum weight is 2, so the flow increases by 2 and the new graph is as follows:



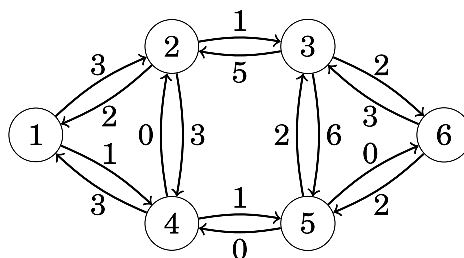
The idea is that increasing the flow decreases the amount of flow that can go through the edges in the future. On the other hand, it is possible to adjust the amount of the flow later using the reverse edges if it turns out that we should route the flow in another way.

The algorithm increases the flow as long as there is a path from the starting node to the ending node through positive edges. In the current example, our next path can be as follows:

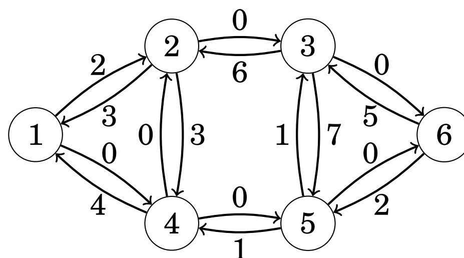


The minimum weight in this path is 3, so the path increases the flow by 3, and the total amount of the flow after processing the path is 5.

The new graph will be as follows:



We still need two more steps before we have reached a 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's not possible to increase the flow anymore, because there is no path from the starting node to the ending node with positive edge weights. Thus, the algorithm terminates and the maximum flow is 7.

Finding paths

The Ford–Fulkerson algorithm doesn't specify how the path that increases the flow should be chosen. In any case, the algorithm will stop sooner or later and produce a 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 the worst case is that each path only increases the flow by 1, and the algorithm becomes slow. Fortunately, we can avoid this by using one of the following algorithms:

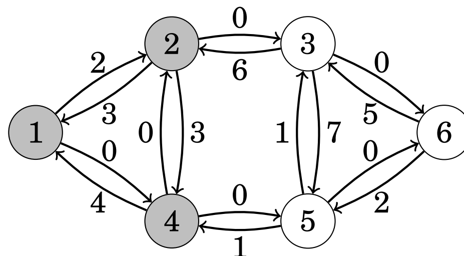
The **Edmonds–Karp algorithm** is an implementation of the Ford–Fulkerson algorithm where each path that increases the flow is chosen so that the number of edges in the path is minimum. This can be done by using breadth-first search instead of depth-first search. It turns out that this guarantees that flow increases quickly, and the time complexity of the algorithm is $O(m^2 n)$.

The **scaling algorithm** uses depth-first search to find paths where the weight of each edge is at least a minimum value. Initially, the minimum value is c, the sum of capacities of the edges that begin at the starting edge. If the algorithm can't find a path, the minimum value is divided by 2, and finally it will be 1. The time complexity of the algorithm is $O(m^2 \log c)$.

In practice, the scaling algorithm is easier to code because we can use depth-first search to find paths. Both algorithms are efficient enough for problems that typically appear in programming contests.

Minimum cut

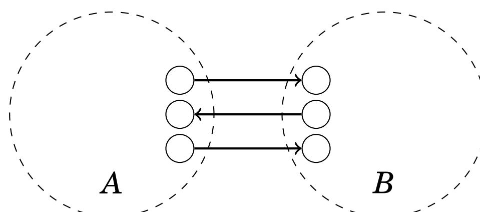
It turns out that once the Ford–Fulkerson algorithm has found a maximum flow, it has also produced a minimum cut. Let A be the set of nodes that can be reached from the starting node using positive edges. In the example graph, A contains nodes 1, 2 and 4:



Now the minimum cut consists of the edges in the original graph that begin at a node in A and end at a node outside A, and whose capacity is fully used in the maximum flow. In the above graph, such edges are $2 \to 3$ and $4 \to 5$, 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 for this is that a graph never contains 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's consider any cut in the graph where the starting node belongs to set A, the ending node belongs to set B and there are edges between the sets:



