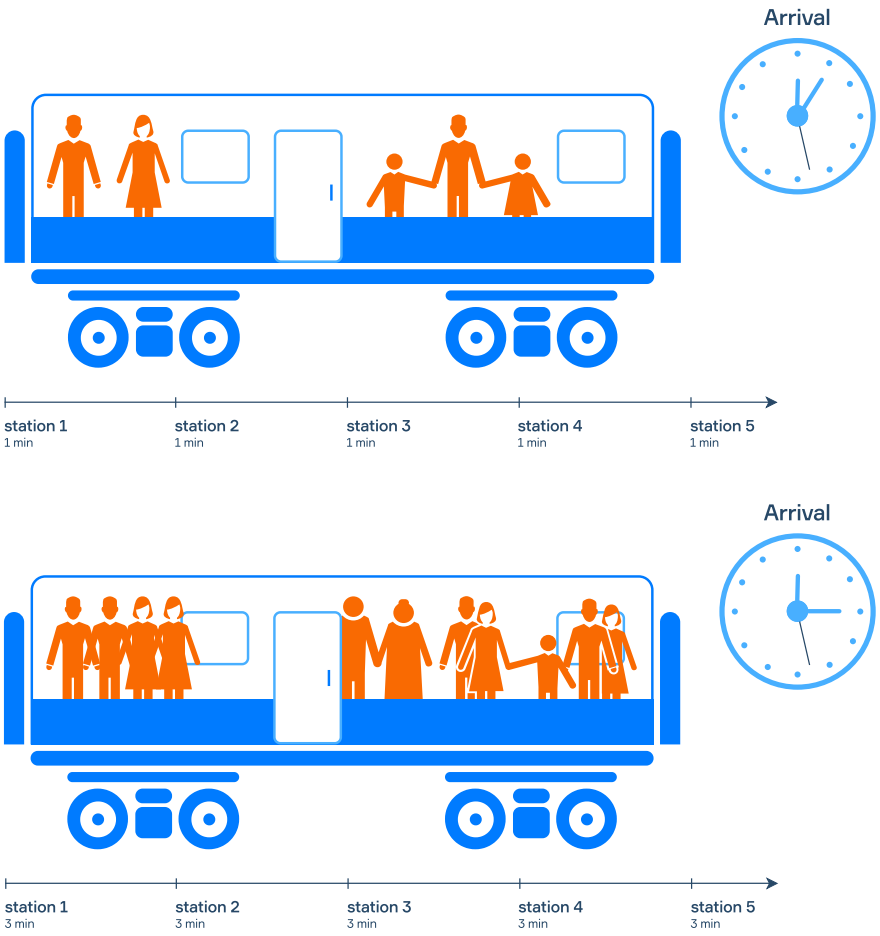
Suppose you need to choose one of several algorithms to solve a problem. How can you pick the best one? To do it, you need to measure the algorithm efficiency somehow.

One of the options might be to measure the time your program needs to process its input. However, different computers may take different time to process the same data. Furthermore, the processing time may depend on the data itself. We obviously need something more universal. So, let's try to estimate the efficiency using **big O notation.**

§1. Input size

What does an algorithm usually do? It makes some calculations. Let's call **operations** the basic actions, such as addition, multiplication, comparison, variable assignment, etc. Of course, the calculation time depends on the machine, but it doesn't matter now because we want to compare algorithms, not machines. Now let's try to estimate the number of operations in an algorithm!

