Chapter 1
Introduction
Competitive programming combines two topics: (1) the design of algorithms and (2) the implementation of algorithms.
The design of algorithms consists of problem solving and mathematical thinking. Skills for analyzing problems and solving them creatively are needed. An algorithm for solving a problem has to be both correct and efficient, and the core of the problem is often about inventing an efficient algorithm.
Theoretical knowledge of algorithms is important to competitive programmers. Typically, a solution to a problem is a combination of well-known techniques and new insights. The techniques that appear in competitive programming also form the basis for the scientific research of algorithms.
The implementation of algorithms requires good programming skills. In competitive programming, the solutions are graded by testing an implemented algorithm using a set of test cases. Thus, it is not enough that the idea of the algorithm is correct, but the implementation also has to be correct.
A good coding style in contests is straightforward and concise. Programs should be written quickly, because there is not much time available. Unlike in traditional software engineering, the programs are short (usually at most a few hundred lines of code), and they do not need to be maintained after the contest.
Programming languages
At the moment, the most popular programming languages used in contests are C++, Python and Java. For example, in Google Code Jam 2017, among the best 3,000 participants, 79 % used C++, 16 % used Python and 8 % used Java [29]. 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 of 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 understand their strengths. For example, if large integers are needed in the problem, Python can be a good choice, because it contains built-in operations for calculating with

large integers. Still, most problems in programming contests are set so that using a specific programming language is not an unfair advantage.
All example programs in this book are written in C++, and the standard library's data structures and algorithms are often used. The programs follow the C++11 standard, which can be used in most contests nowadays. If you cannot program in C++ yet, now is a good time to start learning.
C++ code template
A typical C++ code 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 of the g++ compiler that allows us to include the entire standard library. Thus, it is not needed to

separately include libraries such as iostream, vector and algorithm, but rather they are available automatically.
The using line declares that the classes and functions of the standard library can be used directly in the code. Without the using line we would have to write, for example, std::cout, but now it suffices to write cout.
The code can be compiled using the following command:
g++ -std=c++11 -O2 -Wall test.cpp -o test
This command produces a binary file test from the source code test.cpp. The compiler follows the C++11 standard (-std=c++11), optimizes the code (-O2) and shows warnings about possible errors (-Wall).
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;
```
4
This kind of code always works, assuming that there is at least one space or newline between each element in the input. For example, the above code can read both of the following inputs:
```
123 456 monkey
```
```
123
456
monkey
```
The cout stream is used for output as follows:
```
int a = 123, b = 456;
string x = "monkey";
cout << a << " " << b << " " << x << "\n";
```
Input and output is sometimes a bottleneck in the program. The following lines at the beginning of the code make input and output more efficient:
```
ios::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 containing spaces. This can be accomplished by using the getline function:

```
string s;
getline(cin, s);
```
If the amount of data is unknown, the following loop is useful:
```
while (cin >> x) {
// code
}
```
This loop reads elements from the input one after another, until there is no more data available in the input.

5

In some contest systems, files are used for input and output. An easy solution for this is to write the code as usual using standard streams, but add the following lines to the beginning of the code:
```
freopen("input.txt", "r", stdin);
freopen("output.txt", "w", stdout);
```
After this, the program reads the input from the file ”input.txt” and writes the output to the file ”output.txt”.

Working with numbers

Integers

The most used integer type in competitive programming is int, which is a 32-bit type with a value range of −231...231 −1 or about −2·109...2·109. If the type int is not enough, the 64-bit type long long can be used. It has a value range of −263...263 −1 or about −9·1018...9·1018.

The following code defines a long long variable:
```
long long x = 123456789123456789LL;
```
The suffix LL means that the type of the number is long long.

A common mistake when using the type long long is that the type int is still used somewhere in the code. For example, the following code contains a subtle error:
```
int a = 123456789;
long long b = a*a;
cout << b << "\n"; // -1757895751
```
Even though the variable b is of type long long, both numbers in the expression a*a are of type int and the result is also of type int. Because of this, the variable b will contain a wrong result. The problem can be solved by changing the type of a to long long or by changing the expression to (long long)a*a.