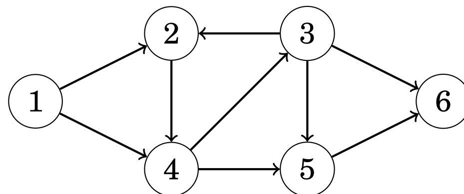
The weight of the cut is the sum of those edges that go from set A to set B. This is an upper bound for the amount of flow in the graph, because the flow has to proceed from set A to set B. Thus, a maximum flow is smaller than or equal to any cut in the graph.

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.

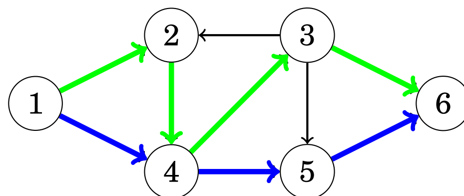
20.2 Parallel paths

As a first application for flows, we consider a problem where the task is to form as many parallel paths as possible from the starting node of the graph to the ending node. It is required that no edge appears in more than one path.

For example, in the graph

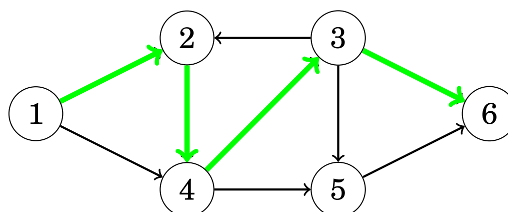


we can form two parallel paths from node 1 to node 6. This can be done by choosing paths $1 \to 2 \to 4 \to 3 \to 6$ and $1 \to 4 \to 5 \to 6$:



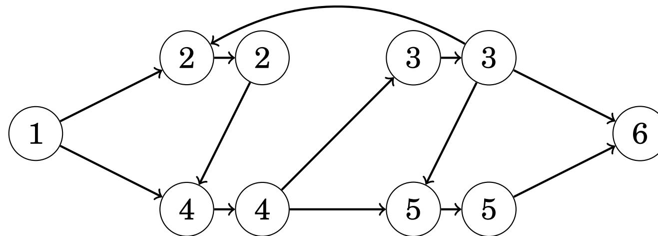
It turns out that the maximum number of parallel paths equals the maximum flow in the graph when the weight of each edge is 1. After the maximum flow has been constructed, the parallel paths can be found greedily by finding paths from the starting node to the ending node.

Let's then consider a variation for the problem where each node (except for the starting and ending nodes) can appear in at most one path. After this restriction, we can construct only one path in the above graph, because node 4 can't appear in more than one path:

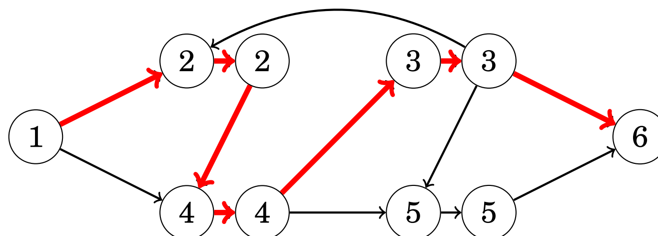


A standard way to restrict the flow through a node is to divide the node into two parts. All incoming edges are connected to the first part, and all outgoing edges are connected to the second part. In addition, there is an edge from the first part to the second part.

In the current example, the graph becomes as follows:



The maximum flow for the graph is as follows:



This means that it is possible to form exactly one path from the starting node to the ending node when a node can't appear in more than one path.

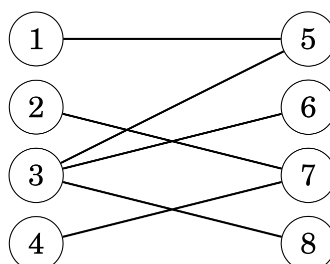
20.3 Maximum matching

A **maximum matching** is the largest possible set of pairs of nodes in a graph such that there is an edge between each pair of nodes, and each node belongs to at most one pair.

