

Competitive Programmer's Handbook

Antti Laaksonen

Draft July 3, 2018

ii

Contents

Preface ix

I Basic techniques 1

1 Introduction	3
1.1 Programming languages	3
1.2 Input and output	4
1.3 Working with numbers	6
1.4 Shortening code	8
1.5 Mathematics	10
1.6 Contests and resources	15

2 Time complexity	17
2.1 Calculation rules	17
2.2 Complexity classes	20
2.3 Estimating efficiency	21
2.4 Maximum subarray sum	21

3 Sorting	25
3.1 Sorting theory	25
3.2 Sorting in C++	29
3.3 Binary search	31

4 Data structures	35
4.1 Dynamic arrays	35
4.2 Set structures	37
4.3 Map structures	38
4.4 Iterators and ranges	39
4.5 Other structures	41
Comparison to sorting	44

5 Complete search	47
5.1 Generating subsets	47
5.2 Generating permutations	49
5.3 Backtracking	50
5.4 Pruning the search	51
5.5 Meet in the middle	54

iii

6 Greedy algorithms	57
6.1 Coin problem	57
6.2 Scheduling	58
6.3 Tasks and deadlines	60
6.4 Minimizing sums	61
6.5 Data compression	62

7 Dynamic programming	65
7.1 Coin problem	

65	7.2 Longest increasing subsequence	70
	7.3 Paths in a grid	71
	7.4 Knapsack problems	72
	7.5 Edit distance	74
	7.6 Counting tilings	75
8	Amortized analysis 77	
	8.1 Two pointers method	77
	8.2 Nearest smaller elements	79
	8.3 Sliding window minimum	81
9	Range queries 83	
	9.1 Static array queries	84
	9.2 Binary indexed tree	86
	9.3 Segment tree	89
	9.4 Additional techniques	93
10	Bit manipulation 95	
	10.1 Bit representation	95
	10.2 Bit operations	96
	10.3 Representing sets	98
	10.4 Bit optimizations	100
	10.5 Dynamic programming	102
II Graph algorithms 107		
11	Basics of graphs 109	
	11.1 Graph terminology	109
	11.2 Graph representation	113
12	Graph traversal 117	
	12.1 Depth-first search	117
	12.2 Breadth-first search	119
	12.3 Applications	121
iv		
13	Shortest paths 123	
	13.1 Bellman–Ford algorithm	123
	13.2 Dijkstra’s algorithm	126
	13.3 Floyd–Warshall algorithm	129
14	Tree algorithms 133	
	14.1 Tree traversal	134
	14.2 Diameter	135
	14.3 All longest paths	137
	14.4 Binary trees	139
15	Spanning trees 141	
	15.1 Kruskal’s algorithm	142
	15.2 Union-find structure	145
	15.3 Prim’s algorithm	147
16	Directed graphs 149	
	16.1 Topological sorting	149
	16.2 Dynamic programming	151
	16.3 Successor paths	154
	16.4 Cycle detection	155

17 Strong connectivity	157	17.1 Kosaraju's algorithm	163
	158	17.2 2SAT problem	160
18 Tree queries	163	18.1 Finding ancestors	163
		18.2 Subtrees and paths	164
		18.3 Lowest common ancestor	167
		18.4 Offline algorithms	170
19 Paths and circuits	173	19.1 Eulerian paths	173
		19.2 Hamiltonian paths	177
		19.3 De Bruijn sequences	178
		19.4 Knight's tours	179
20 Flows and cuts	181	20.1 Ford–Fulkerson algorithm	182
		20.2 Disjoint paths	186
		20.3 Maximum matchings	187
		20.4 Path covers	190

III Advanced topics 195

21 Number theory	197	21.1 Primes and factors	197
		21.2 Modular arithmetic	201
		21.3 Solving equations	204
		21.4 Other results	205
22 Combinatorics	207	22.1 Binomial coefficients	208
		22.2 Catalan numbers	210
		22.3 Inclusion-exclusion	212
		22.4 Burnside's lemma	214
		22.5 Cayley's formula	215
23 Matrices	217	23.1 Operations	217
		23.2 Linear recurrences	220
		23.3 Graphs and matrices	222
24 Probability	225	24.1 Calculation	225
		24.2 Events	226
		24.3 Random variables	228
		24.4 Markov chains	230
		24.5 Randomized algorithms	231
25 Game theory	235	25.1 Game states	235
		25.2 Nim game	237
		25.3 Sprague–Grundy theorem	238
26 String algorithms	243	26.1 String terminology	243
		26.2 Trie structure	244
		26.3 String hashing	245
		26.4 Z-algorithm	247

27 Square root algorithms 251	27.1 Combining algorithms	252
	27.2 Integer partitions	254
	27.3 Mo's algorithm	255

28 Segment trees revisited 257	28.1 Lazy propagation	258
	28.2 Dynamic trees	261
	28.3 Data structures	263
	28.4 Two-dimensionality	264

29 Geometry 265	29.1 Complex numbers	266
	29.2 Points and lines	268
	29.3 Polygon area	271
	29.4 Distance functions	272

30 Sweep line algorithms 275	30.1 Intersection points	276
	30.2 Closest pair problem	277
	30.3 Convex hull problem	278

Preface

The purpose of this book is to give you a thorough introduction to competitive programming. It is assumed that you already know the basics of programming, but no previous background in competitive programming is needed.

The book is especially intended for students who want to learn algorithms and possibly participate in the International Olympiad in Informatics (IOI) or in the International Collegiate Programming Contest (ICPC). Of course, the book is also suitable for 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 get a good general understanding of algorithms if you spend time reading the book, solving problems and taking part in contests.

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

Helsinki, July 2018
Antti Laaksonen

Part I

Basic techniques ¹

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

3

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 $-2^{31} \dots 2^{31} - 1$ or about $-2 \cdot 10^9 \dots 2 \cdot 10^9$. If the type `int` is not enough, the 64-bit type `long long` can be used. It has a value range of $-2^{63} \dots 2^{63} - 1$ or about $-9 \cdot 10^{18} \dots 9 \cdot 10^{18}$.

The following code defines a long long variable:

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

The suffix LL means that the type of the number is long long. A 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 $-2^{127} \dots 2^{127} - 1$ or about $-10^{38} \dots 10^{38}$. However, this type is not available in all contest systems.

