Competitive Programmer's Handbook

Antti Laaksonen

December 29, 2016

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:

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

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

Set theory

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

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

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

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

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

New sets can be constructed as follows using set operations:

- The **intersection** $A \cap B$ consists of elements that are both in A and B. For example, if $A = \{1, 2, 5\}$ and $B = \{2, 4\}$, then $A \cap B = \{2\}$.
- The **union** $A \cup B$ consists of elements that are in A or B or both. For example, if $A = \{3, 7\}$ and $B = \{2, 3, 8\}$, then $A \cup B = \{2, 3, 7, 8\}$.
- The **complement** $\bar{A}$ consists of elements that are not in A. The interpretation of a complement depends on the **universal set** that contains all possible elements. For example, if $A = \{1, 2, 5, 7\}$ and the universal set is $P = \{1, 2, \ldots, 10\}$, then $\bar{A} = \{3, 4, 6, 8, 9, 10\}$.
- The **difference** $A \setminus B = A \cap \bar{B}$ consists of elements that are in A but not in B. Note that B can contain elements that are not in A. For example, if $A = \{2, 3, 7, 8\}$ and $B = \{3, 5, 8\}$, then $A \setminus B = \{2, 7\}$.

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

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

Often used sets are

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

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

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

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

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

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

contains all even integers.

Logic

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

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

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

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

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

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

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

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

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

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

Functions

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

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

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

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

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

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

or recursively

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

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

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

The first Fibonacci numbers are

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

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

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

Logarithm

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

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

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

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

The logarithm of a product is

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

and consequently,

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

In addition, the logarithm of a quotient is

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

Another useful formula is

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

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

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

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

Chapter 2

Time complexity

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

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

2.1 Calculation rules

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

Loops

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

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

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

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

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

Order of magnitude

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

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

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

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

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

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

Phases

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

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

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

Several variables

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

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

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

Recursion

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

For example, consider the following function:

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

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

As another example, consider the following function:

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

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

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

Based on this, the time complexity is

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

2.2 Complexity classes

Typical complexity classes are:

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

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

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

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

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

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

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

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

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

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

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

2.3 Estimating efficiency

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

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

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

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

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

Still, it is important to remember that a time complexity doesn't tell everything about the efficiency because it hides the **constant factors**. For example, an algorithm that runs in $O(n)$ time can perform $n/2$ or $5n$ operations. This has an important effect on the actual running time of the algorithm.

2.4 Maximum subarray sum

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

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

1	2	3	4	5	6	7	8
−1	2	4	−3	5	2	−5	2

the following subarray produces the maximum sum 10:

1	2	3	4	5	6	7	8
−1	2	4	−3	5	2	−5	2

Solution 1

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

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

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

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

Solution 2

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

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

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

Solution 3

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

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

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

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

The following code implements the solution:

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

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

Efficiency comparison

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

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

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

The comparison shows that all algorithms are efficient when the input size is small, but larger inputs bring out remarkable differences in running times of the algorithms. The $O(n^3)$ time solution 1 becomes slower when $n = 10^3$, and the $O(n^2)$ time solution 2 becomes slower when $n = 10^4$. Only the $O(n)$ time solution 3 solves even the largest inputs instantly.

Part II

Graph algorithms

Part III

Advanced topics

Index