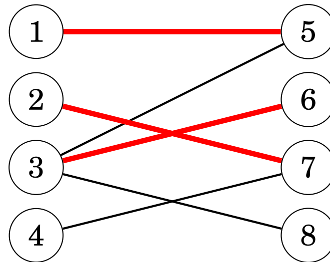
There is a polynomial algorithm for finding a maximum matching in a general graph, but it is very complex. For this reason, we will restrict ourselves to the case where the graph is bipartite. In this case we can easily find the maximum matching using a maximum flow algorithm.

Finding a maximum matching

A bipartite graph can be always presented so that it consists of left-side and right-side nodes, and all edges in the graph go between left and right sides. As an example, consider the following graph:

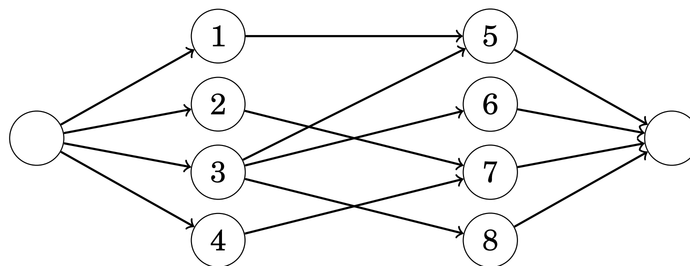


In this graph, the size of a maximum matching is 3:



A maximum matching in a bipartite graph corresponds to a maximum flow in an extended graph that contains a starting node, an ending node and all the nodes of the original graph. There is an edge from the starting node to each left-side node, and an edge from each right-side node to the ending node. The capacity of each edge is 1.

In the example graph, the result is as follows:



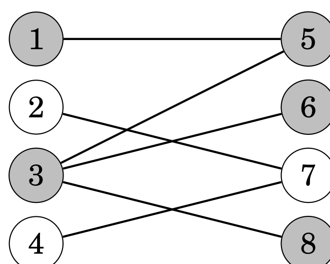
The size of a maximum flow in this graph equals the size of a maximum matching in the original graph, because each path from the starting node to the ending node adds one edge to the matching. In this graph, the maximum flow is 3, so the maximum matching is also 3.

Hall's theorem

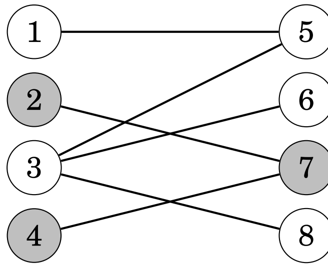
Hall's theorem describes when a bipartite graph has a matching that contains all nodes in one side of the graph. If both sides contain the same number of nodes, Hall's theorem tells us if it's possible to construct a **perfect matching** where all nodes are paired with each other.

Assume that we want to construct a matching that contains all left-side nodes. Let X be a set of left-side nodes, and let $f(X)$ be the set of their neighbors. According to Hall's theorem, a such matching exists exactly when for each X, the condition $|X| \le |f(X)|$ holds.

Let's 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 doesn't hold. This means that it's not possible to form a perfect matching in the graph. This result is not surprising, because we already knew that the maximum matching in the graph is 3 and not 4.

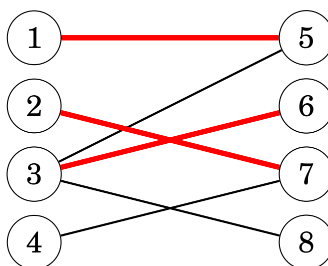
If the condition of Hall's theorem doesn't hold, the set X provides an explanation why we can't form a matching. Since X contains more nodes than $f(X)$, there is no pair for all nodes in X. For example, in the above graph, both nodes 2 and 4 should be connected to node 7 which is not possible.