Modular arithmetic

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

Sometimes, the answer 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 $10^9 + 7$ "). 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:

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

Thus, we can 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 $0 \dots m-1$. 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 ($\epsilon = 10^{-9}$):

```
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 2^{53} .

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

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

where k is a positive integer, has a closed-form formula that is a polynomial of degree $k+1$. For example¹,

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

and

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

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

3, 7, 11, 15

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

10

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

$$\sum_{i=1}^n (a + (i-1)d) = \frac{n}{2}(a + b)$$

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

example,

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

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

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

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

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

By multiplying both sides by k , we get

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

and solving the equation

$$kS - S = bk - a$$

yields the formula.

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

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

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

$$\sum_{x=1}^n \frac{1}{x}$$

$$1 + \frac{1}{2} + \frac{1}{3} + \dots$$

An upper bound for a harmonic sum is $\log_2(n) + 1$. 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: $1 + \frac{1}{2} + \frac{1}{3} + \frac{1}{4} + \frac{1}{5} + \frac{1}{6} \leq$

$$1 + \frac{1}{2} + \frac{1}{2} + \frac{1}{4} + \frac{1}{4} + \frac{1}{4}.$$

This upper bound consists of $\log_2(n)+1$ parts ($1, 2 \cdot 1/2, 4 \cdot 1/4$, etc.), and the 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 \emptyset 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 $x \in S$, and otherwise we write $x \notin S$. For example, in the above set

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

New sets can be constructed using set operations:

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

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

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

Some often used sets are \mathbb{N} (natural numbers), \mathbb{Z} (integers), \mathbb{Q} (rational numbers) and \mathbb{R} (real numbers). The set \mathbb{N} can be defined in two ways, depending on the situation: either $\mathbb{N} = \{0, 1, 2, \dots\}$ or $\mathbb{N} = \{1, 2, 3, \dots\}$.

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

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

where $f(n)$ is some function. This set contains all elements of the form $f(n)$, where

n is an element in S . For example, the set

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

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 \neg (**negation**), \wedge (**conjunction**), \vee (**disjunction**), \Rightarrow (**implication**) and \Leftrightarrow (**equivalence**). The following table shows the meanings of these operators:

$$\begin{array}{cccccccc} A & B & \neg A & \neg B & A \wedge B & A \vee B & A \Rightarrow B & A \Leftrightarrow B \\ 0 & 0 & 1 & 1 & 0 & 0 & 1 & 1 \\ 0 & 1 & 1 & 0 & 0 & 1 & 1 & 0 \\ 1 & 0 & 0 & 1 & 0 & 1 & 0 & 0 \\ 1 & 1 & 0 & 0 & 1 & 1 & 1 & 1 \end{array}$$

The expression $\neg A$ has the opposite value of A . 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 the elements of 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,

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

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

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

The functions $\min(x_1, x_2, \dots, x_n)$ and $\max(x_1, x_2, \dots, x_n)$ give the smallest and

largest of values x_1, x_2, \dots, x_n . For example,

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

13

The **factorial** $n!$ can be defined

$$x = 1 \cdot 2 \cdot 3 \cdot \dots \cdot n$$

or recursively

$$0! = 1$$

$$n! = n \cdot (n-1)!$$

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

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

There is also a closed-form formula for calculating Fibonacci numbers, which is sometimes called **Binet's formula**:

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

Logarithms

The **logarithm** of a number x is denoted $\log_k(x)$, where k is the base of the logarithm. According to the definition, $\log_k(x) = a$ exactly when $k^a = x$. A useful property of logarithms 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 by 2 are needed:

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

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

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

and consequently,

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

In addition, the logarithm of a quotient is

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

$$= \log_k(a) - \log_k(b).$$

Another useful formula is ¹⁴

$$\log_u(x) = \log_k(x) \log_k(u),$$

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 logarithms is that the number of digits of an integer x in base b is $\lceil \log_b(x) \rceil + 1$. For example, the representation of 123 in base 2 is 1111011 and $\lceil \log_2(123) \rceil + 1 = 7$.

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 number² 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

²The exact number of final slots varies from year to year; in 2017, there were 133 final slots. 15

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

Loops

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  
    }  
}
```

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 $O(n^2)$:

```
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 $O(n)$, $O(n^2)$ 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
}

```

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

Based on this, the time complexity is

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

19

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.

$O(\log n)$ A **logarithmic** algorithm often halves the input size at each step. The running time of such an algorithm is logarithmic, because $\log_2 n$ equals the number of times n must be divided by 2 to get 1.

$O(\sqrt{n})$ A **square root algorithm** is slower than $O(\log n)$ but faster than $O(n)$. A special property of square roots is that $\sqrt{n} = n / \sqrt{n}$, so the square root \sqrt{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.

$O(n \log n)$ This time complexity often indicates 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 each operation takes $O(\log n)$ time.

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

$O(2^n)$ 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 \emptyset , $\{1\}$, $\{2\}$, $\{3\}$, $\{1,2\}$, $\{1,3\}$, $\{2,3\}$ 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 $O(n^k)$ 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 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].

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 $n = 10^5$. If the time complexity is $O(n^2)$, the algorithm will perform about $(10^5)^2 = 10^{10}$ 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
$n \leq 10$	$O(n!)$
$n \leq 20$	$O(2^n)$
$n \leq 500$	$O(n^3)$
$n \leq 5000$	$O(n^2)$
$n \leq 10^6$	$O(n \log n)$ or $O(n)$
n is large	$O(1)$ or $O(\log n)$

For example, if the input size is $n = 10^5$, it is probably expected that the time complexity of the algorithm is $O(n)$ or $O(n \log n)$. 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 $O(n^3)$ 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 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 $O(n^3)$, 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 $O(n^2)$.

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
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

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

23

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 $n = 10^4$, and Algorithm 2 becomes slow when $n = 10^5$. Only Algorithm 3 is able to process even the largest inputs instantly.

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 $O(n \log n)$ 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

$O(n^2)$ 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 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 be sorted. For example, in the array

1 3 8 2 9 2 5 6

the first round of bubble sort swaps elements as follows:

1 3 2 8 9 2 5 6

1 3 2 8 2 9 5 6

1 3 2 8 2 5 9 6

1 3 2 8 2 5 6 9

Inversions

Bubble sort is an example of a sorting algorithm that always swaps *consecutive* elements in the array. It turns out that the time complexity of such 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**: a pair of array elements ($\text{array}[a], \text{array}[b]$) such that $a < b$ and $\text{array}[a] > \text{array}[b]$, i.e., the elements are in the wrong order. For example, the array

26

1 2 2 6 3 5 9 8

has three inversions: (6,3), (6,5) and (9,8). The number of inversions indicates how much work is needed to sort the array. An array is completely sorted when there are no inversions. On the other hand, if the array elements are in the reverse order, the number of inversions is the largest possible:

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

Swapping a pair of consecutive elements that are in the wrong order removes exactly one inversion from the array. Hence, if a sorting algorithm can only swap consecutive 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 algorithms that are not limited to swapping consecutive elements. One such algorithm is **merge sort**¹, which is based on recursion.

Merge sort sorts a subarray `array[a...b]` as follows:

1. If $a = b$, do not do anything, because the subarray is already sorted. 2.

Calculate the position of the middle element: $k = \lfloor (a + b) / 2 \rfloor$. 3.

Recursively sort the subarray `array[a...k]`.

4. Recursively sort the subarray `array[k + 1...b]`.

5. *Merge* the sorted subarrays `array[a...k]` and `array[k + 1...b]` into a sorted subarray `array[a...b]`.

Merge sort 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 `array[a...k]` and `array[k + 1...b]` is possible in linear time, because they are already sorted.

For example, consider sorting the following array:

1 3 6 2 8 2 5 9

The array will be divided into two subarrays as follows:

1 3 6 2 8 2 5 9

Then, the subarrays will be sorted recursively as follows:

1 2 3 6 2 5 8 9

¹According to [47], merge sort was invented by J. von Neumann in 1945. 27

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

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 considering 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:

$$x < y?$$

$$x < y? \quad x < y?$$

$$x < y? \quad x < y? \quad x < y? \quad x < y?$$

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 sort the array, a total of $n!$ ways. For this reason, the height of the tree must be at least

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

We get a lower bound for this sum by choosing the 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 a sorting algorithm in the worst case is at least $n \log n$.

