Competitive Programmer's Handbook

Antti Laaksonen

January 1, 2017

Contents

Preface

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

The book is especially intended for high school students who want to learn algorithms and possibly participate in the International Olympiad in Informatics (IOI). The book is also suitable for university students and anybody else interested in competitive programming.

It takes a long time to become a good competitive programmer, but it is also an opportunity to learn a lot. You can be sure that you will learn a great deal about algorithms if you spend time reading the book and solving exercises.

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

Part I

Basic techniques

Chapter 1

Introduction

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

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

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

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

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

1.1 Programming languages

At the moment, the most popular programming languages in contests are C++, Python and Java. For example, in Google Code Jam 2016, among the best 3,000 participants, 73 % used C++, 15 % used Python and 10 % used Java[1]. Some participants also used several languages.

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

On the other hand, it is good to master several languages and know the benefits of them. For example, if big integers are needed in the problem, Python

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

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

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

C++ template

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

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

using namespace std;

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

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

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

The code can be compiled using the following command:

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

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

1.2 Input and output

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

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

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

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

```
123 456 apina
```

```
123   456
apina
```

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

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

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

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

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

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

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

The following code prints two integers:

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

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

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

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

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

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

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

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

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

1.3 Handling numbers

Integers

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

The following code defines a `long long` variable:

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

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

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

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

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

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

Modular arithmetic

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

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

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

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

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

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

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

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

Usually, the answer should be always given so that the remainder is between $0 \ldots m-1$. However, in C++ and other languages, the remainder of a negative number can be negative. An easy way to make sure that this will not happen is to first calculate the remainder as usual and then add m if the result is negative:

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

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

Floating point numbers

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

The required precision of the answer is usually given. The easiest way is to use the `printf` function that can be given the number of decimal places. For example, the following code prints the value of x with 9 decimal places:

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

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

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

Because of a rounding error, the value of x is a bit less than 1, while the correct value would be 1.

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

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

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

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

1.4 Shortening code

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

Type names

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

```
typedef long long ll;
```

After this, the code

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

can be shortened as follows:

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

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

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

Macros

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

For example, we can define the following macros:

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

After this, the code

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

can be shortened as follows:

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

It is also possible to define a macro with parameters which makes it possible to shorten loops and other structures in the code. For example, we can define the following macro:

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

After this, the code

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

can be shortened as follows:

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

1.5 Mathematics

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

Sum formulas

Each sum of the form

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

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

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

and

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

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

$$3 + 7 + 11 + 15$$

is an arithmetic sum with constant 4. An arithmetic sum can be calculated using the formula

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

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

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

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

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

$$3 + 6 + 12 + 24$$

is a geometric sum with constant 2. A geometric sum can be calculated using the formula

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

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

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

This formula can be derived as follows. Let

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

By multiplying both sides by x, we get

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

and solving the equation

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

yields the formula.

A special case of a geometric sum is the formula

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

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

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

An upper bound for the harmonic sum is $\log_2(n) + 1$. The reason for this is that we can change each term $1/k$ so that k becomes a power of two that doesn't exceed k. For example, when $n = 6$, we can estimate the sum as follows:

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

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 $P = \{1, 2, \ldots, 10\}$, then $\bar{A} = \{3, 4, 6, 8, 9, 10\}$.
- The **difference** $A \setminus B = A \cap \bar{B}$ consists of elements that are in A but not in B. Note that B can contain elements that are not in A. For example, if $A = \{2, 3, 7, 8\}$ and $B = \{3, 5, 8\}$, then $A \setminus B = \{2, 7\}$.

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

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

Often used sets are

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

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

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

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

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

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

contains all even integers.

Logic

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

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

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

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

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

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

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

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

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

means that if a number x is larger than 2 and not a prime number, there are numbers a and b that are larger than 1 and whose product is x. This proposition is true in the set of integers.

Functions

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

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

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

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

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

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

or recursively

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

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

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

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

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

Logarithm

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

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

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

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

The logarithm of a product is

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

and consequently,

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

In addition, the logarithm of a quotient is

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

Another useful formula is

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

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

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

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

Chapter 2

Time complexity

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

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

2.1 Calculation rules

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

Loops

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

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

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

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

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

Order of magnitude

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

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

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

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

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

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

Phases

If the code consists of consecutive phases, the total time complexity is the largest time complexity of a single phase. The reason for this is that the slowest phase is usually the bottleneck of the code and the other phases are not important.

