1. **The Big O Notation**

Algorithmic Complexity Made Simple — This Is NOT An Oxymoron!

[[Semi Koen](https://archive.fo/o/K5xAQ/https:/semika.medium.com/)](https://archive.fo/o/K5xAQ/https:/semika.medium.com/" \t "_blank)

[Semi Koen](https://archive.fo/o/K5xAQ/https:/semika.medium.com/" \t "_blank)

[Jun 19, 2020·8 min read](https://archive.fo/o/K5xAQ/https:/towardsdatascience.com/the-big-o-notation-d35d52f38134?source=post_page-----d35d52f38134-----------------------------------" \t "_blank)



Photo by [Dollar Gill](https://archive.fo/o/K5xAQ/https:/unsplash.com/@dollargill?utm_source=medium&utm_medium=referral) on [Unsplash](https://archive.fo/o/K5xAQ/https:/unsplash.com/?utm_source=medium&utm_medium=referral" \t "_blank)

An algorithm’s performance depends on the number of steps it takes. Computer Scientists have borrowed the term ‘Big-O Notation’ from the world of mathematics to accurately describe an algorithm’s efficiency. Many **self-taught** Developers and Data Scientists either settle for a solution that ‘just works’ without thinking how to improve their code’s performance, or go into optimising it without really understanding the basics. Their attempts are either fruitless or have very small, and mostly incidental, impact.

Measuring an algorithm’s complexity is not a difficult concept to grasp — Although this sounds like an [oxymoron](https://archive.fo/o/K5xAQ/https:/en.wiktionary.org/wiki/oxymoron) — it is not!  
In this article, we are going to leave out the [mathematical jargon](https://archive.fo/o/K5xAQ/https:/en.wikipedia.org/wiki/Big_O_notation) and explain the Big-O concept in an **easy to understand** way. We will refine our understanding with standalone Python snippets and we will conclude our journey with an all-in-one cheat-sheet for future reference.

Introduction

Time Complexity

Instead of focusing on units of time, Big-O puts the **number of steps** in the spotlight. The hardware factor is taken out of the equation. Therefore we are not talking about *run time,* but about *time complexity*.  
⚠ We will not cover the *Space Complexity* i.e. the how much *memory* an algorithm takes up. We will talk about it another time :)

Big-O Definition

An algorithm’s Big-O notation is determined by how it responds to different sizes of a given dataset. For instance how it performs when we pass to it 1 element vs 10,000 elements.

O stands for ***Order Of***, so O(N) is read “Order of N” — it is an **approximation** of the duration of the algorithm given N input elements. It answers the question: “*How does the number of steps change as the input data elements increase?*”

O(N) describes how many steps an algorithm takes based on the number of elements that it is acted upon.

⭐️ It is that simple!!

Best vs Worst Scenario

Starting with a gentle example: Given an input array[N], and a value X, our algorithm will search for the value X by traversing the array from the start until the value is found.

Given this 5-element array: [**2**,1,6,3,**8**] if we were searching for X=8 the algorithm would need 5 steps to find it, but if we were searching for X=2 it would only take 1 step. So best case scenario is when we look for a value that is in the *first cell* and worst case scenario is when the value is at the *last cell*, or not there at all.

The Big-O notation takes a **pessimistic** approach to performance and refers to the worst case scenario. This is really important when we describe the complexities below, and also when you try to compute the complexity of your own algorithms: *Always think of the worst case scenarios*.

Now that we have our rules and vocabulary defined, without further ado, let’s dive into the most common complexities you may come across…

O(1) — Constant

O(1) means that the algorithm takes the same number of steps to execute regardless of how much data is passed in.

Example

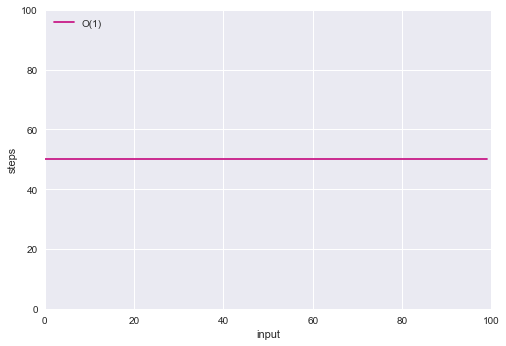
✏️ Determine if the **i-th element** of an array is an odd number.