Counting sort

The lower bound $n \log n$ does not apply to algorithms that do not compare array elements but use some other information. An example of such an algorithm is **counting sort** that sorts an array in $O(n)$ time assuming that every element in the array is an integer between $0 \dots c$ and $c = O(n)$.

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

28

For example, the array

1 3 6 9 9 3 5 9

corresponds to 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 at position 3 in the bookkeeping array is 2, because the element 3 appears 2 times in the original array.

Construction of the bookkeeping array takes $O(n)$ time. After this, the sorted array can be created in $O(n)$ time because the number of occurrences 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 small enough, so that the array elements can be used as indices in the bookkeeping array.

Sorting in C++

It is almost never a good idea to use a home-made 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. Second, 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 a vector in increasing order:

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

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

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

An ordinary array can be sorted as follows:

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

The following code sorts the string `s`:

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

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

Comparison operators

The function `sort` requires that a **comparison operator** is defined for the data type of the elements to be sorted. When sorting, this operator will be used whenever it is necessary 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 in alphabetical order.

Pairs (`pair`) are sorted primarily according to their first elements (`first`). However, if the first elements of two pairs are equal, they are sorted according to their second elements (`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).

In a similar way, tuples (`tuple`) are sorted primarily by the first element, secondarily by the second element, etc.²:

```
vector<tuple<int,int,int>> v;  
v.push_back({2,1,4});  
v.push_back({1,5,3});  
v.push_back({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` coordinates of a point. The comparison operator is defined so that the points are sorted primarily by the

²Note that in some older compilers, the function `make_tuple` has to be used to create a tuple instead of braces (for example, `make_tuple(2,1,4)` instead of `{2,1,4}`).

`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 functions

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

```
bool comp(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(), comp);
```

Binary search

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

```
for (int i = 0; i < n; i++) {  
    if (array[i] == x) {  
        // x found at index i  
    }  
}
```

The time complexity of this approach is $O(n)$, because in the worst case, it is necessary to check all elements of the array. If the order of the elements is arbitrary, 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 the search. The following **binary search** algorithm efficiently searches for an element in a sorted array in $O(\log n)$ time.

Method 1

The usual way to implement binary search resembles looking for a word in a dictionary. The search maintains an active region in the array, which initially contains all array elements. Then, a number of steps is performed, each of which halves the size of the region.

At each step, the search checks the middle element of the active region. If the middle element is the target element, the search terminates. Otherwise, the search recursively continues to the left or right half of the region, depending on the value of the middle element.

The above idea can be implemented as follows:

```
int a = 0, b = n-1;
while (a <= b) {
    int k = (a+b)/2;
    if (array[k] == x) {
        // x found at index k
    }
    if (array[k] > x) b = k-1;
    else a = k+1;
}
```

In this implementation, the active region is $a...b$, and initially the region is $0...n-1$. The algorithm halves the size of the region at each step, so the time complexity is $O(\log n)$.

Method 2

An alternative method to implement binary search is based on an efficient way to iterate through the elements of the array. The idea is to make jumps and slow the speed when we get closer to the target element.

The search goes through the array from left to 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 target element has been found or we know that it does not appear in the array.

The following code implements the above idea:

```
int k = 0;
for (int b = n/2; b >= 1; b /= 2) {
    while (k+b < n && array[k+b] <= x) k += b;
}
if (array[k] == x) {
    // x found at index k
}
```

During the search, the variable b contains the current jump length. 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.

C++ functions

The C++ standard library contains the following functions that are based on binary search and work in logarithmic time:

- `lower_bound` returns a pointer to the first array element whose value is at least x .
- `upper_bound` returns a pointer to the first array element whose value is larger than x .

- `equal_range` returns both above pointers.

The functions assume that the array is sorted. If there is no such element, the pointer points to the element after the last array element. For example, the following code finds out whether an array contains an element with value x :

```
auto k = lower_bound(array, array+n, x) - array;
if (k < n && array[k] == x) {
    // x found at index k
}
```

Then, the following code counts the number of elements whose value is x :

```
auto a = lower_bound(array, array+n, x);
auto b = upper_bound(array, array+n, x);
cout << b-a << "\n";
```

Using `equal_range`, the code becomes shorter:

```
auto r = equal_range(array, array+n, x);
cout << r.second-r.first << "\n";
```

Finding the smallest solution

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

x	0	1	\dots	$k-1$	k	$k+1$	\dots
<code>ok(x)</code>	false	false	\dots	false	true	true	\dots

Now, the value of 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 is $O(n \log z)$.

Finding the maximum value

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

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

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

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

Note that unlike in the ordinary binary search, here it is not allowed that consecutive 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 a computer. It is important to choose an appropriate 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 about more sophisticated data structures that are not available in the standard library.

Dynamic arrays

A **dynamic array** is an array whose size can be changed during the execution of the program. The most popular dynamic array in C++ is the `vector` structure, which can be used almost like an ordinary 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 an ordinary 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";  
}
```

35

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

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

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

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

The following code creates a vector with five elements:

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

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

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

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

The internal implementation of a vector uses an ordinary array. If the size of the vector increases and the array becomes too small, a new array is allocated and all the elements are moved to the new array. However, this does not happen often and the average time complexity of `push_back` is $O(1)$.

The string structure is also 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 position k and has length x , and the function `find(t)` finds the position of the first occurrence of a substring t .

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
```

Set structures

A **set** is a data structure that maintains a collection of elements. The basic operations of sets are element insertion, search and removal. The C++ standard library contains two set implementations: The structure `set` is based on a balanced binary tree and its operations work in $O(\log n)$ time. The structure `unordered_set`

uses hashing, and its operations work in $O(1)$ time on average.

The choice of which set implementation to use is often a matter of taste. The benefit of 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` can be more efficient.

The following code creates a set that contains integers, and shows some of the operations. The function `insert` adds an element to the set, the function `count` returns the number of occurrences of an element 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 sets is that all their 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 there. 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 otherwise work like `set` and `unordered_set` but they can contain multiple instances of an element. For example, in the following code all three instances of the number 5 are added to a multiset:

```

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

```

Map structures

A **map** is a generalized array that consists of key-value-pairs. While the keys in an ordinary array are always the consecutive integers $0, 1, \dots, n-1$, where n is the size of the array, the keys in a map can be of any data type and they do not have to be consecutive values.

The C++ standard library contains two map implementations that correspond to the set implementations: the structure `map` is based on a balanced binary tree and accessing elements takes $O(\log n)$ time, while the structure `unordered_map` uses hashing and accessing elements 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 the value of a key is requested but the map does not contain it, the key is automatically added to the map with a default value. For example, in the following code, the key "aybabbu" with value 0 is added to the map.

