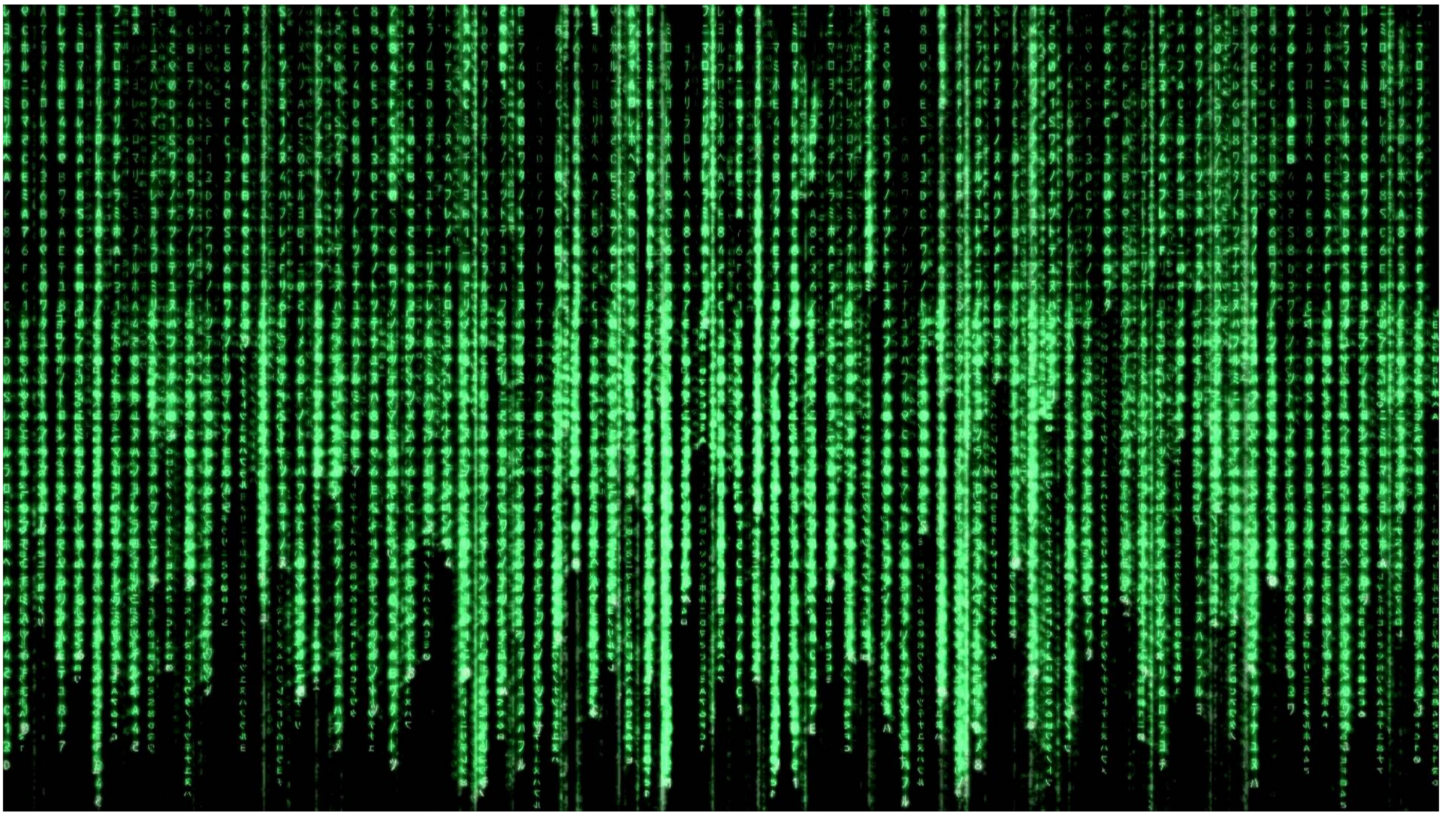
Big O Notation for dummies

Artem Zekov
Nov 22, 2018 · 4 min read

If you are interested in algorithms you must have heard of ***"Big O Notation"***. This thing is easier than it tends to be. Are you curious of it? In this article I will cover main concepts of ***Big O Notation*** to let you use this powerful tool.



Terminology

In order to study algorithm's performance, computer scientists ask how its performance changes as the size of the problem changes. It is simple version of "why **Big O Notation** was created".

Big O Notation uses mathematical functions to describe algorithm's runtime performance(sometimes it is called the program's asymptotic performance). The function is written within parentheses after a capital letter O. For example, $\mathbf{O(N^2)}$

means that algorithm's runtime(or memory or whatever you are measuring) increases as the square of **N** (**N** usually represents array length).