Kőnig's theorem

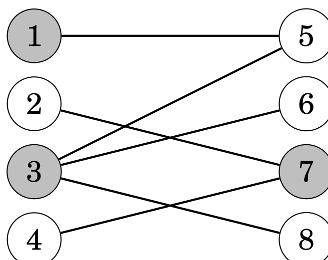
Kőnig's theorem provides an efficient way to construct a **minimum node cover** for a bipartite graph. This is a minimum set of nodes such that each edge in the graph is connected to at least one node in the set.

In a general graph, finding a minimum node cover is a NP-hard problem. However, in a bipartite graph, the size of a maximum matching and a minimum node cover is always the same, according to Kőnig's theorem. Thus, we can efficiently find a minimum node cover using a maximum flow algorithm.

Let's consider the following graph with a maximum matching of size 3:

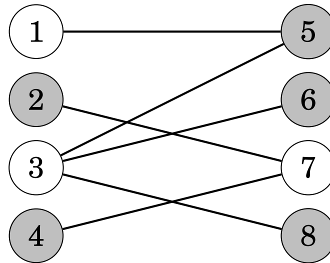


Using Kőnig's theorem, we know that the size of a minimum node cover is also 3. It can be constructed as follows:



For each edge in the maximum matching, exactly one of its end nodes belongs to the minimum node cover.

The set of all nodes that do *not* belong to a minimum node cover forms a **maximum independent set**. This is the largest possible set of nodes where there is no edge between any two nodes in the graph. 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:



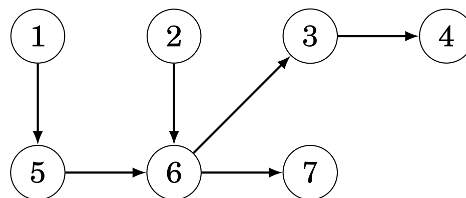
20.4 Path covers

A **path cover** is a set of paths in a graph that is chosen so that each node in the graph belongs to at least one path. It turns out that we can reduce the problem of finding a minimum path cover in a directed, acyclic graph into a maximum flow problem.

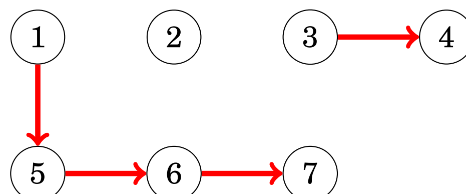
There are two variations for the problem: In a **node-disjoint cover**, every node appears in exactly one path, and in a **general cover**, a node may appear in more than one path. In both cases, the minimum path cover can be found using a similar idea.

Node-disjoint cover

As an example, consider the following graph:



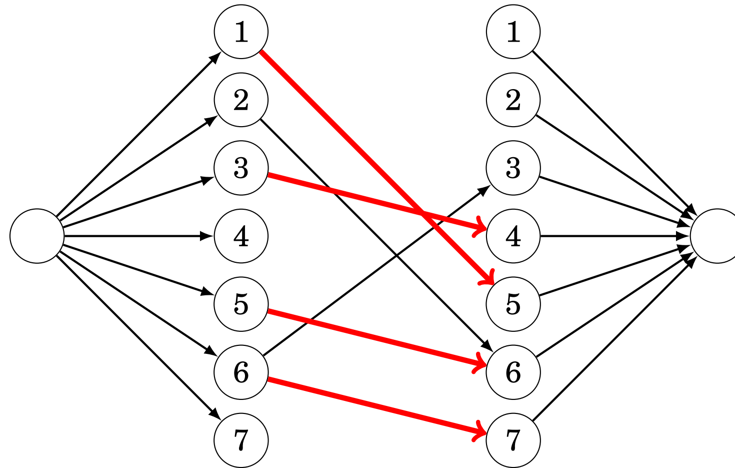
In this case, the minimum node-disjoint path cover consists of three paths. For example, we can choose the following paths:



Note that one of the paths only contains node 2, so it is possible that a path doesn't contain any edges.

Finding a path cover can be interpreted as finding a maximum matching in a graph where each node in 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 such an edge in the original graph. The idea is that the matching determines which edges belong to paths in the original graph.