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

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

Several variables

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

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

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

Recursion

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

For example, consider the following function:

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

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

As another example, consider the following function:

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

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

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

Based on this, the time complexity is

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

2.2 Complexity classes

Typical complexity classes are:

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

$O(\log n)$ A **logarithmic** algorithm often halves the input size at each step. The reason for this is that the logarithm $\log_2 n$ equals the number of times n must be divided by 2 to produce 1.

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

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

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

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

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

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

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

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

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

2.3 Estimating efficiency

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

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

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

input size (n)	desired time complexity
$n \leq 10^{18}$	$O(1)$ 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.

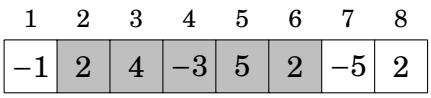
2.4 Maximum subarray sum

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

1	2	3	4	5	6	7	8
−1	2	4	−3	5	2	−5	2

the following subarray produces the maximum sum 10:



Solution 1

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

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

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

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

Solution 2

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

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

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

Solution 3

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

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

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

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

The following code implements the solution:

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

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

Efficiency comparison

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

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

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

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

the algorithms. The $O(n^3)$ time solution 1 becomes slower when $n = 10^3$, and the $O(n^2)$ time solution 2 becomes slower when $n = 10^4$. Only the $O(n)$ time solution 3 solves even the largest inputs instantly.

Chapter 3

Sorting

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

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

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

3.1 Sorting theory

The basic problem in sorting is as follows:

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

For example, the array

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

will be as follows after sorting:

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

$O(n^2)$ algorithms

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

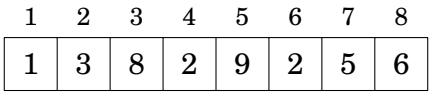
for sorting is **bubble sort** where the elements ”bubble” forward in the array according to their values.

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

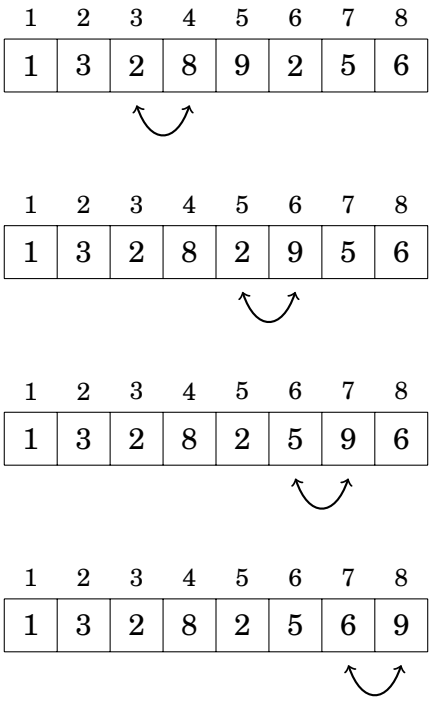
```
for (int i = 1; i <= n-1; i++) {
    for (int j = 1; j <= n-i; j++) {
        if (t[j] > t[j+1]) swap(t[j],t[j+1]);
    }
}
```

After the first round of the algorithm, the largest element is in the correct place, after the second round the second largest element is in the correct place, etc. Thus, after $n-1$ rounds, all elements will be sorted.

For example, in the array



the first round of bubble sort swaps elements as follows:



Inversions

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

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

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

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

the inversions are $(6,3)$, $(6,5)$ and $(9,8)$. The number of inversions indicates how sorted the array is. An array is completely sorted when there are no inversions. On the other hand, if the array elements are in reverse order, the number of inversions is maximum:

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

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

$O(n \log n)$ algorithms

It is possible to sort an array efficiently in $O(n \log n)$ time using an algorithm that is not limited to swapping successive elements. One such algorithm is **mergesort** that sorts an array recursively by dividing it into smaller subarrays.