```

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

```

The function `count` checks if a key exists in a map:

```

if (m.count("aybabbu")) {
    // key exists
}

```

```
}
```

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

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

Iterators and ranges

Many functions in the C++ standard library operate with iterators. An **iterator** is a variable that points to an element in a data structure.

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

```
      { 3, 4, 6, 8, 12, 13, 14, 17 }  
                ↑ ↑  
            s.begin() s.end()
```

Note the asymmetry in the iterators: `s.begin()` points to an element in the data structure, while `s.end()` points outside the data structure. Thus, the range defined by the iterators is *half-open*.

Working with ranges

Iterators are used in C++ standard library functions that are given a range of elements in a data structure. 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 an ordinary array. In this case, the functions are given pointers to the array instead of iterators:

```
sort(a, a+n);
reverse(a, a+n);
random_shuffle(a, a+n);
```

Set iterators

Iterators are often used to access elements of a set. The following code creates an iterator `it` that points to the smallest element in a 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 using 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 the operators `++` (forward) and `--` (backward), meaning that the iterator moves to the next or previous element in the set. The following code prints all the elements in increasing order:

```
for (auto it = s.begin(); it != s.end(); it++) {
    cout << *it << "\n";
}
```

The following code prints the largest 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 does not contain `x`, the iterator will be `end`.

```
auto it = s.find(x);
if (it == s.end()) {
    // x is not found
}
```

The function `lower_bound(x)` returns an iterator to the smallest element in the set whose value is *at least* `x`, and the function `upper_bound(x)` returns an iterator to the smallest element in the set whose value is *larger than* `x`. In both functions, if such an element does not exist, the return value is `end`. These functions are not

supported by the `unordered_set` structure which does not maintain the order of the elements.

40

For example, the following code finds the element nearest to x :

```
auto it = s.lower_bound(x);
if (it == s.begin()) {
    cout << *it << "\n";
} else if (it == s.end()) {
    it--;
    cout << *it << "\n";
} else {
    int a = *it; it--;
    int b = *it;
    if (x-b < a-x) cout << b << "\n";
    else cout << a << "\n";
}
```

The code assumes that the set is not empty, and goes through all possible cases using an iterator `it`. First, the iterator points to the smallest element whose value is at least x . If `it` equals `begin`, the corresponding element is nearest to x . If `it` equals `end`, the largest element in the set is nearest to x . If none of the previous cases hold, the element nearest to x is either the element that corresponds to `it` or the previous element.

Other structures

Bitset

A **bitset** is an array whose each value is either 0 or 1. For example, the following code creates a `bitset` that contains 10 elements:

```
bitset<10> s;
s[1] = 1;
s[3] = 1;
s[4] = 1;
s[7] = 1;
cout << s[4] << "\n"; // 1
cout << s[5] << "\n"; // 0
```

The benefit of using bitsets is that they require less memory than ordinary arrays, because each element in a `bitset` only uses one bit of memory. For example, if n bits are stored in an `int` array, $32n$ bits of memory will be used, but a corresponding `bitset` only requires n bits of memory. In addition, the values of a `bitset` can be efficiently manipulated using bit operators, which makes it possible to optimize algorithms using bit sets.

The following code shows another way to create the above `bitset`:

```
bitset<10> s(string("0010011010")); // from right to
left cout << s[4] << "\n"; // 1
cout << s[5] << "\n"; // 0
```

41

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
```

Deque

A **deque** is a dynamic array whose size can be efficiently changed at both ends of the array. Like a vector, a deque provides the functions `push_back` and `pop_back`, but it also includes the functions `push_front` and `pop_front` which 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 that of a vector, and for this reason, a deque is slower than a vector. Still, both adding and removing elements take $O(1)$ time on average at both ends.

Stack

A **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
```

42

Queue

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

The following code shows how a queue can be used:

```
queue<int> q;
q.push(3);
q.push(2);
q.push(5);
cout << q.front(); // 3
q.pop();
cout << q.front(); // 2
```

Priority queue

A **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 or maximum element. Insertion and removal take $O(\log n)$ time, and retrieval takes $O(1)$ time.

While an ordered set efficiently supports all the operations of a priority queue, the benefit of 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 used in an ordered set.

By default, the elements in a 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 illustrates this:

```
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();
```

If we want to create a priority queue that supports finding and removing the smallest element, we can do it as follows:

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

43

Policy-based data structures

The g++ compiler also supports some data structures that are not part of the C++ standard library. Such structures are called *policy-based* data structures. To use these structures, the following lines must be added to the code:

```
#include <ext/pb_ds/assoc_container.hpp>  
using namespace __gnu_pbds;
```

After this, we can define a data structure `indexed_set` that is like `set` but can be indexed like an array. The definition for `int` values is as follows:

```
typedef tree<int, null_type, less<int>, rb_tree_tag,  
            tree_order_statistics_node_update> indexed_set;
```

Now we can create a set as follows:

```
indexed_set s;  
s.insert(2);  
s.insert(3);  
s.insert(7);  
s.insert(9);
```

The speciality of this set is that we have access to the indices that the elements would have in a sorted array. The function `find_by_order` returns an iterator to the element at a given position:

```
auto x = s.find_by_order(2);  
cout << *x << "\n"; // 7
```

And the function `order_of_key` returns the position of a given element:

```
cout << s.order_of_key(7) << "\n"; // 2
```

If the element does not appear in the set, we get the position that the element would have in the set:

```
cout << s.order_of_key(6) << "\n"; // 2  
cout << s.order_of_key(8) << "\n"; // 3
```

Both the functions work in logarithmic time.

Comparison to sorting

It is often 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 elements. Our task is to calculate the number of elements that belong

44

to both of the lists. For example, for the lists

$$A = [5, 2, 8, 9, 4] \text{ and } B = [3, 2, 9, 5],$$

the answer is 3 because the numbers 2, 5 and 9 belong to both of the lists. A straightforward solution to the problem is to go through all pairs of elements in $O(n^2)$ time, but next we will focus on more efficient algorithms.

Algorithm 1

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

Algorithm 2

It is not necessary 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 underlying data structure. The time complexity of the new algorithm is $O(n)$.

Algorithm 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 of the lists are random integers between $1 \dots 10^9$:

n	Algorithm 1	Algorithm 2	Algorithm 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

$$\begin{array}{r}
4 \cdot 10^6 \text{ s } 7.7 \text{ s } 1.7 \text{ s } 0.7 \text{ s} \\
5 \cdot 10^6 \text{ s } 10.0 \text{ s } 2.3 \text{ s } 0.9 \text{ s}
\end{array}$$

Algorithms 1 and 2 are equal except that they use different set structures. In this problem, this choice has an important effect on the running time, because Algorithm 2 is 4–5 times faster than Algorithm 1.

However, the most efficient algorithm is Algorithm 3 which uses sorting. It only uses half the time compared to Algorithm 2. Interestingly, the time complexity of both Algorithm 1 and Algorithm 3 is $O(n \log n)$, but despite this, Algorithm 3 is ten times faster. This can be explained by the fact that sorting is a

45

simple procedure and it is done only once at the beginning of Algorithm 3, and the rest of the algorithm works in linear time. On the other hand, Algorithm 1 maintains a complex balanced binary tree during the whole algorithm.

Chapter 5

Complete search

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

Complete search is a good technique if there is enough time 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, other techniques, such as greedy algorithms or dynamic programming, may be needed.

Generating subsets

We first consider the problem of generating all subsets of a set of n elements. For example, the subsets of $\{0,1,2\}$ are \emptyset , $\{0\}$, $\{1\}$, $\{2\}$, $\{0,1\}$, $\{0,2\}$, $\{1,2\}$ and $\{0,1,2\}$. There are two common methods to generate subsets: we can either perform a recursive search or exploit the bit representation of integers.

Method 1

An elegant way to go through all subsets of a set is to use recursion. The following function `search` generates the subsets of the set $\{0,1,\dots,n-1\}$. The function maintains a vector `subset` that will contain the elements of each subset. The search begins when the function is called with parameter 0.

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

47

When the function `search` is called with parameter k , it decides whether to include the element k in the subset or not, and in both cases, then calls itself with parameter $k+1$. However, if $k = n$, the function notices that all elements have been processed and a subset has been generated.

The following tree illustrates the function calls when $n = 3$. We can always choose either the left branch (k is not included in the subset) or the right branch (k is included in the subset).

search(0)

search(1) search(1)

search(2) search(2) search(2) search(2)

search(3) search(3) search(3) search(3) search(3) search(3) search(3) search(3) ; {2}
{1} {1,2} {0} {0,2} {0,1} {0,1,2}

Method 2

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

The usual convention is that the last bit corresponds to element 0, the second last bit corresponds to element 1, and so on. For example, the bit representation of 25 is 11001, which corresponds to the subset {0,3,4}.

The following code goes through the subsets of a set of n elements

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

The following code shows how we can find the elements of a subset that corresponds to a bit sequence. When processing each subset, the code builds a vector that contains the elements in the subset.

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

Generating permutations

Next we consider the problem of generating all permutations of a set of n elements. For example, the permutations of {0,1,2} are (0,1,2), (0,2,1), (1,0,2), (1,2,0), (2,0,1) and (2,1,0). Again, there are two approaches: we can either use recursion or go through the permutations iteratively.

Method 1

Like subsets, permutations can be generated using recursion. The following function `search` goes through the permutations of the set {0,1,..., $n - 1$ }. The function builds a vector permutation that contains the permutation, and the search begins when the function is called without parameters.