Basics

There are five basic rules for calculating algorithm's ***Big O Notation***:

1. If an algorithm performs a certain sequence of steps **f(N)** times for a mathematical function **f**, it takes **O(f(N))** steps.

2. If an algorithm performs an operation that takes **f(N)** steps and then performs another operation that takes **g(N)** steps for function f and g, the algorithm's total performance is is **O(g(N) + f(N))**.

3. If an algorithm takes **O(g(N) + f(N))** steps and the function **f(N)** is bigger than **g(N)**, algorithm's performance can be simplified to **O(f(N))**.

4. If an algorithm performs an operation that takes **f(N)** steps, and for every step performs another operation that takes **g(N)** steps, algorithm's total performance is **O(f(N)×g(N))**.

The following examples should make it easier for you to understand the essentials of **Big O Notation**.

Rule 1

> If an algorithm performs a certain sequence of steps f(N)times for a mathematical function f, it takes O(f(N)) steps.

Look through the following algorithm:

```
function findBiggestNumber(array) {
    let biggest = array[0];

    for (let i = 0; i < array.length; i++) {
        if (array[i] > biggest) {
            biggest = array[i];
```

```
7               }
8           }
9
10          return biggest;
11      }
```

This algorithm takes an **array** as an argument and returns the biggest number in that **array**. We define the ***biggest*** variable equal to the first value in the **array**. Then we loop through the **array** and find the biggest value in it. After the loop is finished we return the ***biggest*** variable.

This algorithm examines each of the **N** items once (where **N** is an array length), so it's performance **O(N)**.

Rule 2

> If an algorithm performs an operation that takes f(N) steps and then performs another operation that takes g(N) steps for function f and g, the algorithm's total performance is is O(g(N) + f(N)).

If you look again at the ***findBiggestNumber*** shown at the previous section, you'll see that there are few operations outside the loop(line 2 and 10). Each outer operation takes constant amount of time, so both of them has performance ***O(1)*** :

```
1   function findBiggestNumber(array) {
2       let biggest = array[0];          // O(1)
3
4       for (let i = 0; i < array.length; i++) {     //O(N)
5           if (array[i] > biggest) {
6               biggest = array[i];
7           }
8       }
9
10      return biggest;                  // O(1)
```

```
}
```

Then the total runtime of the algorithm is **O(1 + N + 1)**. You can use algebra rules inside of **Big O Notation**, so final algorithm's performance is **O(N + 2)**.

Rule 3

> If an algorithm takes O(f(N) + g(N)) steps and the function f(N) is bigger than g(N), algorithm's performance can be simplified to O(f(N)).

The previous ***findBiggestNumber*** algorithm has **O(N + 2)** runtime. When **N**grows large, the function **N** is larger than our constant value **2**, so algorithm's runtime can be simplified to **O(N)**.

Ignoring smaller functions helps you to concentrate on the algorithm's behavior as N size becomes large.

Rule 4

> If an algorithm performs an operation that takes f(N)steps, and for every step performs another operation that takes g(N) steps, algorithm's total performance is O(f(N)×g(N)).

Consider the following algorithm:

```
function containsDuplicates(array) {
    for (let i = 0; i < array.length; i++) {
        for (let r = 0; r < array.length; r++) {
            if (i === r) {
                continue;
            }
```

```
                if (array[i] === array[r]) {
                    return true;
                }
            }
        }

        return false;
    }
```

These algorithm takes an **array** as the only one argument and returns if the **array** contains duplicate values. Algorithm has two nested loops. The outer loop iterates through all items in **array**, so it takes **O(N)** steps. For each iteration of the outer loop the inner loop also iterates over all items in **array**, so it takes **O(N)** steps too. Because of the fact that one loop is nested inside the other we can combine their performances — **O(N × N)** or **O(N^2)**.

Conclusion

> **Notice that Big O Notation is not the only one way to analyse algorithm's behavior**

As you can see, **Big O Notation** isn't as complicated as it tends to be. To get deeper understanding of **Big O Notation** and algorithms in general check these books out: *Essential Algorithms* and *Grokking Algorithms*. They've built my foundation in understanding algorithms and their behavior. Moreover, a major part of information in this article was taken from those books, so I highly recommend you to read them. Hope this article has upgraded your knowledge in programming.

Thanks for reading! I'm planning to make series of articles about algorithms analysis, so you'd better to follow me on medium and clap for story to help me in growing my community! Read my other articles about programming:

ESlint introduction

This article was made for beginners in JS world as me, I have problems installing ESlint to my project. When I have…

medium.com

dependencies vs devDependencies

Have you ever asked yourself what is the difference between these two types of dependencies inside your package.json…

medium.com

JavaScript Programming Machine Learning Algorithms Development