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

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

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

For example, consider sorting the following array:

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

The array will be divided into two subarrays as follows:

1	2	3	4
1	3	6	2

5	6	7	8
8	2	5	9

Then, the subarrays will be sorted recursively as follows:

1	2	3	4
1	2	3	6

5	6	7	8
2	5	8	9

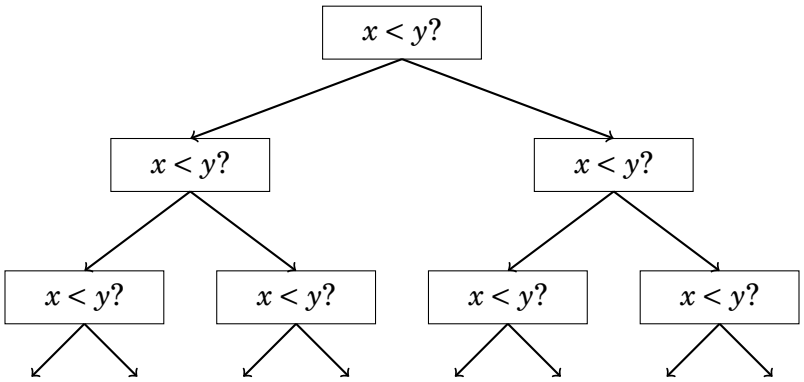
Finally, the algorithm merges the sorted subarrays and creates the final sorted array:

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

Sorting lower bound

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

The lower bound for the time complexity can be proved by examining the sorting as a process where each comparison of two elements gives more information about the contents of the array. The process creates the following tree:



Here "$x < y$?" means that some elements x and y are compared. If $x < y$, the process continues to the left, and otherwise to the right. The results of the process are the possible ways to order the array, a total of $n!$ ways. For this reason, the height of the tree must be at least

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

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

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

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

Counting sort

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

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

For example, the array

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

produces the following bookkeeping array:

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

For example, the value of element 3 in the bookkeeping array is 2, because the element 3 appears two times in the original array (indices 2 and 6).

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

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

3.2 Sorting in C++

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

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

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

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

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

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

A regular array can be sorted as follows:

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

The following code sorts the string `s`:

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

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

Comparison operator

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

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

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

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

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

Correspondingly, tuples (`tuple`) are sorted primarily by the first element, secondarily by the second element, etc.:

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

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

User-defined structs

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

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

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

Comparison function

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

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

Now a vector of strings can be sorted as follows:

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

3.3 Binary search

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

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

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

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

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

Method 1

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

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

The above idea can be implemented as follows:

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

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

Method 2

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

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

The following code implements the above idea:

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

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

Finding the smallest solution

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

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

x	0	1	$\cdots$	$k-1$	k	$k+1$	$\cdots$
`ok`(x)	`false`	`false`	$\cdots$	`false`	`true`	`true`	$\cdots$

The value k can be found using binary search:

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

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

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

Finding the maximum value

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

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

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

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

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

Chapter 4

Data structures

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

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

4.1 Dynamic array

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

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

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

After this, the elements can be accessed like in a regular array:

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

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

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

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

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

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

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

The following code creates a vector with five elements:

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

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

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

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

The internal implementation of the vector uses a regular array. If the size of the vector increases and the array becomes too small, a new array is allocated and all the elements are copied to the new array. However, this doesn't happen often and the time complexity of `push_back` is $O(1)$ on average.

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

The following code presents some string operations:

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

4.2 Set structure

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

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

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

The following code creates a set that consists of integers, and shows how to use it. The function `insert` adds an element to the set, the function `count` returns how many times an element appears in the set, and the function `erase` removes an element from the set.

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

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

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

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

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

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

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

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

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

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

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

4.3 Map structure

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

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

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

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

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

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

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

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

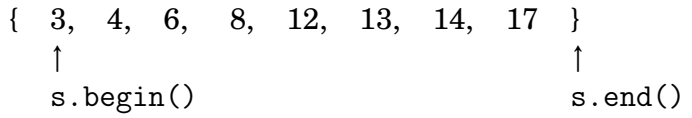
The following code prints all keys and values in the map:

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

4.4 Iterators and ranges

Many functions in the C++ standard library are given iterators to data structures, and iterators often correspond to ranges. An **iterator** is a variable that points to an element in a data structure.

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



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 $1 \ldots 10^9$:

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

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

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

Chapter 5

Complete search

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

Complete search is a good technique if it is feasible to go through all the solutions, because the search is usually easy to implement and it always gives the correct answer. If complete search is too slow, greedy algorithms or dynamic programming, presented in the next chapters, may be used.