```

void search() {
    if (permutation.size() == n) {
        // process permutation
    } else {
        for (int i = 0; i < n; i++) {
            if (chosen[i]) continue;
            chosen[i] = true;
            permutation.push_back(i);
            search();
            chosen[i] = false;
            permutation.pop_back();
        }
    }
}

```

Each function call adds a new element to permutation. The array chosen indicates which elements are already included in the permutation. If the size of permutation equals the size of the set, a permutation has been generated.

Method 2

Another method for generating permutations is to begin with the permutation $\{0, 1, \dots, n - 1\}$ and repeatedly use a function that constructs the next permutation in increasing order. The C++ standard library contains the function next_permutation that can be used for this:

```

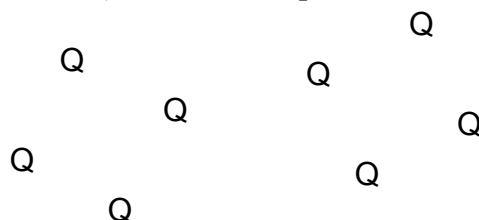
vector<int> permutation;
for (int i = 0; i < n; i++) {
    permutation.push_back(i);
}
do {
    // process permutation
} while (next_permutation(permutation.begin(), permutation.end()));

```

Backtracking

A **backtracking** algorithm begins with an empty solution and extends the solution step by step. The search recursively goes through all different ways how a solution can be constructed.

As an example, consider the problem of calculating the number of ways n queens can be placed on an $n \times n$ chessboard so that no two queens attack each other. For example, when $n = 4$, there are two possible solutions:



The problem can be solved using backtracking by placing queens to the board row by row. More precisely, exactly one queen will be placed on each row so that no queen attacks any of the queens placed before. A solution has been found when all n queens have been placed on the board.

For example, when $n = 4$, some partial solutions generated by the backtracking algorithm are as follows:

Q Q Q Q

Q Q Q Q
Q Q Q Q

illegal illegal illegal valid

At the bottom level, the three first configurations are illegal, because the queens attack each other. However, the fourth configuration is valid and it can be extended to a complete solution by placing two more queens to the board. There is only one way to place the two remaining queens.

The algorithm can be implemented as follows:

50