Usually contest problems are set so that the type long long is enough. Still, it is good to know that the g++ compiler also provides a 128-bit type __int128_t with a value range of −2127...2127 −1 or about −1038...1038. However, this type is not available in all contest systems.

Modular arithmetic

We denote by x mod m the remainder when x is divided by m. For example, 17 mod 5 = 2, because 17 = 3·5+2.

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

6

example, ”modulo 109 +7”). The idea is that even if the actual answer is very large, it suffices to use the types int and long long.

An important property of the remainder is that in addition, subtraction and

multiplication, the remainder can be taken before the operation:
(a+ b) mod m
=
(a mod m+ b mod m) mod m
(a− b) mod m
=
(a mod m− b mod m) mod m
(a· b) mod m
=
(a mod m· b mod m) mod m
Thus, we can take the remainder after every operation and the numbers will never become too large.
For example, the following code calculates n!, the factorial of n, modulo m:
long long x = 1;
for (int i = 2; i <= n; i++) {
x = (x*i)%m;
}
cout << x%m << "\n";
Usually we want the remainder to always be between 0...m−1. However, in C++ and other languages, the remainder of a negative number is either zero or negative. An easy way to make sure there are no negative remainders is to first calculate the remainder as usual and then add m if the result is negative:
x = x%m;
if (x < 0) x += m;
However, this is only needed when there are subtractions in the code and the remainder may become negative.
Floating point numbers
The usual floating point types in competitive programming are the 64-bit double and, as an extension in the g++ compiler, the 80-bit long double. In most cases, double is enough, but long double is more accurate.
The required precision of the answer is usually given in the problem statement. An easy way to output the answer is to use the printf function and give the number of decimal places in the formatting string. For example, the following code prints the value of x with 9 decimal places:
printf("%.9f\n", x);
A difficulty when using floating point numbers is that some numbers cannot be represented accurately as floating point numbers, and 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
7
Due to a rounding error, the value of x is a bit smaller than 1, while the correct value would be 1.
It is risky to compare floating point numbers with the == operator, because it is possible that the values should be equal but they are not because of precision errors. A better way to compare floating point numbers is to assume that two numbers are equal if the difference between them is less than ε, where ε is a small number.

In practice, the numbers can be compared as follows (ε = 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 still be represented accurately. For example, using double, it is possible to accurately represent all integers whose absolute value is at most 253.

Shortening code

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

Type names

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

```
typedef long long ll;
```

After this, the code

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

can be shortened as follows:

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

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

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


Macros

Another way to shorten 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 #define keyword.

For example, we can define the following macros:

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

After this, the code

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

can be shortened as follows:

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

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

#define REP(i,a,b) for (int i = a; i <= b; i++)
After this, the code
for (int i = 1; i <= n; i++) {
search(i);
}
can be shortened as follows:
REP(i,1,n) {
search(i);
}
Sometimes macros cause bugs that may be difficult to detect. For example, consider the following macro that calculates the square of a number:
#define SQ(a) a*a
This macro does not always work as expected. For example, the code
cout << SQ(3+3) << "\n";
9
corresponds to the code
cout << 3+3*3+3 << "\n"; // 15
A better version of the macro is as follows:
#define SQ(a) (a)*(a)
Now the code
cout << SQ(3+3) << "\n";
corresponds to the code
cout << (3+3)*(3+3) << "\n"; // 36
Mathematics
Mathematics plays an important role in competitive programming, and it is not possible to become a successful competitive programmer without having good mathematical skills. This section discusses some important mathematical concepts and formulas that are needed later in the book.
Sum formulas
Each sum of the form
n
x=1
xk = 1k +2k +3k +...+ nk,
where k is a positive integer, has a closed-form formula that is a polynomial of degree k +1. For example1,
n
x=1
x = 1+2+3+...+ n = n(n+1)
2
and
n
x=1
x2 = 12 +22 +32 +...+ n2 = n(n+1)(2n+1)
6
.
An arithmetic progression is a sequence of numbers where the difference between any two consecutive numbers is constant. For example,
3,7,11,15

1 There is even a general formula for such sums, called Faulhaber's formula, but it is too complex to be presented here.

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

$$\underbrace{a+\cdots+b}_{n \text{ numbers}} = n\frac{(a+b)}{2}$$

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

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

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

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

3,6,12,24

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

$$a+ ak + ak^2 +\cdots+ b = \frac{bk - a}{k - 1}$$

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

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

This formula can be derived as follows. Let

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

By multiplying both sides by k, we get

$$kS = ak + ak^2 + ak^3 +\cdots+ bk,$$

and solving the equation

$$kS - S = bk - a$$

yields the formula.