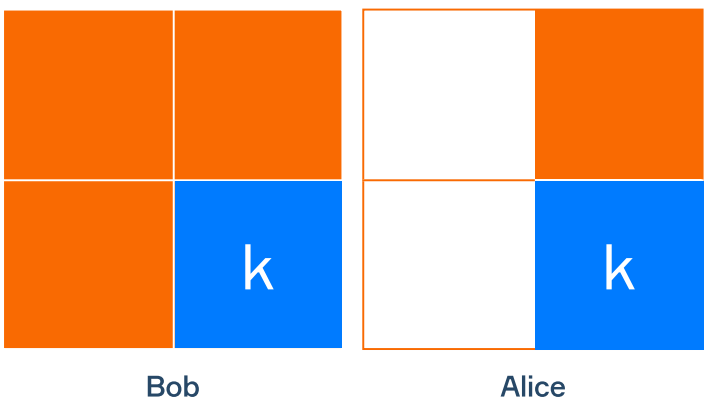


Buses aren't always punctual, are they? One day it may happen that they are there on time, while the other day they will take a lifetime to arrive. You can't blame solely the driver for that: the time of the trip depends directly on the number of passengers on the bus. The more passengers, the more stops, the longer the time to arrive. Likewise, the running time of an algorithm depends on the **input data.** Naturally, the program will take a different time to proceed with 10 or 1 000 000 numbers. We will use the term **input size** as a proxy measure of the size of input data. If you need to work with **m** numbers, then **m** is the input size. The input size isn't always the amount of the input data itself. If you need to find the first n prime numbers, then searching for 10 first primes or 10 000 first primes will also take a different time, however, you only enter a single number **n** as input. In such cases, that number's value is typically considered the input size.

If we can estimate how the number of operations depends on the input size, we will have a machine-independent measure of algorithm complexity. This is exactly what we need! Also, if we want to find a good algorithm, we are mostly interested in its behavior with big data. For this, we can compare the behavior of the algorithm's running time with some standard functions.

§2. Big O notation

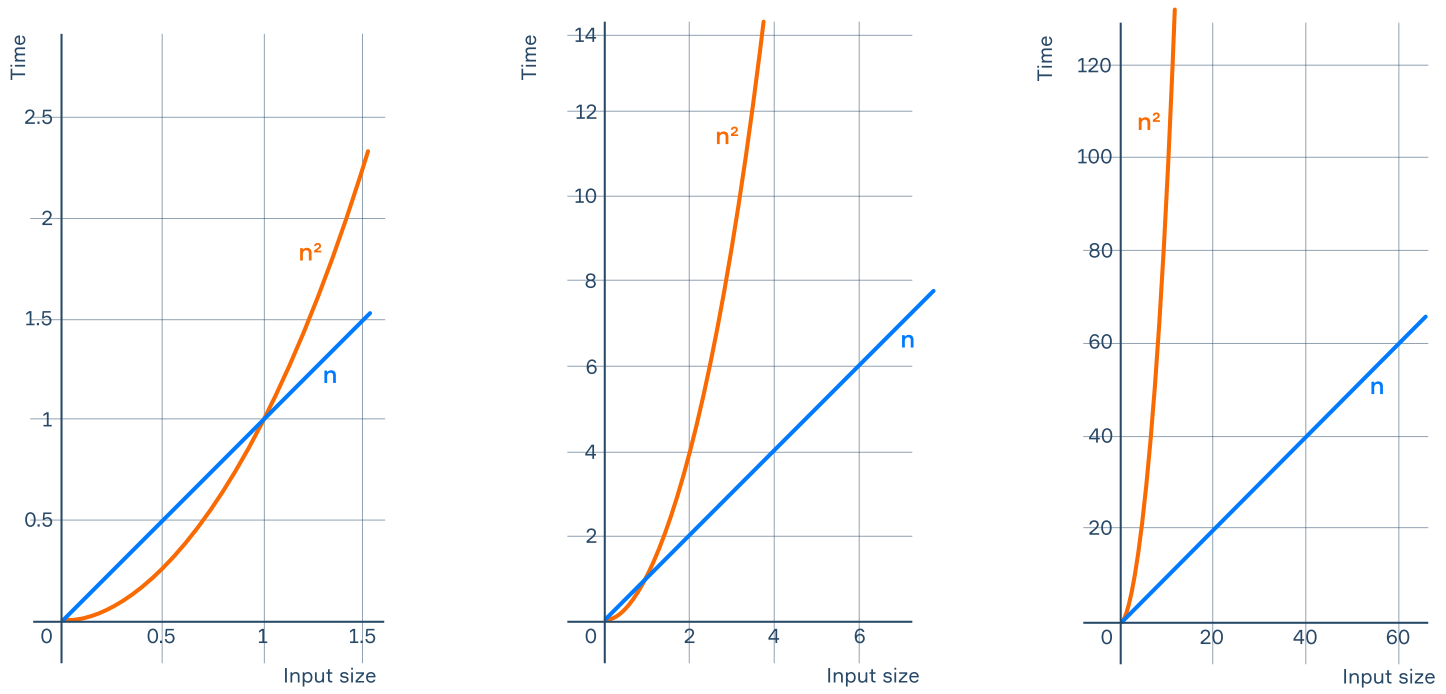
As we already mentioned, we will use the **big O notation** to measure the efficiency of algorithms. As a matter of fact, we have borrowed this symbol from mathematics; however, we shall not worry about the mathematical meaning or definition of the big O. Less formally, we can say that an algorithm has the **time complexity** $O(f(n))$ if its number of operations grows bigger similar to (or slower than) the function $f(n)$ when the input size n is a large number. In order to avoid unnecessary abstractness, let's consider the following task: given a $n \times n$ table with integers in its cells. Find the number k in the given table.



