

Competitive Programmer's Handbook

Antti Laaksonen

January 3, 2017

Contents

Preface	v
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	Tilings	73
8	Amortized analysis	75
8.1	Two pointers method	75
8.2	Nearest smaller elements	77
8.3	Sliding window minimum	79
II	Graph algorithms	81
III	Advanced topics	83

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 Java¹. 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`, because `endl` always causes a flush operation.

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

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

The following code prints two integers:

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

Sometimes the program should read a whole line from the input, possibly with spaces. This can be accomplished 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 $-2^{31} \dots 2^{31} - 1$, i.e., about $-2 \cdot 10^9 \dots 2 \cdot 10^9$. If the type `int` is not enough, the 64-bit type `long long` can be used, with value range $-2^{63} \dots 2^{63} - 1$, i.e., about $-9 \cdot 10^{18} \dots 9 \cdot 10^{18}$.

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 $-2^{127} \dots 2^{127} - 1$, i.e., $-10^{38} \dots 10^{38}$. 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 $0 \dots m - 1$. 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); // 0.999999999999999988898
```

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

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

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

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

and

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

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

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

By multiplying both sides by x , we get

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

and solving the equation

$$xS - S = bx - a.$$

yields the formula.

A special case of a geometric sum is the formula

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

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

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

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:

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

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 $P = \{1, 2, \dots, 10\}$, 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

$\emptyset, \{2\}, \{4\}, \{7\}, \{2, 4\}, \{2, 7\}, \{4, 7\}$ 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 $\mathbb{N} = \{0, 1, 2, \dots\}$ or $\mathbb{N} = \{1, 2, 3, \dots\}$.

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 $\min(x_1, x_2, \dots, x_n)$ and $\max(x_1, x_2, \dots, x_n)$ return the smallest and the largest of values x_1, x_2, \dots, x_n . For example,

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

The **factorial** $n!$ is defined

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

or recursively

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

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

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

The first Fibonacci numbers are

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

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 $O(\dots)$ 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
...	...
$g(1)$	2^{n-1}

Based on this, the time complexity is

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

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 through all permutations of the input elements. For example, the permutations of $\{1, 2, 3\}$ are $(1, 2, 3)$, $(1, 3, 2)$, $(2, 1, 3)$, $(2, 3, 1)$, $(3, 1, 2)$ and $(3, 2, 1)$.

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

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

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
$n \leq 10^{18}$	$O(1)$ tai $O(\log n)$
$n \leq 10^{12}$	$O(\sqrt{n})$
$n \leq 10^6$	$O(n)$ tai $O(n \log n)$
$n \leq 5000$	$O(n^2)$
$n \leq 500$	$O(n^3)$
$n \leq 25$	$O(2^n)$
$n \leq 10$	$O(n!)$

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, \dots, 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 $1 \dots n$. 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.

Consider 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 $t[1], t[2], \dots, t[n]$:

```
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 $(\tau[a], \tau[b])$ in the array such that $a < b$ and $\tau[a] > \tau[b]$, 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:

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

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

Then, the subarrays will be sorted recursively as follows:

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

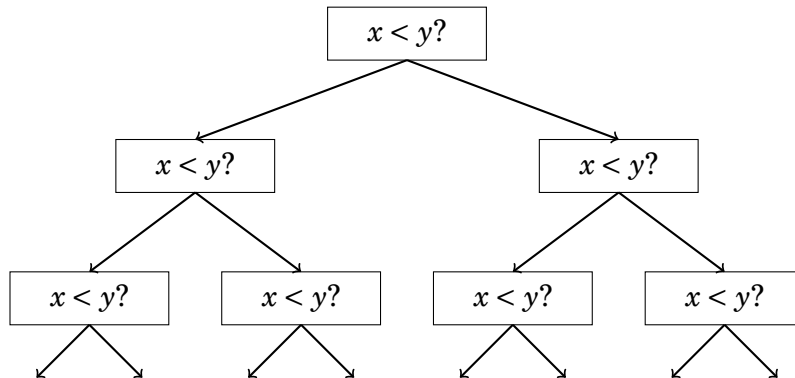
Finally, the algorithm merges the sorted subarrays and creates the final sorted array:

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

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

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 is 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	\dots	$k-1$	k	$k+1$	\dots
<code>ok(x)</code>	false	false	\dots	false	true	true	\dots

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 $x \geq k$.

