Efficiency

3

The efficiency of algorithms plays a central role in competitive programming. In this chapter, we learn tools that make it easier to design efficient algorithms.

Section 3.1 introduces the concept of time complexity, which allows us to estimate running times of algorithms without implementing them. The time complexity of an algorithm shows how quickly its running time increases when the size of the input grows.

Section 3.2 presents two example problems which can be solved in many ways. In both problems, we can easily design a slow brute force solution, but it turns out that we can also create much more efficient algorithms.

3.1 Time Complexity

The *time complexity* of an algorithm estimates how much time the algorithm will use for a given input. By calculating the time complexity, we can often find out whether the algorithm is fast enough for solving a problem—without implementing it.

A time complexity 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.

3.1.1 Calculation Rules

If a code consists of single commands, its time complexity is $O(1)$. For example, the time complexity of the following code is $O(1)$.

A. Laaksonen, *Guide to Competitive Programming*, Undergraduate
Topics in Computer Science, https://doi.org/10.1007/978-3-319-72547-5_3

```
a++;
b++;
c = a+b;
```

The time complexity of a loop estimates the number of times the code inside the loop is executed. For example, the time complexity of the following code is $O(n)$, because the code inside the loop is executed n times. We assume that "..." denotes a code whose time complexity is $O(1)$.

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

Then, 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++) {
        ...
    }
}
```

In general, if there are k nested loops and each loop goes through n values, the time complexity is $O(n^k)$.

A time complexity does not tell us the exact number of times the code inside a loop is executed, because it only shows the order of growth and ignores the constant factors. 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++) {
    ...
}
```

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

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

As another example, the time complexity of the following code is $O(n^2)$, because the code inside the loop is executed $1 + 2 + \ldots + n = \frac{1}{2}(n^2 + n)$ times.

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

If an 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 the bottleneck of the algorithm. 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++) {
    ...
}
for (int i = 1; i <= n; i++) {
    for (int j = 1; j <= n; j++) {
        ...
    }
}
for (int i = 1; i <= n; i++) {
    ...
}
```

Sometimes the time complexity depends on several factors, and 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++) {
        ...
    }
}
```

