

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 $\lceil n/2 \rceil$ 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 $f(n)$ 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

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

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].