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

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

Note that unlike in the 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 through 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 $0, 1, \dots, n-1$, 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 "aybaltu" with value 0 is added to the map.

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

The function `count` determines if a key exists in the map:

```
if (m.count("aybabbtu")) {  
    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 because 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

Complete 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.

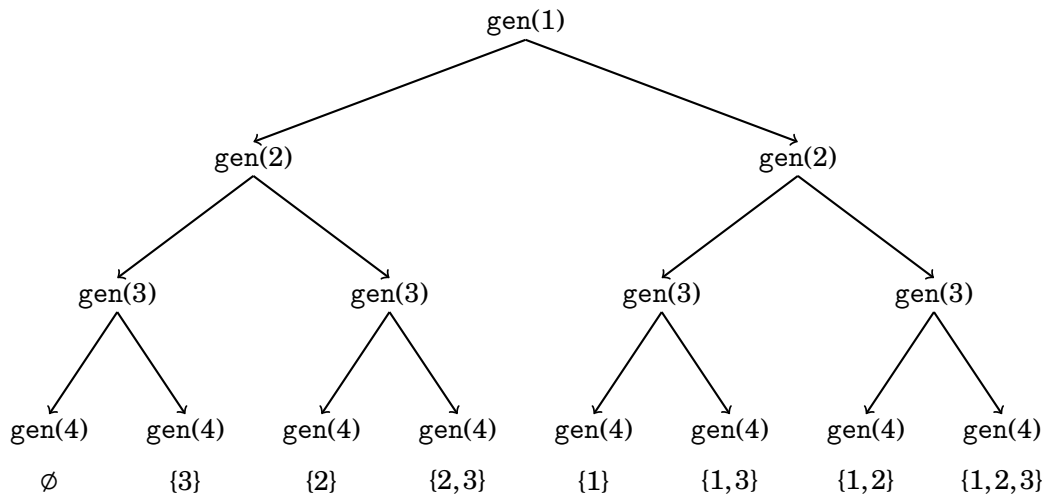
Method 1

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 $\{1, 2, \dots, n\}$. 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 $0 \dots 2^n - 1$. 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 k th 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 $\{1, 2, \dots, n\}$. 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 $p[k] = 0$, element k is not included, and if $p[k] = 1$, element k is included. If the size of the vector equals the size of the set, a permutation has been generated.

Method 2

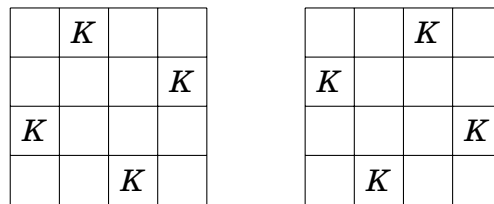
Another method is to begin from permutation $\{1, 2, \dots, n\}$ and at each step generate the next permutation in increasing order. The C++ standard library contains the function `next_permutation` that can be used for this. The following code generates the permutations of the set $\{1, 2, \dots, n\}$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

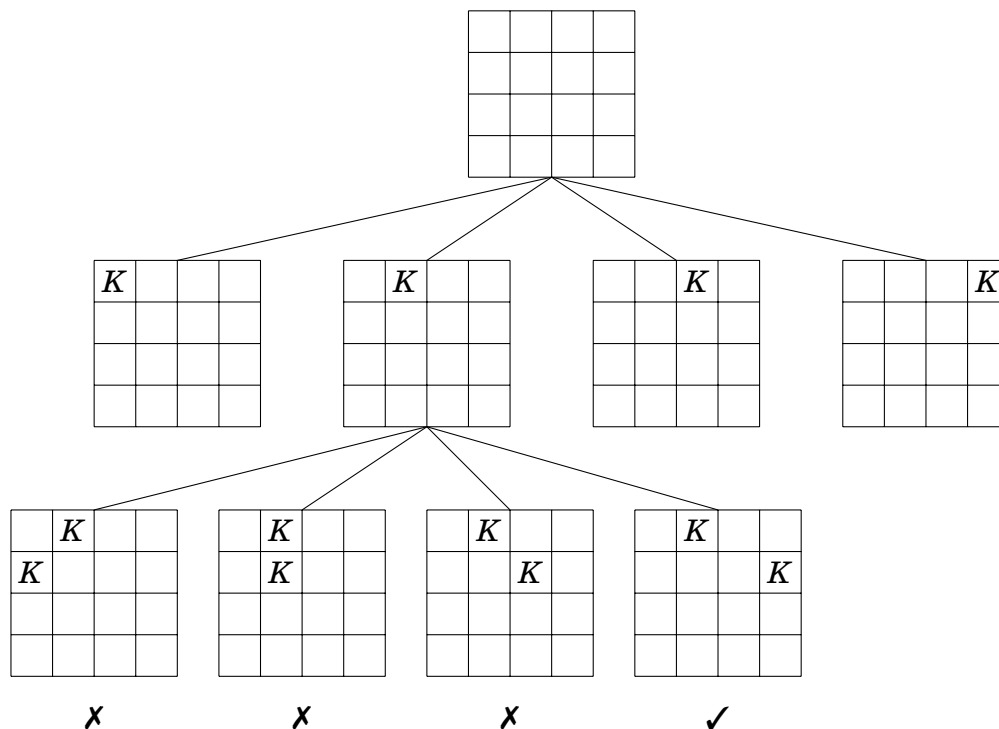
A **backtracking** algorithm begins from an empty solution and extends the solution step by step. At each step, the search branches to all possible directions how the solution can be extended. After processing one branch, the search continues to other possible directions.

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 $0 \leq y < n$. 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`

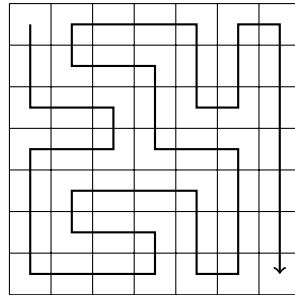
Using the presented backtracking algorithm, we can calculate that, for example, there are 92 ways to place 8 queens to an 8×8 chessboard. When n increases, the search quickly becomes slow because the number of the solutions increases exponentially. For example, calculating the ways to place 16 queens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

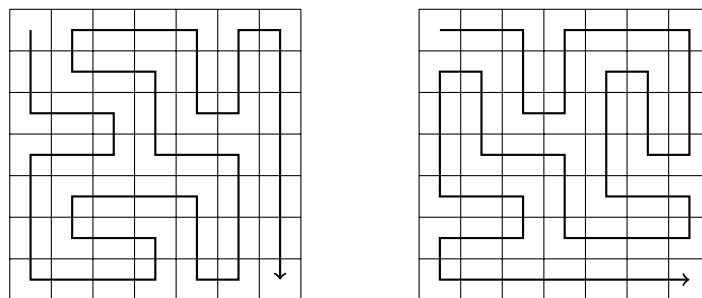
Basic algorithm

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:

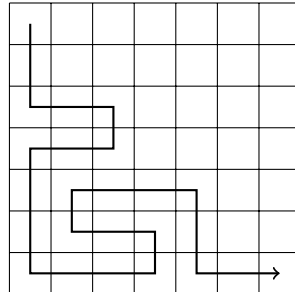


Thus, we can decide that the first step in the solution is always downward, and finally multiply the number of the solutions by two.