Alice and Bob have come up with their own algorithms to solve the problem. Bob's algorithm consists in scanning every cell of the table and checking if the corresponding value is equal to k. Well, this implies a maximum of n^2 comparisons, which means that the time complexity of Bob's algorithm is $O(n^2)$. On the other hand, Alice somehow knows earlier in which column the number k will be located, hence, she only needs to scan the elements of that column. A column consists of n cells, meaning that Alice's algorithm will take $O(n)$ time.

Basically, on a table 2×2, Bob will have to perform a maximum of 4 operations; meanwhile, Alice will perform no more than 2. Not a big difference really, is it? What if we have a table $n \times n$ for a large n? In this case, n^2 will be considerably bigger than n, as shown below. This is exactly what determines the efficiency of an algorithm – the way it behaves with large input sizes. Hence, we conclude that Alice's algorithm is faster than Bob's, as the big O notation suggests.

However, a simple question arises: why can't we write simply n^2 or n for the complexities? Why do we need to add this beautiful round letter in front of these functions? Well, imagine that the element k is placed in the first cell of the table. Bob will find it immediately and terminate his algorithm. How many steps does he perform: n^2? No, just one.



That is why we use the big O: roughly speaking, it describes the upper bound for the function's growth rate. This is one of the big O notation's essential advantages. It means that you can calculate how much time to allocate for processing a certain amount of input data and be sure that the algorithm will process it all in due time. In practice, an algorithm might sometimes work even better than what the big O notation shows for its complexity, but not worse.

§3. Common growth rates

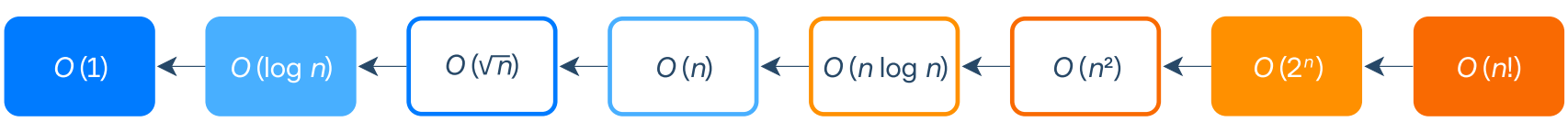
Below are, from best to worse, some common values of the big O function for the time complexity, also known as complexity classes.