5.1 Generating subsets

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

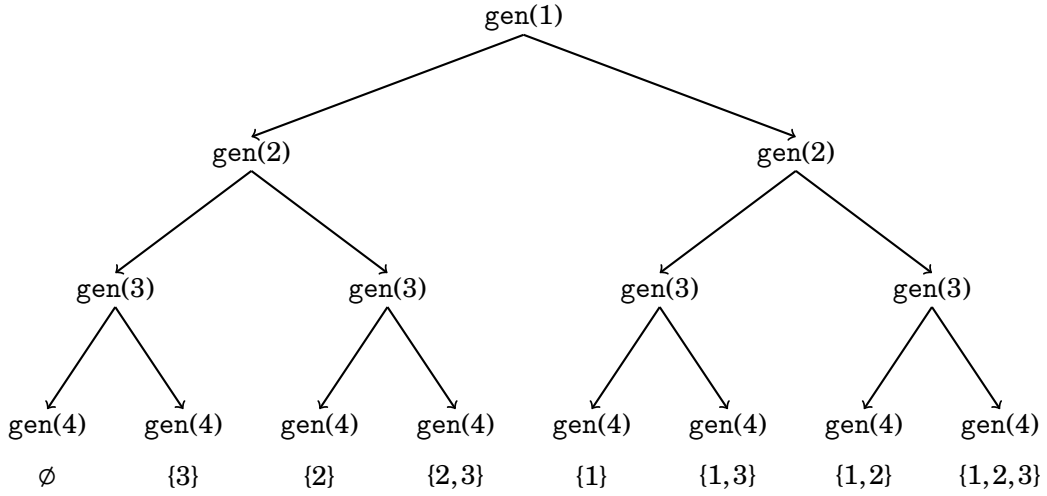
Method 1

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

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

The parameter k is the number that is the next candidate to be included in the subset. The function branches to two cases: either k is included or it is not included in the subset. Finally, when $k = n + 1$, a decision has been made for all the numbers and one subset has been generated.

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



Method 2

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

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

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

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

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

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

5.2 Generating permutations

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

Method 1

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

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

Each function call adds a new element to the permutation in the vector v. The array p indicates which elements are already included in the permutation. If 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

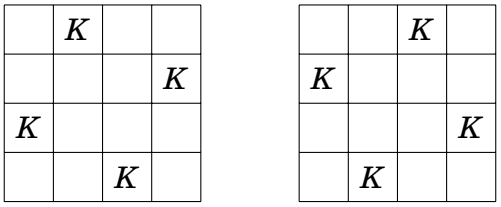
Another method is to begin from permutation $\{1,2,\ldots,n\}$ and at each step generate the next permutation in increasing order. The C++ standard library contains the function next_permutation that can be used for this. The following code generates the permutations of the set $\{1,2,\ldots,n\}$ using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

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

5.3 Backtracking

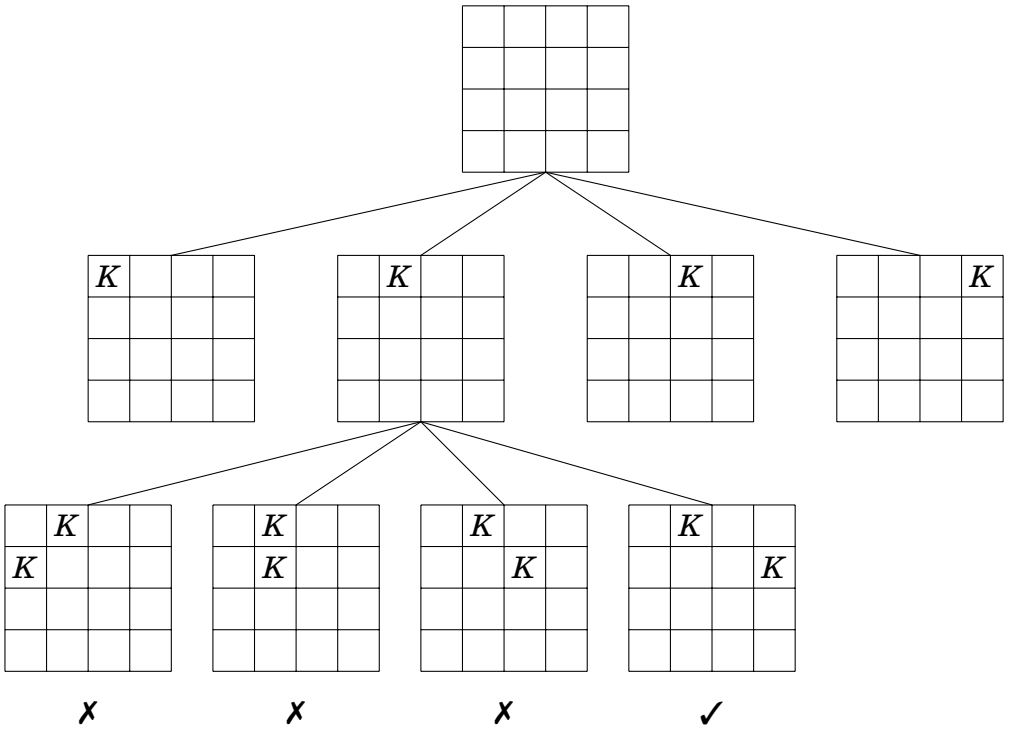
A **backtracking** algorithm begins from an empty solution and extends the solution step by step. At each step, the search branches to all possible directions how the solution can be extended. After processing one branch, the search continues to other possible directions.