The matching in the example case is as follows:

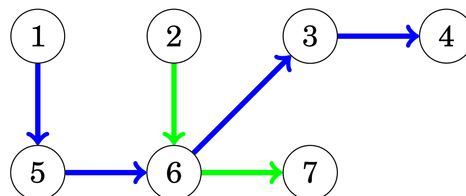


In this case, the maximum matching consists of four edges that corresponds to edges $1 \to 5$, $3 \to 4$, $5 \to 6$ and $6 \to 7$ in the original graph. Thus, a minimum node-disjoint path cover consists of paths that contain these edges.

The size of a minimum path cover is $n - c$ where n is the number of nodes in the graph, and c is the number of edges in the maximum matching. For example, in the above graph the size of the minimum path cover is $7 - 4 = 3$.

General cover

In a general path cover, a node can belong to more than one path which may decrease the number of paths needed. In the example graph, the minimum general path cover consists of two paths as follows:

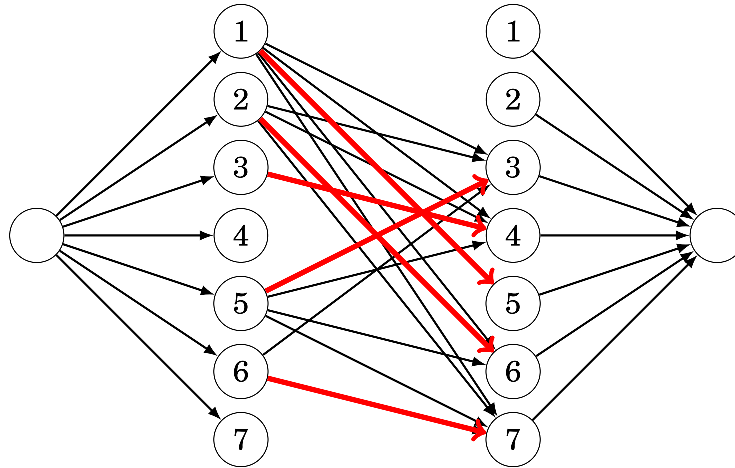


In this graph, a minimum general path cover contains 2 paths, while a minimum node-disjoint path cover contains 3 paths. The difference is that in the general path cover, node 6 appears in two paths.

A minimum general path cover can be found almost like a minimum node-disjoint path cover. It suffices to extend the matching graph so that there is an

edge $a \to b$ always when there is a path from node a to node b in the original graph (possibly through several edges).

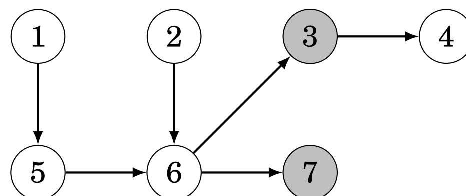
The matching graph for the example case looks as follows:



Dilworth's theorem

Dilworth's theorem states that the size of a minimum general path cover in a directed, acyclic graph equals the maximum size of an **antichain**, i.e., a set of nodes such that there is no path from any node to another node.

For example, in the example graph, the minimum general path cover contains two paths, so the largest antichain contains two nodes. We can construct such an antichain by choosing nodes 3 and 7:



There is no path from node 3 to node 7, and no path from node 7 to node 3, so nodes 3 and 7 form an antichain. On the other hand, if we choose any three nodes in the graph, there is certainly a path from one node to another node.

Part III

Advanced topics

Index

2SAT problem, 152
2SUM problem, 76
3SAT problem, 154
3SUM-ongelma, 77

acyclic graph, 105
adjacency list, 106
adjacency matrix, 108
amortized analysis, 75
and operation, 94
antichain, 182
arithmetic sum, 10

backtracking, 48
Bellman–Ford algorithm, 117
binary code, 60
binary indexed tree, 84
binary search, 29
binary tree, 131
bipartite graph, 106, 116
bit representation, 93
bit shift, 95
bitset, 39
`bitset`, 39
breadth-first search, 113
bubble sort, 23