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

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

A harmonic sum is a sum of the form

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

An upper bound for a harmonic sum is log2(n)+1. Namely, we can modify each term 1/k so that k becomes the nearest power of two that does not exceed k.

For example, when n = 6, we can estimate the sum as follows:
1+ 1
2 + 1
3 + 1
4 + 1
5 + 1
6 ≤ 1+ 1
2 + 1
2 + 1
4 + 1
4 + 1
4.
This upper bound consists of log2(n)+1 parts (1, 2·1/2, 4·1/4, etc.), and the value of each part is at most 1.
11
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 ∅ denotes an empty set, and |S| denotes the size of a set S, i.e., the number of elements in the set. For example, in the above set, |X| = 3.
If a set S contains an element x, we write x ∈ S, and otherwise we write x ∉ S. For example, in the above set
4 ∈ X
and
5 ∉ X.
New sets can be constructed using set operations:
• The intersection A ∩B consists of elements that are in both A and B. For example, if A = {1,2,5} and B = {2,4}, then A ∩B = {2}.
• The union A ∪ 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 ∪B = {2,3,7,8}.
• The complement $\overline{A}$ consists of elements that are not in A. The interpretation of a complement depends on the universal set, which contains all possible elements. For example, if A = {1,2,5,7} and the universal set is {1,2,...,10}, then $\overline{A}$ = {3,4,6,8,9,10}.
• The difference A \ B = A ∩ $\overline{B}$ 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 \B = {2,7}.
If each element of A also belongs to S, we say that A is a subset of S, denoted by A ⊂ S. A set S always has 2|S| subsets, including the empty set. For example, the subsets of the set {2,4,7} are
∅, {2}, {4}, {7}, {2,4}, {2,7}, {4,7} and {2,4,7}.
Some often used sets are N (natural numbers), Z (integers), Q (rational numbers) and R (real numbers). The set N can be defined in two ways, depending on the situation: either N = {0,1,2,...} or N = {1,2,3,...}.
We can also construct a set using a rule of the form
{f (n) : n ∈ S},
where f (n) is some function. This set contains all elements of the form f (n),

where n is an element in S. For example, the set
$X = \{2n : n \in Z\}$
contains all even integers.
12
Logic
The value of a logical expression is either true (1) or false (0). The most important logical operators are ¬ (negation), $\wedge$ (conjunction), $\vee$ (disjunction), $\Rightarrow$ (implication) and $\Leftrightarrow$ (equivalence). The following table shows the meanings of these operators:

A	B	¬A	¬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

1
The expression ¬A has the opposite value of A. The expression A ∧B is true if both A and B are true, and the expression A ∨B is true if A or B or both are true. The expression A ⇒ B is true if whenever A is true, also B is true. The expression A ⇔ B is true if A and B are both true or both false.
A predicate is an expression that is true or false depending on its parameters. Predicates are usually denoted by capital letters. For example, we can define a predicate P(x) that is true exactly when x is a prime number. Using this definition, P(7) is true but P(8) is false.
A quantifier connects a logical expression to the elements of a set. The most important quantifiers are ∀ (for all) and ∃ (there is). For example,
∀x(∃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,
∀x((x > 1∧¬P(x)) ⇒ (∃a(∃b(a > 1∧ b > 1∧ x = ab))))
means that if a number x is larger than 1 and not a prime number, then there are numbers a and b that are larger than 1 and whose product is x. This proposition is true in the set of integers.
Functions
The function ⌊x⌋ rounds the number x down to an integer, and the function ⌈x⌉ rounds the number x up to an integer. For example,
⌊3/2⌋ = 1
and
⌈3/2⌉ = 2.
The functions min(x1,x2,...,xn) and max(x1,x2,...,xn) give the smallest and largest of values x1,x2,...,xn. For example,
min(1,2,3) = 1
and
max(1,2,3) = 3.
13
The factorial n! can be defined
n
∏
x=1
x = 1·2·3·...· n
or recursively
0!
=
1
n!
=
n·(n−1)!
The Fibonacci numbers arise in many situations. They can be defined recursively as follows:
f (0)