Whether we access the 1st or 2nd or millionth item it doesn’t matter… We can access it directly by using the index operator array[i] .



Plot

If we were to represent the number of steps (y-axis) vs the number of input elements (x-axis), O(1) is a perfect horizontal line, since the number of steps in the algorithm remains constant no matter how much data there is. This is why it is called *constant time*.



O(1)

O(N) — Linear

An algorithm that is O(N) will take as many steps as there are elements of data. So when an array increases in size by one element, an O(N) algorithm will increase by one step.

Example

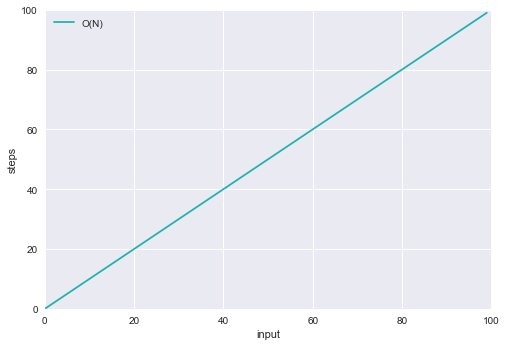
✏️ **Traverse** an array and print each element.

Here, we need to access all the elements one by one, so the calculation time increases at the same pace as the input.



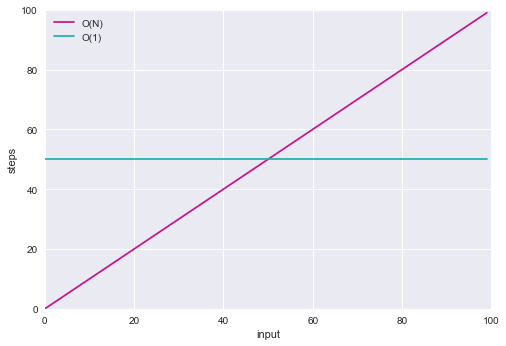
Plot

O(N) is a perfect diagonal line, as for every additional piece of data, the algorithm takes one additional step. This is why it is also referred to as *linear time*.



O(N)

Let’s plot the O(1) and O(N) algorithms in the same graph and let’s assume that the O(1) algorithm constantly takes 50 steps.



O(1) vs O(N)

What can we observe?

→ When the input array has less than 50 elements, the O(N) is more efficient.  
→ At exactly 50 elements the two algorithms take the same number of steps.  
→ As the data increases the O(N) takes more steps.

Since the Big-O notation looks at how the algorithm performs as the data grows to **infinity**, this is why O(N) is considered to be less efficient than O(1).

O(N²) — Quadratic

O(N²) represents the complexity of an algorithm, whose performance is proportional to the square of the size of the input elements. It is generally quite slow: If the input array has 1 element it will do 1 operation, if it has 10 elements it will do 100 operations, and so on.

Example

✏️ Find **duplicates** in an array.

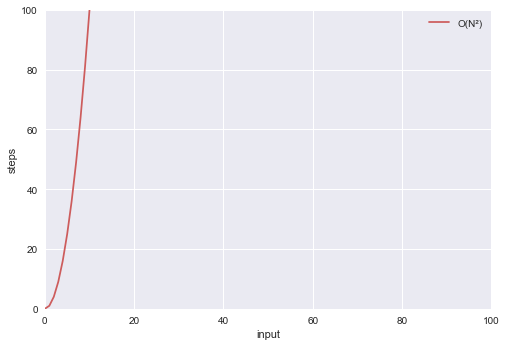
This is a ‘naive’ implementation, but it traverses the array twice:



Adding more nested iterations through the input will increase the algorithm’s complexity: e.g. if the number of iterations is 3 then its complexity will be O(N³) and so forth. Usually, we want to stay away from *polynomial* running times (quadratic, cubic, Nˣ, etc).

Plot

The O(N²) line is a sharp curve:



O(N²)

O(logN) — Logarithmic

Simply put, O(logN) describes an algorithm that its number of operations increases by one each time the data is doubled.