child, 127
codeword, 60
coloring, 106
comparison function, 29
comparison operator, 28
complement, 11
complete graph, 105
complexity classes, 18
compomnent, 104
component graph, 149
conjuction, 12
connected graph, 104, 115
constant factor, 19
constant-time algorithm, 18
counting sort, 27
cubic algorithm, 18
cut, 172
cycle, 105, 115, 141, 147
cycle detection, 147

data compression, 60
data structure, 33
De Bruijn sequence, 168
degree, 105
depth-first search, 111
deque, 40
`deque`, 40
diameter, 129
difference, 11
Dijkstra's algorithm, 120, 144
Dilworth's theorem, 182
Dirac's theorem, 167
directed graph, 104
disjunction, 12
dynamic array, 33
dynamic programming, 63

edge, 103
edge list, 109
edit distance, 71
Edmonds–Karp algorithm, 174
equivalence, 12
Eulerian circuit, 163
Eulerian path, 163

factorial, 13
Fenwick tree, 84
Fibonacci number, 13
floating point number, 7
flow, 171
Floyd's algorithm, 147
Floyd–Warshall algorithm, 123
Ford–Fulkerson algorithm, 172

functional graph, 146

geometric sum, 10
graph, 103
greedy algorithm, 55

hakemisto, 36
Hall's theorem, 178
Hamiltonian circuit, 167
Hamiltonian path, 167
harmonic sum, 11
heap, 41
heuristic, 169
Hierholzer's algorithm, 165
Huffman coding, 61

implication, 12
in-order, 132
indegree, 105
independent set, 180
index compression, 91
input and output, 4
integer, 6
intersection, 11
inversion, 24
iterator, 37

Kőnig's theorem, 179
knapsack, 70
knight's tour, 169
Kosaraju's algorithm, 150
Kruskal's algorithm, 134

leaf, 127
Levenshtein distance, 71
linear algorithm, 18
logarithm, 14
logarithmic algorithm, 18
logic, 12
longest increasing subsequence, 68
lowest common ancestor, 159

macro, 9
`map`, 36
matching, 177
maximum flow, 171
maximum independent set, 180
maximum matching, 177
maximum query, 81
maximum spanning tree, 134
maximum subarray sum, 19
meet in the middle, 52
memoization, 65
merge sort, 25
minimum cut, 172, 175
minimum node cover, 179
minimum query, 81
minimum spanning tree, 133
modular arithmetic, 6
`multiset`, 36

natural logarithm, 14
nearest smaller elements, 77
negation, 12
negative cycle, 119
neighbor, 105
`next_permutation`, 47
node, 103
node array, 156
node cover, 179
not operation, 95
NP-hard problem, 18

or operation, 94
Ore's theorem, 167
outdegree, 105

`pair`, 28
parent, 127
path, 103
path cover, 180
perfect matching, 178
permutation, 47
polynomial algorithm, 18
post-order, 132
pre-order, 132
predicate, 12
prefix sum array, 82
Prim's algorithm, 139
priority queue, 41
`priority_queue`, 41
programming language, 3

quadratic algorithm, 18
quantifier, 12
queen problem, 48

queue, 41
queue, 41

random_shuffle, 37
range query, 81
regular graph, 105
remainder, 6
reverse, 37
root, 127
rooted tree, 127

scaling algorithm, 175
segment tree, 86
set, 11, 35
set, 35
set theory, 11
shortest path, 117
simple graph, 106
sliding window, 79
sliding window minimum, 79
sort, 27, 37
sorting, 23
spanning tree, 133
SPFA algorithm, 120
stack, 40
stack, 40
string, 34
string, 34
strongly connected component, 149
strongly connected graph, 149
subset, 11, 45
subtree, 127
successor graph, 146
sum query, 81

time complexity, 15
topological sorting, 141
tree, 104, 127
tree query, 155
tuple, 28
typedef, 8
two pointers method, 75

union, 11
union-find structure, 137
universal set, 11
unordered_map, 36
unordered_multiset, 36
unordered_set, 35

vector, 33
vector, 33

Warnsdorff's rule, 169
weighted graph, 105

xor operation, 94