```
void search(int y) {
    if (y == n) {
        count++;
        return;
    }
    for (int x = 0; x < n; x++) {
        if (column[x] || diag1[x+y] || diag2[x-y+n-1]) continue;
        column[x] = diag1[x+y] = diag2[x-y+n-1] = 1;
        search(y+1);
        column[x] = diag1[x+y] = diag2[x-y+n-1] = 0;
    }
}
```

The search begins by calling `search(0)`. The size of the board is $n \times n$, and the code calculates the number of solutions to count.

The code assumes that the rows and columns of the board are numbered from 0 to $n-1$. When the function `search` is called with parameter y , it places a queen on row y and then calls itself with parameter $y+1$. Then, if $y = n$, a solution has been found and the variable `count` is increased by one.

The array `column` keeps track of columns that contain a queen, and the arrays `diag1` and `diag2` keep track of diagonals. It is not allowed to add another queen to a column or diagonal that already contains a queen. For example, the columns and diagonals of the 4×4 board are numbered as follows:

	0 1 2 3 1 2 3	3 4 5 6 2 3 4
0 1 2 3 0 1 2	4 2 3 4 5 3 4	5 1 2 3 4 0 1
3 0 1 2 3 0 1	5 6	2 3
2 3		
	column	diag1 diag2

Let $q(n)$ denote the number of ways to place n queens on an $n \times n$ chessboard. The above backtracking algorithm tells us that, for example, $q(8) = 92$. When n increases, the search quickly becomes slow, because the number of solutions increases exponentially. For example, calculating $q(16) = 14772512$ using the above algorithm already takes about a minute on a modern computer¹.

Pruning the search

We can often optimize backtracking by pruning the search tree. The idea is to add "intelligence" to the algorithm so that it will notice as soon as possible if a partial solution cannot be extended to a complete solution. Such optimizations can have a tremendous effect on the efficiency of the search.

¹There is no known way to efficiently calculate larger values of $q(n)$. The current record is $q(27) = 234907967154122528$, calculated in 2016 [55].

Let us consider the problem of calculating the number of paths in an $n \times n$ grid from the upper-left corner to the lower-right corner such that the path visits each square exactly once. For example, in a 7×7 grid, there are 111712 such paths. One of the paths is as follows:

We focus on the 7×7 case, because its level of difficulty is appropriate to our needs. We begin with a straightforward backtracking algorithm, and then optimize it step by step using observations of how the search can be pruned. After each optimization, we measure the running time of the algorithm and the number of recursive calls, so that we clearly see the effect of each optimization on the efficiency of the search.

Basic algorithm

The first version of the algorithm does not contain any optimizations. We simply use backtracking to generate all possible paths from the upper-left corner to the lower-right corner and count the number of such paths.

- running time: 483 seconds
- number of recursive calls: 76 billion

Optimization 1

In any solution, we first move one step down or 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:

Hence, we can decide that we always first move one step down (or right), and finally multiply the number of solutions by two.

- running time: 244 seconds
- number of recursive calls: 38 billion

52

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 path:

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

- running time: 119 seconds
- number of recursive calls: 20 billion

Optimization 3

If the path touches a wall and can turn either left or right, the grid splits into two parts that contain unvisited squares. For example, in the following situation, the path can turn either left or right:

In this case, we cannot visit all squares anymore, so we can terminate the search. This optimization is very useful:

- running time: 1.8 seconds
- number of recursive calls: 221 million

Optimization 4

The idea of Optimization 3 can be generalized: if the path cannot continue forward but can turn either left or right, the grid splits into two parts that both contain unvisited squares. For example, consider the following path:

53

It is clear that we cannot visit all squares anymore, so we can terminate the search. After this optimization, the search is very efficient:

- running time: 0.6 seconds
- number of recursive calls: 69 million

Now is a good moment to stop optimizing the algorithm and see what we have achieved. 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 observations can effectively prune the search. Especially useful are optimizations that occur during the first steps of the algorithm, i.e., at the top of the search tree.

Meet in the middle

Meet in the middle is a technique where the search space is divided into two parts of about equal size. A separate search is performed for both of the parts, and finally

the results of the searches are combined.

The technique can be used if there is an efficient way to combine the results of the searches. In such a situation, 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 a number x , and we want to find out if it is possible to choose some numbers from the list so that their sum 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 $x = 10$ for the same list, it is not possible to form the sum.

A simple algorithm to the problem is to go through all subsets of the elements and check if the sum of any of the subsets is x . The running time of such an algorithm is $O(2^n)$, because there are 2^n subsets. However, using the meet in the middle technique, we can achieve a more efficient $O(2^{n/2})$ time algorithm². Note that $O(2^n)$ and $O(2^{n/2})$ are different complexities because $2^{n/2}$ equals $\sqrt{2^n}$.

²This idea was introduced in 1974 by E. Horowitz and S. Sahni [39].

54

The idea is to divide the list into two lists A and B such that both lists contain about half of the numbers. The first search generates all subsets of A and stores their sums to a list S_A . Correspondingly, the second search creates a list S_B from B . After this, it suffices to check if it is possible to choose one element from S_A and another element from S_B such that their sum is x . This is possible exactly when there is a way to form the sum x using the numbers of the original list.

For example, suppose 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 lists $S_A = [0,2,4,6]$ and $S_B = [0,5,9,14]$. In this case, the sum $x = 15$ is possible to form, because S_A contains the sum 6, S_B contains the sum 9, and $6+9 = 15$. This corresponds to the solution $[2,4,9]$.

We can implement the algorithm so that its time complexity is $O(2^{n/2})$. First, we generate *sorted* lists S_A and S_B , which can be done in $O(2^{n/2})$ time using a merge-like technique. After this, since the lists are sorted, we can check in $O(2^{n/2})$ time if the sum x can be created from S_A and S_B .

Chapter 6

Greedy algorithms

A **greedy algorithm** constructs a solution to the problem by always making a choice that looks the best at the moment. A greedy algorithm never takes back its choices, but directly constructs the final solution. For this reason, greedy algorithms are usually very efficient.

The difficulty in designing greedy algorithms is to find a greedy strategy that always produces an optimal solution to the problem. The locally optimal choices in a greedy algorithm should also be globally optimal. It is often difficult to argue that a greedy algorithm works.

Coin problem

As a first example, we consider a problem where we are given a set of coins and our task is to form a sum of money n using the coins. The values of the coins are $\text{coins} = \{c_1, c_2, \dots, c_k\}$, and each coin can be used as many times we want. What is the minimum number of coins needed?

For example, if the coins are the euro coins (in cents)

$$\{1, 2, 5, 10, 20, 50, 100, 200\}$$

and $n = 520$, we need at least four coins. The optimal solution is to select coins $200+200+100+20$ whose sum is 520.

Greedy algorithm

A simple greedy algorithm to the problem always selects the largest possible coin, until the required sum of money has been constructed. This algorithm works in the example case, because we first select two 200 cent coins, then one 100 cent coin and finally one 20 cent coin. But does this algorithm always work?

It turns out that if the coins are the euro coins, the greedy algorithm *always* works, i.e., it always produces a solution with the fewest possible number of coins. The correctness of the algorithm can be shown as follows:

First, each coin 1, 5, 10, 50 and 100 appears at most once in an optimal solution, because if the solution would contain two such coins, we could replace

57

them by one coin and obtain a better solution. For example, if the solution would contain coins $5+5$, we could replace them by coin 10.

In the same way, coins 2 and 20 appear at most twice in an optimal solution, because we could replace coins $2+2+2$ by coins $5+1$ and coins $20+20+20$ by coins $50+10$. Moreover, an optimal solution cannot contain coins $2+2+1$ or $20+20+10$, because we could replace them by coins 5 and 50.

Using these observations, we can show for each coin x that it is not possible to optimally construct a sum x or any larger sum by only using coins that are smaller than x . For example, if $x = 100$, the largest optimal sum using the smaller coins is $50+20+20+5+2+2 = 99$. Thus, the greedy algorithm that always selects the largest coin produces the optimal solution.

This example shows that it can be difficult to argue that a greedy algorithm works, even if the algorithm itself is simple.

General case

In the general case, the coin set can contain any coins and the greedy algorithm *does not* necessarily produce an optimal solution.

We can prove that a greedy algorithm does not work by showing a counterexample where the algorithm gives a wrong answer. In this problem we can easily find a counterexample: if the coins are $\{1, 3, 4\}$ and the target sum is 6, the greedy algorithm produces the solution $4+1+1$ while the optimal solution is $3+3$.

It is not known if the general coin problem can be solved using any greedy algorithm¹. However, as we will see in Chapter 7, in some cases, the general problem can be efficiently solved using a dynamic programming algorithm that always gives the correct answer.

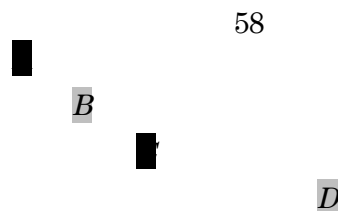
Scheduling

Many scheduling problems can be solved using greedy algorithms. A classic problem is as follows: Given n events with their starting and ending times, find a schedule that includes as many events as possible. It is not possible to select an event partially. For example, consider the following events:

event	starting time	ending time
<i>A</i>	1	3
<i>B</i>	2	5
<i>C</i>	3	9
<i>D</i>	6	8

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

¹However, it is possible to *check* in polynomial time if the greedy algorithm presented in this chapter works for a given set of coins [53].



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

Algorithm 1

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

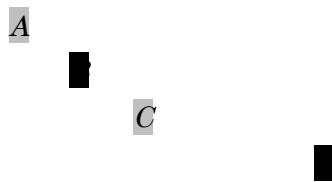


However, selecting short events is not always a correct strategy. For example, the algorithm fails in the following case:

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

Algorithm 2

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



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

If we select the first event, it is not possible to select any other events. However, it would be possible to select the other two events.

59

Algorithm 3

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



It turns out that this algorithm *always* produces an optimal solution. The reason for this is that it is always an optimal choice to first select an event that ends as early as possible. After this, it is an optimal choice to select the next event using the same strategy, etc., until we cannot select any more events.

One way to argue that the algorithm works is to consider what happens if we first select an event that ends later than the event that ends as early as possible. Now, we will have at most an equal number of choices how we can select the next event. Hence, selecting an event that ends later can never yield a better solution, and the greedy algorithm is correct.

Tasks and deadlines

Let us now consider a problem where we are given n tasks with durations and deadlines and our task is to choose an order to perform the tasks. For each task, we earn $d - x$ points where d is the task's deadline and x is the moment when we finish the task. What is the largest possible total score we can obtain?

For example, suppose that the tasks are as follows:

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

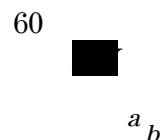
In this case, an optimal schedule for the tasks is as follows:

0 5 10



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

Surprisingly, the optimal solution to the problem does not depend on the deadlines at all, but a correct greedy strategy is to simply perform the tasks *sorted by their durations* in increasing order. The reason for this is that if we ever perform two tasks one after another such that the first task takes longer than the second task, we can obtain a better solution if we swap the tasks. For example, consider the following schedule:



Here $a > b$, so we should swap the tasks:



Now X gives b points less and Y gives a points more, so the total score increases by $a - b > 0$. In an optimal solution, for any two consecutive tasks, it must hold that the shorter task comes before the longer task. Thus, the tasks must be performed sorted by their durations.

Minimizing sums

We next consider a problem where we are given n numbers a_1, a_2, \dots, a_n and our task is to find a value x that minimizes the sum

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

We focus on the cases $c = 1$ and $c = 2$.

Case $c = 1$

In this case, we should minimize the sum

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

For example, if the numbers are [1,2,9,2,6], the best solution is to select $x = 2$ which produces the sum

$$|1-2| + |2-2| + |9-2| + |2-2| + |6-2| = 12.$$

In the general case, the best choice for x is the *median* of the numbers, i.e., the middle number after sorting. For example, the list [1,2,9,2,6] becomes [1,2,2,6,9] after sorting, so the median is 2.

The median is an optimal choice, because if x is smaller than the median, the sum becomes smaller by increasing x , and if x is larger than the median, the sum becomes smaller by decreasing x . Hence, the optimal solution is that x is the median. If n is even and there are two medians, both medians and all values between them are optimal choices.

Case $c = 2$

In this case, we should minimize the sum

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

61

For example, if the numbers are [1,2,9,2,6], the best solution is to select $x = 4$ which produces the sum

$$(1-4)^2 + (2-4)^2 + (9-4)^2 + (2-4)^2 + (6-4)^2 = 46.$$

In the general case, the best choice for x is the *average* of the numbers. In the example the average is $(1 + 2 + 9 + 2 + 6)/5 = 4$. This result can be derived by presenting the sum as follows:

$$nx^2 - 2x(a_1 + a_2 + \cdots + a_n) + (a_1^2 + a_2^2 + \cdots + a_n^2)$$

The last part does not depend on x , so we can ignore it. The remaining parts form a function $nx^2 - 2xs$ where $s = a_1 + a_2 + \cdots + a_n$. This is a parabola opening upwards with roots $x = 0$ and $x = 2s/n$, and the minimum value is the average of the roots $x = s/n$, i.e., the average of the numbers a_1, a_2, \dots, a_n .

Data compression

A **binary code** assigns for each character of a string a **codeword** that consists of bits. We can *compress* the string using the binary code by replacing each character by the corresponding codeword. For example, the following binary code assigns codewords for characters A–D:

character codeword

A 00

B	01
C	10
D	11

This is a **constant-length** code which means that the length of each codeword is the same. For example, we can compress the string AABACDACA as follows:

000001001011001000

Using this code, the length of the compressed string is 18 bits. However, we can compress the string better if we use a **variable-length** code where codewords may have different lengths. Then we can give short codewords for characters that appear often and long codewords for characters that appear rarely. It turns out that an **optimal** code for the above string is as follows:

character	codeword
A	0
B	110
C	10
D	111

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

001100101110100,

62

so only 15 bits are needed instead of 18 bits. Thus, thanks to a better code it was possible to save 3 bits in the compressed string.

We require that no codeword is a prefix of another codeword. For example, it is not allowed that a code would contain both codewords 10 and 1011. The reason for this is that we want to be able to generate the original string from the compressed string. If a codeword could be a prefix of another codeword, this would not always be possible. For example, the following code is *not* valid:

character	codeword
A	10
B	11
C	1011
D	111

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

Huffman coding

Huffman coding² is a greedy algorithm that constructs an optimal code for compressing a given string. The algorithm builds a binary tree based on the frequencies of the characters in the string, and each character's codeword can be read by following a path from the root to the corresponding node. A move to the left corresponds to bit 0, and a move to the right corresponds to bit 1.

Initially, each character of the string is represented by a node whose weight is the number of times the character occurs in the string. Then at each step two nodes with minimum weights are combined by creating a new node whose weight is the

sum of the weights of the original nodes. The process continues until all nodes have been combined.

Next we will see how Huffman coding creates the optimal code for the string AABACDACA. Initially, there are four nodes that correspond to the characters of the string:

5 1 2 1
A B C D

The node that represents character A has weight 5 because character A appears 5 times in the string. The other weights have been calculated in the same way. The first step is to combine the nodes that correspond to characters B and D, both with weight 1. The result is:

2
0 1
5 2 1 1
A C B D

²D. A. Huffman discovered this method when solving a university course assignment and published the algorithm in 1952 [40].

63

After this, the nodes with weight 2 are combined:

4
0 1
2 C 1 1 B D
2
5 A 0 1

Finally, the two remaining nodes are combined:

9
0 1
5 A
4 2
0 1 2
C 1 1 B D
0 1

Now all nodes are in the tree, so the code is ready. The following codewords can be read from the tree:

character	codeword
A	0
B	110
C	10
D	111

Chapter 7

Dynamic programming

Dynamic programming is a technique that combines the correctness of complete search and the efficiency of greedy algorithms. Dynamic programming can be applied if the problem can be divided into overlapping subproblems that can be solved independently.

There are two uses for dynamic programming:

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

We will first see how dynamic programming can be used to find an optimal solution, and then we will use the same idea for counting the solutions. Understanding dynamic programming is a milestone in every competitive programmer's career. While the basic idea is simple, the challenge is how to apply dynamic programming to different problems. This chapter introduces a set of classic problems that are a good starting point.

Coin problem

We first focus on a problem that we have already seen in Chapter 6: Given a set of coin values $\text{coins} = \{c_1, c_2, \dots, c_k\}$ and a target sum of money n , our task is to form the sum n using as few coins as possible.

In Chapter 6, we solved the problem using a greedy algorithm that always chooses the largest possible coin. The greedy algorithm works, for example, when the coins are the euro coins, but in the general case the greedy algorithm does not necessarily produce an optimal solution.

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

65

Recursive formulation

The idea in dynamic programming is to formulate the problem recursively so that the solution to the problem can be calculated from solutions to smaller subproblems. In the coin problem, a natural recursive problem is as follows: what is the smallest number of coins required to form a sum x ?

Let $\text{solve}(x)$ denote the minimum number of coins required for a sum x . The values of the function depend on the values of the coins. For example, if $\text{coins} = \{1, 3, 4\}$, the first values of the function are as follows:

```
solve(0) = 0
solve(1) = 1
solve(2) = 2
solve(3) = 1
solve(4) = 1
solve(5) = 2
solve(6) = 2
solve(7) = 2
solve(8) = 2
solve(9) = 3
solve(10) = 3
```

For example, $\text{solve}(10) = 3$, because at least 3 coins are needed to form the sum 10. The optimal solution is $3+3+4 = 10$.

The essential property of solve is that its values can be recursively calculated from its smaller values. The idea is to focus on the *first* coin that we choose for the sum. For example, in the above scenario, the first coin can be either 1, 3 or 4. If we first choose coin 1, the remaining task is to form the sum 9 using the minimum number of coins, which is a subproblem of the original problem. Of course, the same applies to coins 3 and 4. Thus, we can use the following recursive formula to calculate the minimum number of coins:

$$\begin{aligned} \text{solve}(x) = \min(&\text{solve}(x-1)+1, \\ &\text{solve}(x-3)+1, \\ &\text{solve}(x-4)+1). \end{aligned}$$

The base case of the recursion is $\text{solve}(0) = 0$, because no coins are needed to form an empty sum. For example,

$$\text{solve}(10) = \text{solve}(7)+1 = \text{solve}(4)+2 = \text{solve}(0)+3 = 3.$$

Now we are ready to give a general recursive function that calculates the minimum number of coins needed to form a sum x :

$$\text{solve}(x) = \begin{cases} \infty & x < 0 \\ 0 & x = 0 \\ \min_{c \in \text{coins}} \text{solve}(x-c)+1 & x > 0 \end{cases}$$

First, if $x < 0$, the value is ∞ , because it is impossible to form a negative sum of money. Then, if $x = 0$, the value is 0, because no coins are needed to form an

66

empty sum. Finally, if $x > 0$, the variable c goes through all possibilities how to choose the first coin of the sum.

Once a recursive function that solves the problem has been found, we can directly implement a solution in C++ (the constant `INF` denotes infinity):