**📌 Logarithms Refresher:**  
You may not remember what logarithms are, but you probably know what exponents are:  
2³ = 2 \* 2 \* 2 = 8 — Here we multiply the number 2, 3 times.**Logarithms are the flips of exponents.**  
log₂8 answers the question: how many 2s do we multiply together to get 8? The answer is 3.In other words, if we keep dividing 8 by 2 until we end up to 1, how many 2s do we have in our equation?8 / **2** / **2** / **2** = 1. The answer is 3 again.

Example

Logarithmic time complexities usually apply to algorithms that divide problems in half every time.

✏️ **Dictionary** lookup (aka binary search).

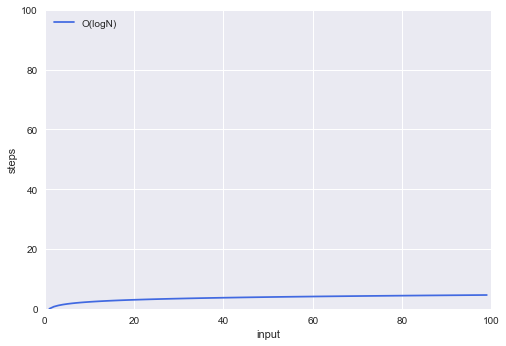
1) Open the dictionary in the middle and check the first word.  
2) If our word is alphabetically more significant, look in the right half, else look in the left half.  
3) Divide the remainder in half again, and repeat steps 2 and 3 until we find our word.



We can only pick one possibility per iteration, and our pool of possible matches gets divided by two in each iteration. This makes the time complexity of binary search O(logN).

Plot

The number of steps barely increase, as the input grows (i.e. it takes just one additional step each time the data doubles):



O(logN)

And So On…

By now we have learnt the four most important Big-O taxonomies. Of course there are a few more, but I am confident you will be able to understand them. Let’s quickly cover them:

O(N logN) — Log-linear

An algorithm of this complexity class is doing log(N) work N times and therefore its performance is slightly worse than O(N). Many practical algorithms belong in this category (from [sorting](https://archive.fo/o/K5xAQ/https:/en.wikipedia.org/wiki/Merge_sort), to [pathfinding](https://archive.fo/o/K5xAQ/https:/en.wikipedia.org/wiki/Dijkstra's_algorithm), to [compression](https://archive.fo/o/K5xAQ/https:/en.wikipedia.org/wiki/Huffman_coding)), so we are mentioning it for completeness.

**✏️ Example**: Merge Sort — it is a ‘Divide and Conquer’ algorithm: it divides the input array in two halves, calls itself for each one and then merges the two sorted halves.  
💡 **Scalability**: Average.

O(2ᴺ) — Exponential

Exponential growth means that the algorithm takes twice as long for every new element added.

**✏️ Example**: Find all subsets in a dataset.  
💡 **Scalability**: Poor.

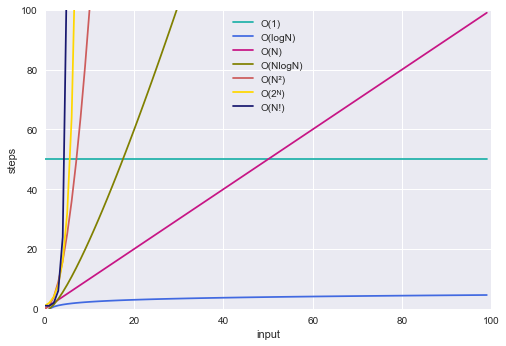
O(N!) — Factorial

This class of algorithms has a run time proportional to the factorial of the input size: n! = n \* (n-1) \* (n-2) \* (n-3) \* . . . \* 1.

**✏️ Example**: Find all different permutations in a dataset.  
💡 **Scalability**: Very Poor.

Growth Hierarchy

The Big-O notation offers a consistent mechanism to compare any two algorithms and hence help us make our code faster and more scalable. Putting all of the complexities in a single graph, we can observe in a visual manner how they compare in terms of performance:



Big-O Complexity Classes

Conclusion

I hope you now realise that the Big-O notation is not an intimidating concept after all!

Recap

* Algorithm speed is not measured in seconds but in terms of growth
* The Big-O Notation tells us how an algorithm scales against changes in the input dataset size
* O stands for *Order Of*— as such the Big-O Notation is approximate
* Algorithm running times grow at different rates:

O(1) < O(logN) < O(N) < O(N logN) < O(N²) < O(2ᴺ) < O(N!)