- 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:

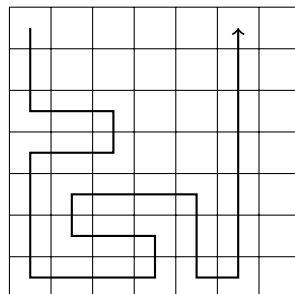


Using this observation, we can terminate the search branch immediately if we reach the lower-right square too early.

- 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:



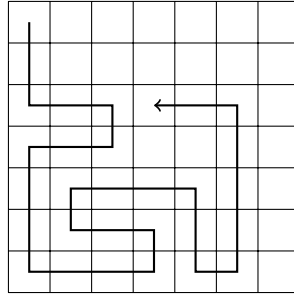
Now it will not be possible to visit every square, so we can terminate the search branch. This optimization is very useful:

- 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 $\{c_1, c_2, \dots, c_k\}$, 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 $3 + 3$.

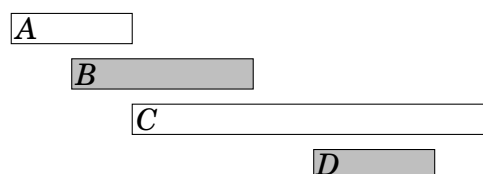
We don't know if the general coin problem can be solved using any greedy algorithm. However, we will revisit the problem in the next chapter because the general problem can be solved using a dynamic programming algorithm that always gives the correct answer.

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
<i>A</i>	1	3
<i>B</i>	2	5
<i>C</i>	3	9
<i>D</i>	6	8

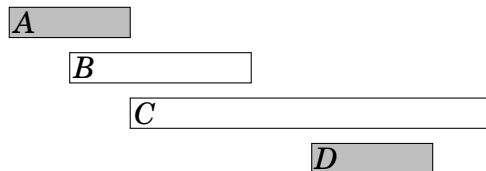
In this case the maximum number of events is two. For example, we can join events *B* and *D* as follows:



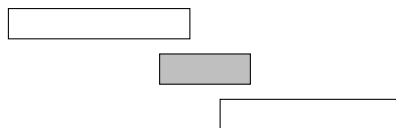
It is possible to invent several greedy algorithms for the problem, but which of them works in every case?

Algorithm 1

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



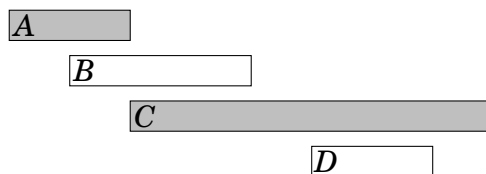
However, choosing short events is not always a correct strategy but the algorithm fails, for example, in the following case:



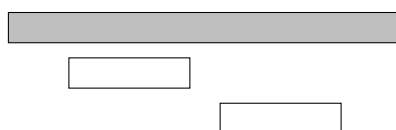
If we select the short event, we can only select one event. However, it would be possible to select both the long events.

Algorithm 2

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



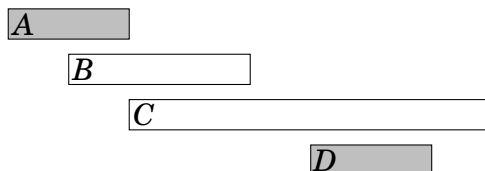
However, we can find a counterexample for this algorithm, too. For example, in the following case, the algorithm selects only one event:



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

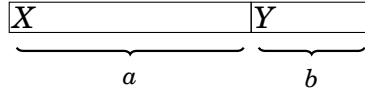
task	duration	deadline
<i>A</i>	4	2
<i>B</i>	3	5
<i>C</i>	2	7
<i>D</i>	4	5

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

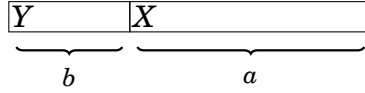


In this solution, *C* yields 5 points, *B* yields 0 points, *A* yields -7 points and *D* yields -8 points, so the total score is -10 .

Surprisingly, the optimal solution for the problem doesn't depend on the deadlines at all, but a correct greedy strategy is to simply perform the tasks *sorted by their durations* in increasing order. The reason for this is that if we ever perform two 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, \dots, a_n and our task is to find a value x such that the sum