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 $O(1)$, so 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);
}
```

What happens when the function is called with a parameter n? First, there are two calls with parameter $n-1$, then four calls with parameter $n-2$, then eight calls with parameter $n-3$, and so on. In general, there will be 2^k calls with parameter $n-k$ where $k = 0, 1, \ldots, n-1$. Thus, the time complexity is

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

3.1.2 Common Time Complexities

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. Note that the base of the logarithm is not shown in the time complexity.

$O(\sqrt{n})$ 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 n elements can be divided into $O(\sqrt{n})$ blocks of $O(\sqrt{n})$ elements.

$O(n)$ 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 can process large inputs.

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.

3.1.3 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 solving a problem. The starting point for estimations is the fact that a modern computer can perform some hundreds of millions of simple 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. However, if the time complexity is $O(n \log n)$, there will be only about $10^5 \log 10^5 \approx 1.6 \cdot 10^6$ operations, and the algorithm will surely fit the time limit.

On the other hand, given the input size, we can try to *guess* the required time complexity of the algorithm that solves the problem. Table 3.1 contains some useful estimates assuming a time limit of one second.

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, which has an important effect on the actual running time of the algorithm.

Table 3.1 Estimating time complexity from input size

Input size	Expected 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 $O(\log n)$

3.1.4 Formal Definitions

What does it *exactly* mean that an algorithm works in $O(f(n))$ time? It means that there are constants c and n_0 such that the algorithm performs *at most* $cf(n)$ operations for all inputs where $n \geq n_0$. Thus, the O notation gives an *upper bound* for the running time of the algorithm for sufficiently large inputs.

For example, it is technically correct to say that the time complexity of the following algorithm is $O(n^2)$.

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

However, a better bound is $O(n)$, and it would be very misleading to give the bound $O(n^2)$, because everybody actually assumes that the O notation is used to give an accurate estimate of the time complexity.

There are also two other common notations. The Ω notation gives a *lower bound* for the running time of an algorithm. The time complexity of an algorithm is $\Omega(f(n))$, if there are constants c and n_0 such that the algorithm performs *at least* $cf(n)$ operations for all inputs where $n \geq n_0$. Finally, the Θ notation gives an *exact bound*: the time complexity of an algorithm is $\Theta(f(n))$ if it is both $O(f(n))$ and $\Omega(f(n))$. For example, since the time complexity of the above algorithm is both $O(n)$ and $\Omega(n)$, it is also $\Theta(n)$.

We can use the above notations in many situations, not only for referring to time complexities of algorithms. For example, we might say that an array contains $O(n)$ values, or that an algorithm consists of $O(\log n)$ rounds.

3.2 Examples

In this section we discuss two algorithm design problems that can be solved in several different ways. We start with simple brute force algorithms, and then create more efficient solutions by using various algorithm design ideas.

3.2.1 Maximum Subarray Sum

Given an array of n numbers, our first task is to calculate the *maximum subarray sum*, i.e., the largest possible sum of a sequence of consecutive values in the array. The problem is interesting when there may be negative values in the array. For example, Fig. 3.1 shows an array and its maximum-sum subarray.

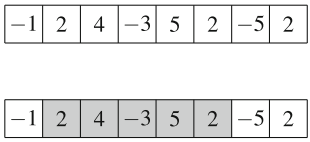

Fig. 3.1 The maximum-sum subarray of this array is [2, 4, −3, 5, 2], whose sum is 10

$O(n^3)$ **Time Solution** 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.

$O(n^2)$ **Time Solution** It is easy to make the algorithm 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 $O(n^2)$.

$O(n)$ **Time Solution** It turns out that it is possible to solve the problem in $O(n)$ time, 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 to 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 How efficient are the above algorithms in practice? Table 3.2 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, and the time needed for reading the input was not measured.

The comparison shows that all algorithms work quickly when the input size is small, but larger inputs bring out remarkable differences in the running times. The $O(n^3)$ algorithm becomes slow when $n = 10^4$, and the $O(n^2)$ algorithm becomes slow when $n = 10^5$. Only the $O(n)$ algorithm is able to process even the largest inputs instantly.

Table 3.2 Comparing running times of the maximum subarray sum algorithms

Array size n	$O(n^3)$ (s)	$O(n^2)$ (s)	$O(n)$ (s)
10^2	0.0	0.0	0.0
10^3	0.1	0.0	0.0
10^4	>10.0	0.1	0.0
10^5	>10.0	5.3	0.0
10^6	>10.0	>10.0	0.0
10^7	>10.0	>10.0	0.0

3.2.2 Two Queens Problem

Given an $n \times n$ chessboard, our next problem is to count the number of ways we can place *two* queens on the board in such a way that they do not attack each other. For example, as Fig. 3.2 shows, there are eight ways to place two queens on the 3×3 board. Let $q(n)$ denote the number of valid combinations for an $n \times n$ board. For example, $q(3) = 8$, and Table 3.3 shows the values of $q(n)$ for $1 \leq n \leq 10$.

To start with, a simple way to solve the problem is to go through all possible ways to place two queens on the board and count the combinations where the queens do not attack each other. Such an algorithm works in $O(n^4)$ time, because there are n^2 ways to choose the position of the first queen, and for each such position, there are $n^2 - 1$ ways to choose the position of the second queen.

Since the number of combinations grows fast, an algorithm that counts combinations one by one will certainly be too slow for processing larger values of n. Thus, to create an efficient algorithm, we need to find a way to count combinations in *groups*. One useful observation is that it is quite easy to calculate the number of squares that a single queen attacks (Fig. 3.3). First, it always attacks $n - 1$ squares horizontally and $n - 1$ squares vertically. Then, for both diagonals, it attacks $d - 1$ squares where d is the number of squares on the diagonal. Using this information, we can calculate

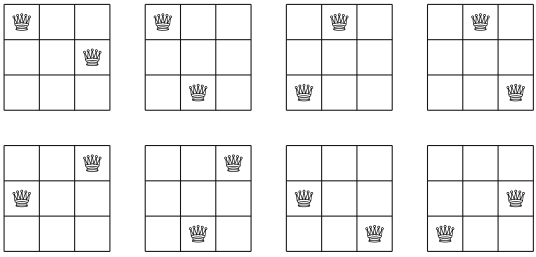

Fig. 3.2 All possible ways to place two non-attacking queens on the 3×3 chessboard

Table 3.3 First values of the function $q(n)$: the number of ways to place two non-attacking queens on an $n \times n$ chessboard

Board size n	Number of ways $q(n)$
1	0
2	0
3	8
4	44
5	140
6	340
7	700
8	1288
9	2184
10	3480

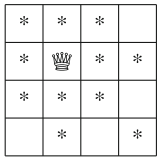

Fig. 3.3 The queen attacks all squares marked with "*" on the board

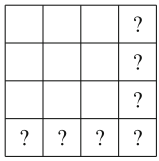

Fig. 3.4 Possible positions for queens on the last row and column

in $O(1)$ time the number of squares where the other queen can be placed, which yields an $O(n^2)$ time algorithm.

Another way to approach the problem is to try to formulate a recursive function that counts the number of combinations. The question is: if we know the value of $q(n)$, how can we use it to calculate the value of $q(n+1)$?

To get a recursive solution, we may focus on the last row and last column of the $n \times n$ board (Fig. 3.4). First, if there are no queens on the last row or column, the number of combinations is simply $q(n-1)$. Then, there are $2n-1$ positions for a queen on the last row or column. It attacks $3(n-1)$ squares, so there are $n^2 - 3(n-1) - 1$ positions for the other queen. Finally, there are $(n-1)(n-2)$ combinations where both queens are on the last row or column. Since we counted those combinations twice, we have to remove this number from the result. By combining all this, we get a recursive formula

$$\begin{aligned} q(n) &= q(n-1) + (2n-1)(n^2 - 3(n-1) - 1) - (n-1)(n-2) \\ &= q(n-1) + 2(n-1)^2(n-2), \end{aligned}$$

which provides an $O(n)$ solution to the problem.

Finally, it turns out that there is also a closed-form formula

$$q(n) = \frac{n^4}{2} - \frac{5n^3}{3} + \frac{3n^2}{2} - \frac{n}{3},$$

which can be proved using induction and the recursive formula. Using this formula, we can solve the problem in $O(1)$ time.