=
0
f (1)
=
1
f (n)
=
f (n−1)+ f (n−2)
The first Fibonacci numbers are
0,1,1,2,3,5,8,13,21,34,55,...
There is also a closed-form formula for calculating Fibonacci numbers, which is sometimes called Binet's formula:
f (n) = (1+
√
5)n −(1−
√
5)n
2n√
5
.
Logarithms
The logarithm of a number x is denoted logk(x), where k is the base of the logarithm. According to the definition, logk(x) = a exactly when ka = x.
A useful property of logarithms is that logk(x) equals the number of times we have to divide x by k before we reach the number 1. For example, log2(32) = 5 because 5 divisions by 2 are needed:
32 → 16 → 8 → 4 → 2 → 1
Logarithms are often used in the analysis of algorithms, because many efficient algorithms halve something at each step. Hence, we can estimate the efficiency of such algorithms using logarithms.
The logarithm of a product is
logk(ab) = logk(a)+logk(b),
and consequently,
logk(xn) = n·logk(x).
In addition, the logarithm of a quotient is
logk
(a
b
) = logk(a)−logk(b).
Another useful formula is
logu(x) = logk(x)
logk(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 ≈ 2.71828. Another property of logarithms is that the number of digits of an

integer x in base b is ⌊logb(x)+1⌋. For example, the representation of 123 in base 2 is 1111011 and ⌊log2(123)+1⌋ = 7.

Contests and resources

IOI

The International Olympiad in Informatics (IOI) is an annual programming contest for secondary school students. Each country is allowed to send a team of four students to the contest. There are usually about 300 participants from 80 countries.

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

The IOI syllabus [41] regulates the topics that may appear in IOI tasks. Almost all the topics in the IOI syllabus are covered by this book.

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

Some countries organize online practice contests for future IOI participants, such as the Croatian Open Competition in Informatics [11] and the USA Computing Olympiad [68]. In addition, a large collection of problems from Polish contests is available online [60].

ICPC

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

The ICPC consists of several stages, and finally the best teams are invited to the World Finals. While there are tens of thousands of participants in the contest, there are only a small number[2] of final slots available, so even advancing to the finals is a great achievement in some regions.

In each ICPC contest, the teams have five hours of time to solve about ten algorithm problems. A solution to a problem is accepted only if it solves all test cases efficiently. During the contest, competitors may view the results of other

[2]The exact number of final slots varies from year to year; in 2017, there were 133 final slots.

teams, but for the last hour the scoreboard is frozen and it is not possible to see the results of the last submissions.

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

Online contests

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

Some companies organize online contests with onsite finals. Examples of such

contests are Facebook Hacker Cup, Google Code Jam and Yandex.Algorithm. Of course, companies also use those contests for recruiting: performing well in a contest is a good way to prove one's skills.

Books

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

- S. S. Skiena and M. A. Revilla: Programming Challenges: The Programming Contest Training Manual [59]
- S. Halim and F. Halim: Competitive Programming 3: The New Lower Bound of Programming Contests [33]
- K. Diks et al.: Looking for a Challenge? The Ultimate Problem Set from the University of Warsaw Programming Competitions [15]

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

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

- T. H. Cormen, C. E. Leiserson, R. L. Rivest and C. Stein: Introduction to Algorithms [13]
- J. Kleinberg and É. Tardos: Algorithm Design [45]
- S. S. Skiena: The Algorithm Design Manual [58]