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

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

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

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 than 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

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

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, \dots, 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

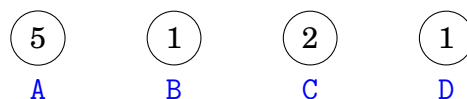
Using this code, it would not be possible to know if the compressed string 1011 means the string AB or the string C.

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 corresponds to bit 0, and a move to the right corresponds to bit 1.

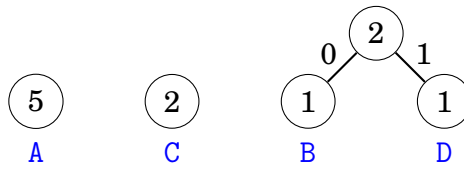
Initially, each character of the string is represented by a node whose weight is the number of times the character appears in the string. Then at each step two nodes with minimum weights are selected and they are combined by creating a new node whose weight is the sum of the weights of the original nodes. 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:

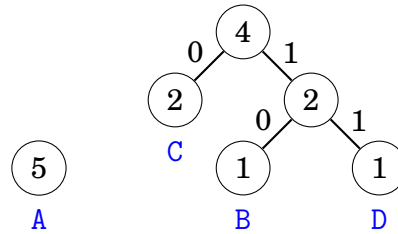


The node that represents character A has weight 5 because character A appears 5 times in the string. The other weights have been calculated in the same way.

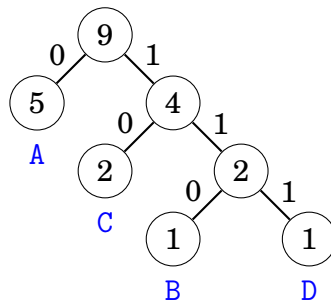
The first step is to combine the nodes that correspond to characters B and D, both with weight 1. The result is:



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:

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

We will first see how dynamic programming can be used 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 $\{c_1, c_2, \dots, c_k\}$ 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 algorithm 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:

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

First, $f(0) = 0$ because no coins are needed for 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 $\{c_1, c_2, \dots, c_k\}$, the recursive formula is

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

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

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

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 $c[1], c[2], \dots, c[k]$, 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 $d[x]$ 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 $d[x]$. 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 $d[x]$.

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

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

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:

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

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