- $O(1)$ (**constant time**). The algorithm performs a constant number of operations. Maybe one, two, twenty-six, or two hundred – it doesn't matter. What is important is that it doesn't depend on the input size. Typical algorithms of this class include calculating the answer using a direct formula, printing a couple of values, all letters of the English alphabet, etc.
- $O(\log n)$ (**logarithmic time**). Perhaps a quick reminder on logarithms is necessary. We usually refer to logarithms of base 2; however, the base does not affect the class. By definition, $\log_2 n$ equals the number of times n must be divided by 2 to get 1. That being said, it should not be difficult to guess that such algorithms divide the input size into **halves** at each step. They are relatively fast: if the size of the input is huge, say, 2^{31} (programmers should know the importance of this number), the algorithm will perform approximately $\log_2(2^{31}) = 31$ operations, which is pretty effective.
- $O(n)$ (**linear time**). The time is proportional to the input size, i.e., the time grows linearly as the input size increases. Often, such algorithms are iterated only once. They occur quite frequently, because it is usually necessary to go through every input element before calculating the final answer. This makes the $O(n)$ class one of the most effective classes in practice.
- $O(n^2)$ (**quadratic time**). Normally, such algorithms go through all pairs of input elements. Why? Well, mathematics is generous, it constantly provides us with important results: in this case, basic maths confirms that the number of unordered pairs in a set of n elements is equal to $\frac{n(n-1)}{2}$, which, as we will learn later in this topic, is $O(n^2)$. If you find it scary or difficult to understand, it is completely normal, it happens to the best of us. On the other hand, for those who are familiar with programming terms, the following sentence might come in handy: quadratic time algorithms usually contain two nested loops.
- $O(2^n)$ (**exponential time**). Just in case, let's mention that 2^n is the same as multiplying 2 by itself n times. Again, maths states that the number of subsets of a set of n elements is equal to 2^n, therefore, it is reasonable to expect that such algorithms scan all the subsets of the input elements. It is worth noting that this class is extremely inefficient in practice; even for small input sizes, the time taken by the algorithm will be remarkably high.

There are also other less common complexity classes, which you will come across in some following topics:

- $O(\sqrt{n})$ (**square root time**);
- $O(n \log n)$ (**log-linear time**);
- $O(n^k)$ (**polynomial time**);
- $O(n!)$ (**factorial time**).

Now let's gather all the classes together and sort them from the best to the worst, so that you remember which ones are the most effective, and which ones you should stay away from.



Complexity classes from the fastest (left) to the slowest...