Chapter 2

Time complexity

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

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

Calculation rules

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

Loops

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

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

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

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

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

```
}
}
```
17

Order of magnitude

A time complexity does not tell us the exact number of times the code inside a loop is executed, but it only shows the order of magnitude. In the following examples, the code inside the loop is executed 3n, n+5 and ⌈n/2⌉ 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(n2):

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

Phases

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

For example, the following code consists of three phases with time complexities O(n), O(n2) and O(n). Thus, the total time complexity is O(n2).

```
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
}
```

18

Several variables

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

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

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

```
}
}
```

Recursion

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

For example, consider the following function:

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

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

As another example, consider the following function:

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

In this case each function call generates two other calls, except for n = 1. Let us see what happens when g is called with parameter n. The following table shows the function calls produced by this single call:

function call
number of calls
g(n)
1
g(n−1)
2
g(n−2)
4
…
…
g(1)
2n−1

Based on this, the time complexity is
1+2+4+⋯+2n−1 = 2n −1 = O(2n).

Complexity classes

The following list contains common time complexities of algorithms:

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

O(logn) A logarithmic algorithm often halves the input size at each step. The running time of such an algorithm is logarithmic, because log2 n equals the number of times n must be divided by 2 to get 1.

O(√n) A square root algorithm is slower than O(logn) but faster than O(n). A special property of square roots is that √n = n/√n, so the square root √n lies, in some sense, in the middle of the input.

O(n) A linear algorithm goes through the input a constant number of times. This is often the best possible time complexity, because it is usually necessary to access each input element at least once before reporting the answer.
O(nlogn) This time complexity often indicates that the algorithm sorts the input, because the time complexity of efficient sorting algorithms is O(nlogn). Another possibility is that the algorithm uses a data structure where each operation takes O(logn) time.
O(n2) A quadratic algorithm often contains two nested loops. It is possible to go through all pairs of the input elements in O(n2) time.
O(n3) A cubic algorithm often contains three nested loops. It is possible to go through all triplets of the input elements in O(n3) time.
O(2n) This time complexity often indicates that the algorithm iterates through all subsets of the input elements. For example, the subsets of {1,2,3} are ∅, {1}, {2}, {3}, {1,2}, {1,3}, {2,3} and {1,2,3}.
O(n!) This time complexity often indicates that the algorithm iterates through all permutations of the input elements. For example, the permutations of {1,2,3} are (1,2,3), (1,3,2), (2,1,3), (2,3,1), (3,1,2) and (3,2,1).
An algorithm is polynomial if its time complexity is at most O(nk) where k is a constant. All the above time complexities except O(2n) 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 known1.
1A classic book on the topic is M. R. Garey’s and D. S. Johnson’s Computers and Intractability:
A Guide to the Theory of NP-Completeness [28].
20
Estimating efficiency
By calculating the time complexity of an algorithm, it is possible to check, before implementing the algorithm, that it is efficient enough for the problem. The starting point for estimations is the fact that a modern computer can perform some hundreds of millions of operations in a second.
For example, assume that the time limit for a problem is one second and the input size is n = 105. If the time complexity is O(n2), the algorithm will perform about (105)2 = 1010 operations. This should take at least some tens of seconds, so the algorithm seems to be too slow for solving the problem.
On the other hand, given the input size, we can try to guess the required time complexity of the algorithm that solves the problem. The following table contains some useful estimates assuming a time limit of one second.
input size
required time complexity
n ≤ 10
O(n!)
n ≤ 20
O(2n)
n ≤ 500