```
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, \dots, 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 $f(1), f(2), \dots, f(n)$.

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

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

When calculating the value $f(k)$, 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 $i = 1, 2, \dots, k - 1$ 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:

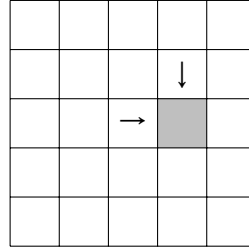
3	7	9	2	7
9	8	3	5	5
1	7	9	8	5
3	8	6	4	10
6	3	9	7	8

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:

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

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

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

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

7.4 Knapsack

Knapsack is a classic problem where we are given n objects with weights p_1, p_2, \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 $1 \dots k$ is selected such that the total weight is u . The solution for the problem is the largest value $f(n, u)$ where $0 \leq u \leq x$. 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 complexity 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 $1 \dots k$ is selected such that the total weight is u . The solution for the problem is the largest value u for which $0 \leq u \leq s$ and $f(n, u) \leq x$ 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. $ABC \rightarrow ABCA$)
- remove a character (e.g. $ABC \rightarrow AC$)

- change a character (e.g. ABC \rightarrow ADC)

For example, the edit distance between LOVE and MOVIE is 2 because we can first perform operation LOVE \rightarrow MOVE (change) and then operation MOVE \rightarrow 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 a th character of x equals the b th 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 character is removed from x
- $f(a-1, b-1)$ means that x and y contain the same character ($c = 0$), or a character in x is transformed into a character in y ($c = 1$)

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

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

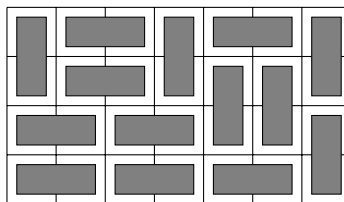
The lower-right corner of the table indicates that the edit distance between LOVE and MOVIE is 2. The table also shows how to construct the shortest sequence of editing operations. In this case the path is as follows:

		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 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:

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

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 programming here because the state of a row is constrained only by 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 $\sqcup \sqsubset \sqsubset \sqcup \sqcap \sqcap \sqcup$ and $\sqsubset \sqsubset \sqsubset \sqcup \sqcup \sqcap$ are compatible, while the rows $\sqcap \sqsubset \sqsubset \sqcap \sqsubset \sqsubset \sqcap$ and $\sqsubset \sqsubset \sqsubset \sqsubset \sqsubset \sqcup$ 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 \sqcap and \square where \square is a combination of characters \sqcap , \square and \sqcap . 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:

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

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.

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




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:

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



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:

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

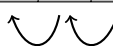
The sliding window minima can be calculated using the same idea that we used for calculating the nearest smaller elements. The idea is to maintain a chain whose first element is the last element in the window, and each element is smaller than the previous element. The last element in the chain is always 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:

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




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

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



After this, the window moves again and the smallest element 1 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.

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




The next new element 1 is smaller than all elements in the chain. Thus, all elements are removed from the chain and it will only contain the element 1:

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

•

Finally the window reaches its last position. The number 2 is added to the chain, but the smallest element inside the window is still 1.

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



Also in this algorithm, each element in the array is added to the chain exactly once and removed from the chain at most once. Thus, the total time complexity of the algorithm is $O(n)$.

Part II

Graph algorithms

Part III

Advanced topics

Index

- 2SUM problem, 76
- 3SUM-ongelma, 77
- amortized analysis, 75
- arithmetic sum, 10
- backtracking, 48
- binary code, 60
- binary search, 29
- bitset, 39
- bitset, 39
- bubble sort, 23
- codeword, 60
- comparison function, 29
- comparison operator, 28
- complement, 11
- complexity classes, 18
- conjunction, 12
- constant factor, 19
- constant-time algorithm, 18
- counting sort, 27
- cubic algorithm, 18
- data compression, 60
- data structure, 33
- deque, 40
- deque, 40
- difference, 11
- disjunction, 12
- dynamic array, 33
- dynamic programming, 63
- edit distance, 71
- equivalence, 12
- factorial, 13
- Fibonacci number, 13
- floating point number, 7
- geometric sum, 10
- greedy algorithm, 55
- hakemisto, 36
- harmonic sum, 11
- heap, 41
- Huffman coding, 61
- implication, 12
- input and output, 4
- integer, 6
- intersection, 11
- inversion, 24
- iterator, 37
- knapsack, 70
- Levenshtein distance, 71
- linear algorithm, 18
- logarithm, 14
- logarithmic algorithm, 18
- logic, 12
- longest increasing subsequence, 68
- macro, 9
- map, 36
- maximum subarray sum, 19
- meet in the middle, 52
- memoization, 65
- merge sort, 25
- modular arithmetic, 6
- multiset, 36
- natural logarithm, 14
- nearest smaller elements, 77
- negation, 12
- next_permutation, 47
- NP-hard problem, 18
- pair, 28
- permutation, 47
- polynomial algorithm, 18

- predicate, 12
- priority queue, 41
- priority_queue, 41
- programming language, 3

- quadratic algorithm, 18
- quantifier, 12
- queen problem, 48
- queue, 41
- queue, 41

- random_shuffle, 37
- remainder, 6
- reverse, 37

- set, 11, 35
- set, 35
- set theory, 11
- sliding window, 79
- sliding window minimum, 79
- sort, 27, 37
- sorting, 23
- stack, 40
- stack, 40
- string, 34
- string, 34
- subset, 11, 45

- time complexity, 15
- tuple, 28
- typedef, 8
- two pointers method, 75

- union, 11
- universal set, 11
- unordered_map, 36
- unordered_multiset, 36
- unordered_set, 35

- vector, 33
- vector, 33