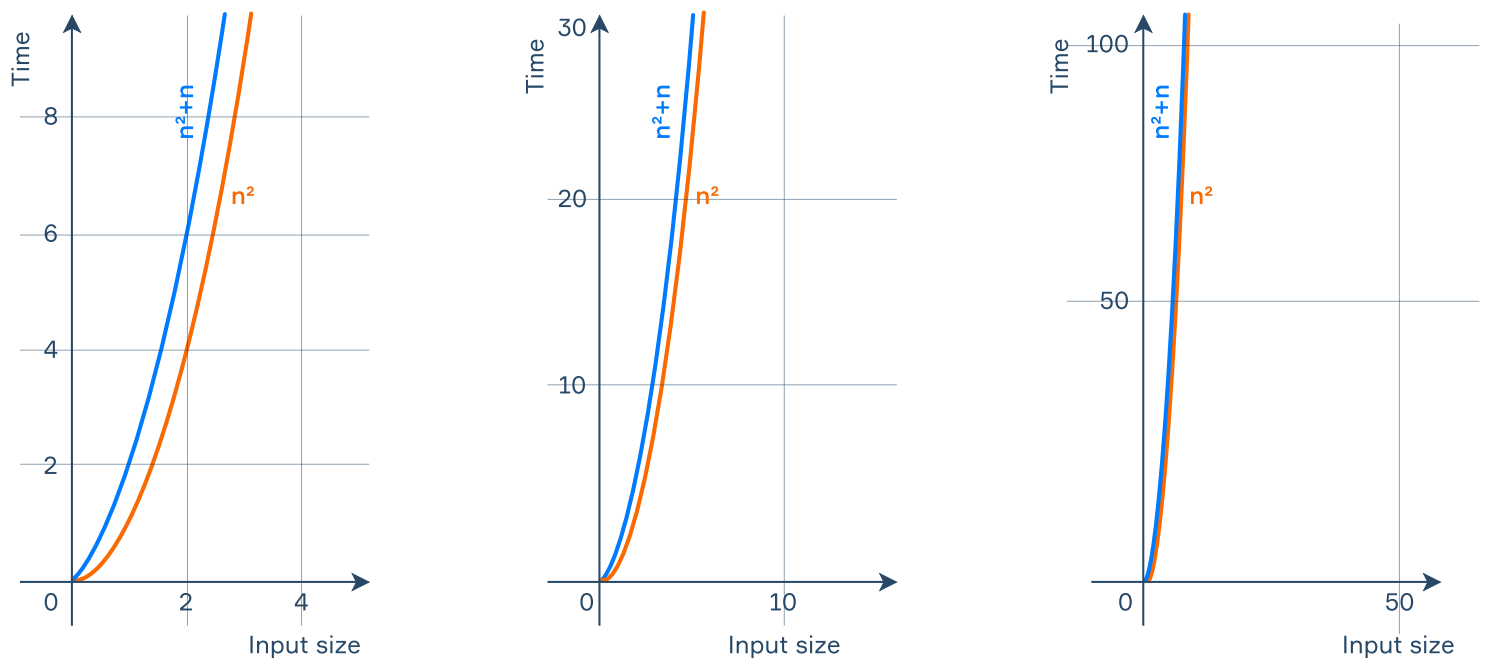
§4. Calculating complexity

Let's look at a simple example. You want to find the maximum of n numbers. You will probably decide to go through them and compare every new element with the maximum so far. You will make n comparisons, so the time complexity is $O(n)$.

However, algorithms are usually quite complex and consist of several steps, whose time complexity may belong to different time complexity classes from the list above. Therefore, to be able to calculate complexities by yourself, it is essential for you to get familiar with the basic properties of the Big O:

- **Ignore the constants.** As we discussed above, while calculating complexities, we focus solely on the behavior of our algorithm with large input sizes. Therefore, repeating some steps a constant number of times does not affect the complexity. For example, if you traverse n elements 5 times, we say that the algorithm's time complexity is $O(n)$, and not $O(5n)$. Indeed, there is no significant difference between 1 000 000 000 and 5 000 000 000 operations performed by the algorithm. In either case, we conclude that it is relatively slow. Formally, we write $c \cdot O(n) = O(n)$. It is similar for the rest of the complexity classes.
- **Applying a procedure n times.** What if you need to go over n elements n times? It is not a constant anymore, as it depends on the input size. In this case, the time complexity becomes $O(n^2)$. It's simple: you do n times an action proportional to n, which means the result is proportional to n^2. In big O notation, we write it as $n \cdot O(n) = O(n^2)$.

- **Smaller terms do not matter.** Another common case is when after doing some actions, you need to do something else. For instance, you traverse n elements n times and then traverse n elements again. In this case, the complexity is still $O(n^2)$. Additional n actions do not affect your complexity, which is proportional to n^2. In big O notation, it looks like this: $O(n) + O(n^2) = O(n^2)$. All in all, always keep the largest term in Big O and forget about all others. It is rather easy to understand which terms are larger based on the order provided in the previous section. Naturally, a question arises: why is it correct to ignore the smaller terms? Let's illustrate the example above:



The images show that when the input size n is large, the graphs of n^2 and $n^2 + n$ almost coincide (their growth rate is similar). As for n^2, for large n this value is considerably greater than n, therefore adding n to it does not affect the value of the function much. This is why we can rightfully write $O(n^2)$ instead of $O(n^2 + n)$.

§5. Conclusion

Big O notation is an essential instrument for algorithm performance evaluation. We can use it to assess both the time and the memory complexity. The greatest advantage of big O notation is that it classifies an algorithm rather than gives you a real running time in seconds or required memory in megabytes.

We should note that it is completely normal if you still find the concept of Big O a bit confusing. It is similar to reading the rules of a board game for the first time without actually having the board in front of you. As soon as you start playing, you will better realize the meaning of those rules. Analogously, in the following topics on algorithms, we will describe in detail how to calculate algorithm complexity. That will definitely lead to a better understanding of this topic as well. In a nutshell, we hope this topic hasn't demotivated you, on the contrary, you should be motivated and hungrier for more.