As an example, consider the **queen problem** where our task is to calculate the number of ways we can place n queens to an $n \times n$ chessboard so that no two queens attack each other. For example, when $n = 4$, there are two possible solutions for the problem:



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

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



At the bottom level, the three first subsolutions are not valid because the queens attack each other. However, the fourth subsolution is valid and it can be extended to a full solution by placing two more queens to the board.

The following code implements the search:

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

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

The code assumes that the rows and columns of the board are numbered from 0. The function places a queen to row y when $0 \leq y < n$. 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

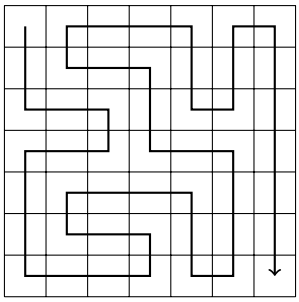
Using the presented backtracking algorithm, we can calculate that, for example, there are 92 ways to place 8 queens to an 8×8 chessboard. When n increases, the search quickly becomes slow because the number of the solutions increases exponentially. For example, calculating the ways to place 16 queens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

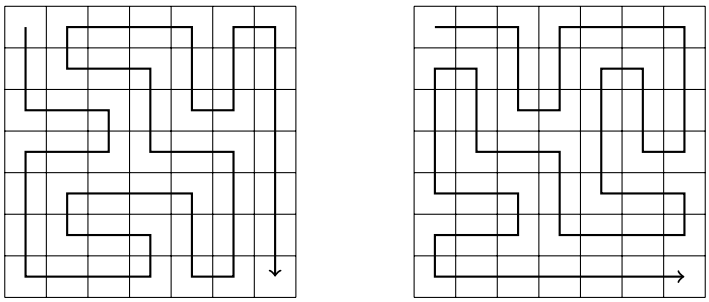
Basic algorithm

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:

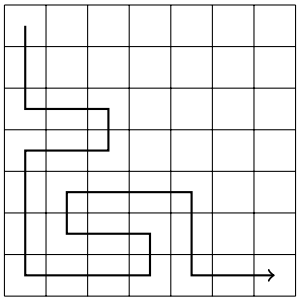


Thus, we can decide that the first step in the solution is always downward, and finally multiply the number of the solutions by two.

- 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:

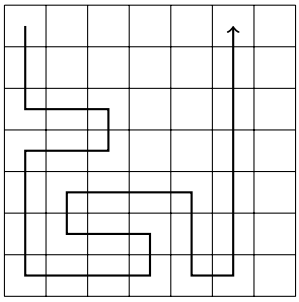


Using this observation, we can terminate the search branch immediately if we reach the lower-right square too early.

- 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:



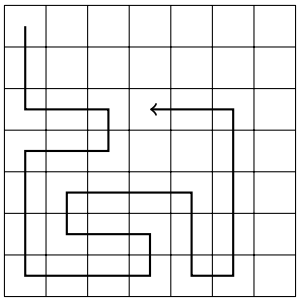
Now it will not be possible to visit every square, so we can terminate the search branch. This optimization is very useful:

- 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 trough 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.

Part II

Graph algorithms

Part III

Advanced topics

Index