O(n3)
n ≤ 5000
O(n2)
n ≤ 106
O(nlogn) or O(n)
n is large
O(1) or O(logn)
For example, if the input size is n = 105, it is probably expected that the time complexity of the algorithm is O(n) or O(nlogn). This information makes it easier to design the algorithm, because it rules out approaches that would yield an algorithm with a worse time complexity.
Still, it is important to remember that a time complexity is only an estimate of efficiency, because it hides the constant factors. For example, an algorithm that runs in O(n) time may perform n/2 or 5n operations. This has an important effect on the actual running time of the algorithm.
Maximum subarray sum
There are often several possible algorithms for solving a problem such that their time complexities are different. This section discusses a classic problem that has a straightforward O(n3) solution. However, by designing a better algorithm, it is possible to solve the problem in O(n2) time and even in O(n) time.
Given an array of n numbers, our task is to calculate the maximum subarray sum, i.e., the largest possible sum of a sequence of consecutive values in the array2. The problem is interesting when there may be negative values in the array. For example, in the array
−1
2
4
−3
5
2
−5
2
2J. Bentley's book Programming Pearls [8] made the problem popular.
21
the following subarray produces the maximum sum 10:
−1
2
4
−3
5
2
−5
2
We assume that an empty subarray is allowed, so the maximum subarray sum is always at least 0.
Algorithm 1
A straightforward way to solve the problem is to go through all possible subarrays, calculate the sum of values in each subarray and maintain the maximum sum.

The following code implements this algorithm:

```
int best = 0;
for (int a = 0; a < n; a++) {
for (int b = a; b < n; b++) {
int sum = 0;
for (int k = a; k <= b; k++) {
sum += array[k];
}
best = max(best,sum);
}
}
cout << best << "\n";
```

The variables a and b fix the first and last index of the subarray, and the sum of values is calculated to the variable sum. The variable best contains the maximum sum found during the search.

The time complexity of the algorithm is O(n3), because it consists of three nested loops that go through the input.

Algorithm 2

It is easy to make Algorithm 1 more efficient by removing one loop from it. This is possible by calculating the sum at the same time when the right end of the subarray moves. The result is the following code:

```
int best = 0;
for (int a = 0; a < n; a++) {
int sum = 0;
for (int b = a; b < n; b++) {
sum += array[b];
best = max(best,sum);
}
}
cout << best << "\n";
```

After this change, the time complexity is O(n2).

22

Algorithm 3

Surprisingly, it is possible to solve the problem in O(n) time3, which means that just one loop is enough. The idea is to calculate, for each array position, the maximum sum of a subarray that ends at that position. After this, the answer for the problem is the maximum of those sums.

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

1. The subarray only contains the element at position k.
2. The subarray consists of a subarray that ends at position k −1, followed by the element at position k.

In the latter case, since we want to find a subarray with maximum sum, the subarray that ends at position k −1 should also have the maximum sum. Thus, we can solve the problem efficiently by calculating the maximum subarray sum for each ending position from left to right.

The following code implements the algorithm:

```
int best = 0, sum = 0;
```

```
for (int k = 0; k < n; k++) {
sum = max(array[k],sum+array[k]);
best = max(best,sum);
}
cout << best << "\n";
```

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

Efficiency comparison

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

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

array size n	Algorithm 1	Algorithm 2	Algorithm 3
102	0.0 s	0.0 s	0.0 s
103	0.1 s	0.0 s	0.0 s
104	> 10.0 s	0.1 s	0.0 s
105	> 10.0 s	5.3 s	0.0 s
106	> 10.0 s	> 10.0 s	0.0 s
107	> 10.0 s	> 10.0 s	0.0 s

3In [8], this linear-time algorithm is attributed to J. B. Kadane, and the algorithm is sometimes called Kadane's algorithm.

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

of the algorithms. Algorithm 1 becomes slow when n = 104, and Algorithm 2 becomes slow when n = 105. Only Algorithm 3 is able to process even the largest inputs instantly.

24

Chapter 3
Sorting

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

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

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

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
3
8
2
9
2
5
6

will be as follows after sorting:

1
2
2
3
5
6
8
9

O(n2) algorithms

Simple algorithms for sorting an array work in O(n2) time. Such algorithms are short and usually consist of two nested loops. A famous O(n2) time sorting

25

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

Bubble sort consists of n rounds. On each round, the algorithm iterates through the elements of the array. Whenever two consecutive elements are found that are not in correct order, the algorithm swaps them. The algorithm can be

implemented as follows:

```
for (int i = 0; i < n; i++) {
for (int j = 0; j < n-1; j++) {
if (array[j] > array[j+1]) {
swap(array[j],array[j+1]);
}
}
}
```

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