```
int solve(int x) {
    if (x < 0) return INF;
    if (x == 0) return 0;
    int best = INF;
    for (auto c : coins) {
        best = min(best, solve(x-c)+1);
    }
    return best;
}
```

Still, this function is not efficient, because there may be an exponential number of ways to construct the sum. However, next we will see how to make the function efficient using a technique called memoization.

Using memoization

The idea of dynamic programming is to use **memoization** to efficiently calculate values of a recursive function. This means that the values of the function are stored in an array after calculating them. For each parameter, the value of the function is calculated recursively only once, and after this, the value can be directly retrieved from the array.

In this problem, we use arrays

```
bool ready[N];
int value[N];
```

where `ready[x]` indicates whether the value of `solve(x)` has been calculated, and if it is, `value[x]` contains this value. The constant N has been chosen so that all required values fit in the arrays.

Now the function can be efficiently implemented as follows:

```
int solve(int x) {
    if (x < 0) return INF;
    if (x == 0) return 0;
    if (ready[x]) return value[x];
    int best = INF;
    for (auto c : coins) {
        best = min(best, solve(x-c)+1);
    }
    value[x] = best;
    ready[x] = true;
    return best;
}
```

67

The function handles the base cases $x < 0$ and $x = 0$ as previously. Then the function checks from `ready[x]` if `solve(x)` has already been stored in `value[x]`, and if it is, the function directly returns it. Otherwise the function calculates the value of `solve(x)` recursively and stores it in `value[x]`.

This function works efficiently, because the answer for each parameter x is calculated recursively only once. After a value of `solve(x)` has been stored in `value[x]`, it can be efficiently retrieved whenever the function will be called again with the parameter x . The time complexity of the algorithm is $O(nk)$, where n is the target sum and k is the number of coins.

Note that we can also *iteratively* construct the array `value` using a loop that simply calculates all the values of `solve` for parameters $0 \dots n$:

```
value[0] = 0;
for (int x = 1; x <= n; x++) {
    value[x] = INF;
    for (auto c : coins) {
        if (x-c >= 0) {
            value[x] = min(value[x], value[x-c]+1);
        }
    }
}
```

In fact, most competitive programmers prefer this implementation, because it is shorter and has lower constant factors. From now on, we also use iterative implementations in our examples. Still, it is often easier to think about dynamic programming solutions in terms of recursive functions.

Constructing a solution

Sometimes we are asked both to find the value of an optimal solution and to give an example how such a solution can be constructed. In the coin problem, for example, we can declare another array that indicates for each sum of money the first coin in an optimal solution:

```
int first[N];
```

Then, we can modify the algorithm as follows:

```
value[0] = 0;
for (int x = 1; x <= n; x++) {
    value[x] = INF;
    for (auto c : coins) {
        if (x-c >= 0 && value[x-c]+1 < value[x]) {
            value[x] = value[x-c]+1;
            first[x] = c;
        }
    }
}
```

68

After this, the following code can be used to print the coins that appear in an optimal solution for the sum n :

```
while (n > 0) {
    cout << first[n] << "\n";
    n -= first[n];
}
```

Counting the number of solutions

Let us now consider another version of the coin problem where our task is to calculate the total number of ways to produce a sum x using the coins. For example, if $\text{coins} = \{1,3,4\}$ and $x = 5$, there are a total of 6 ways:

- 3+1+1 • 1+4
- 1+1+1+1+1 • 1+1+3
- 4+1
- 1+3+1

Again, we can solve the problem recursively. Let $\text{solve}(x)$ denote the number of ways we can form the sum x . For example, if $\text{coins} = \{1,3,4\}$, then $\text{solve}(5) = 6$ and the recursive formula is

$$\begin{aligned} \text{solve}(x) = & \text{solve}(x-1) + \\ & \text{solve}(x-3) + \\ & \text{solve}(x-4). \end{aligned}$$

Then, the general recursive function is as follows:

0

$$\text{solve}(x) = \begin{cases} 0 & x < 0 \\ 1 & x = 0 \\ \sum_{c \in \text{coins}} \text{solve}(x-c) & x > 0 \end{cases}$$

If $x < 0$, the value is 0, because there are no solutions. If $x = 0$, the value is 1, because there is only one way to form an empty sum. Otherwise we calculate the

sum of all values of the form $\text{solve}(x - c)$ where c is in coins.

The following code constructs an array `count` such that `count[x]` equals the value of `solve(x)` for $0 \leq x \leq n$:

```
count[0] = 1;
for (int x = 1; x <= n; x++) {
    for (auto c : coins) {
        if (x - c >= 0) {
            count[x] += count[x - c];
        }
    }
}
```

69

Often the number of solutions is so large that it is not required to calculate the exact number but it is enough to give the answer modulo m where, for example, $m = 10^9 + 7$. This can be done by changing the code so that all calculations are done modulo m . In the above code, it suffices to add the line

```
count[x] %= m;
```

after the line

```
count[x] += count[x - c];
```

Now we have discussed all basic ideas of dynamic programming. Since dynamic programming can be used in many different situations, we will now go through a set of problems that show further examples about the possibilities of dynamic programming.

Longest increasing subsequence

Our first problem is to find the **longest increasing subsequence** in an array of n elements. This is a maximum-length sequence of array elements that goes from left to right, and each element in the sequence is larger than the previous element. For example, in the array

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

the longest increasing subsequence contains 4 elements:

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

Let $\text{length}(k)$ denote the length of the longest increasing subsequence that ends at position k . Thus, if we calculate all values of $\text{length}(k)$ where $0 \leq k \leq n-1$, we will find out the length of the longest increasing subsequence. For example, the values

of the function for the above array are as follows:

length(0) = 1
length(1) = 1
length(2) = 2
length(3) = 1
length(4) = 3
length(5) = 2
length(6) = 4
length(7) = 2

For example, $\text{length}(6) = 4$, because the longest increasing subsequence that ends at position 6 consists of 4 elements.