Chapter 2

Time complexity

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

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

Calculation rules

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

Loops

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

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

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

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

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

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 $\lceil n/2 \rceil$ times, but the time complexity of each code is $O(n)$.

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

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

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

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

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

Phases

If the 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
}
```

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 $\texttt{f}(n)$ 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
$\cdots$	$\cdots$
$g(1)$	2^{n-1}

Based on this, the time complexity is

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

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 known[1].

[1] A 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 array[2]. 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

[2] J. Bentley's book *Programming Pearls* [8] made the problem popular.

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 $O(n^2)$.

Algorithm 3

Surprisingly, it is possible to solve the problem in $O(n)$ time[3], 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

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

The comparison shows that all algorithms are efficient when the input size is small, but larger inputs bring out remarkable differences in the running times of the algorithms. Algorithm 1 becomes slow when $n = 10^4$, and Algorithm 2 becomes slow when $n = 10^5$. Only Algorithm 3 is able to process even the largest inputs instantly.