Published in Better Programming

Klement Omeri Follow
Dec 29, 2021 · 12 min read · Listen

Save

Visualizing Big O Notation in 12 Minutes

Let’s take a look at the BigO Notation from a different perspective

Photo by Alexandr Holovko on Unsplash

I truly believe that there is nothing you can’t learn if you start from the most fundamentals.

If you combine the correct resources and the will to learn nothing can stand in front of you and your

Get unlimited access Open in app

you all better understand this topic.

Let’s begin with the definition:

> *Big O notation is a* ***mathematical notation*** *that describes the limiting behavior of a function when the argument tends towards a particular value or infinity.*

How do we understand that?

In computer science, there are almost always several ways to solve a problem. Our job as software engineers is to find the most effective solution and implement it. But what does it mean effective anyway? Is it the fastest way? Is it the one that takes less space than the others? Actually, it depends. This is purely related to your particular problem. If you are working in embedded systems with limited memory, for example, if the problem is to calculate the required power in watts to defrost 200 gr of meat in a microwave you can trade an algorithm that is more memory efficient and takes 1s to make this calculation to another one which makes this calculation in milliseconds but will take much more memory. After all, even if it takes milliseconds to start, the defrost process itself will require 10 to 15 mins.

If we talk about the algorithm which locks missiles to a target airplane it is clear that we are dealing with milliseconds here and the memory consumption can be sacrificed. The plane is big enough to have free space for some more memory slots.

> *In general, software engineering is about the trade-offs. A good software engineer has to be aware of the requirements and come up with solutions to fulfill them.*

With all that being said, it is understandable right now that we need to somehow quantify and measure the performance and memory implications of any algorithm. One way to do that is to take a look at how many seconds one algorithm requires to complete. That can provide some value but the problem is that if my search algorithm takes 2 to 3 seconds on my laptop with an array of 1000 items it can take less than that on another more powerful laptop right? Even if we agree to take my laptop's performance as a base, we are not aware of telling what happens when the size of the array doubles? What happens when the size of the array goes to infinity?

To answer these questions we will need a measurement that is independent of the machine and can tell us what will happen with our algorithm when the size of the input gets larger and larger. Here comes the BigO Notation.

BigO aims to find how many operations you need to perform in the worst-case scenario given an input of any size. It aims to find what is the limit for the number of operations of your algorithm

If you copy an array of 10 items in another array then you will need to loop all over the array, which means 10 operations. Which in BigO notation is expressed as O(N) where N is the size of the input array. The Space Complexity for this example is again O(N) because you are going to allocate some more memory for the copied array.

What BigO does is to give you a math function that is purely focused on finding the limit of the number of operations you need to perform when the size of the input gets larger. For example, if you are searching for number 5 inside of a given list with a linear search. Then in the **worst case**, this number will be at the end of the list, but since you will start the iteration from the beginning you will need to perform as many lookup operations as the number of the input.

```
[1, 2, 3, 4, 5]  # you will perform 5 operations here to find it
```

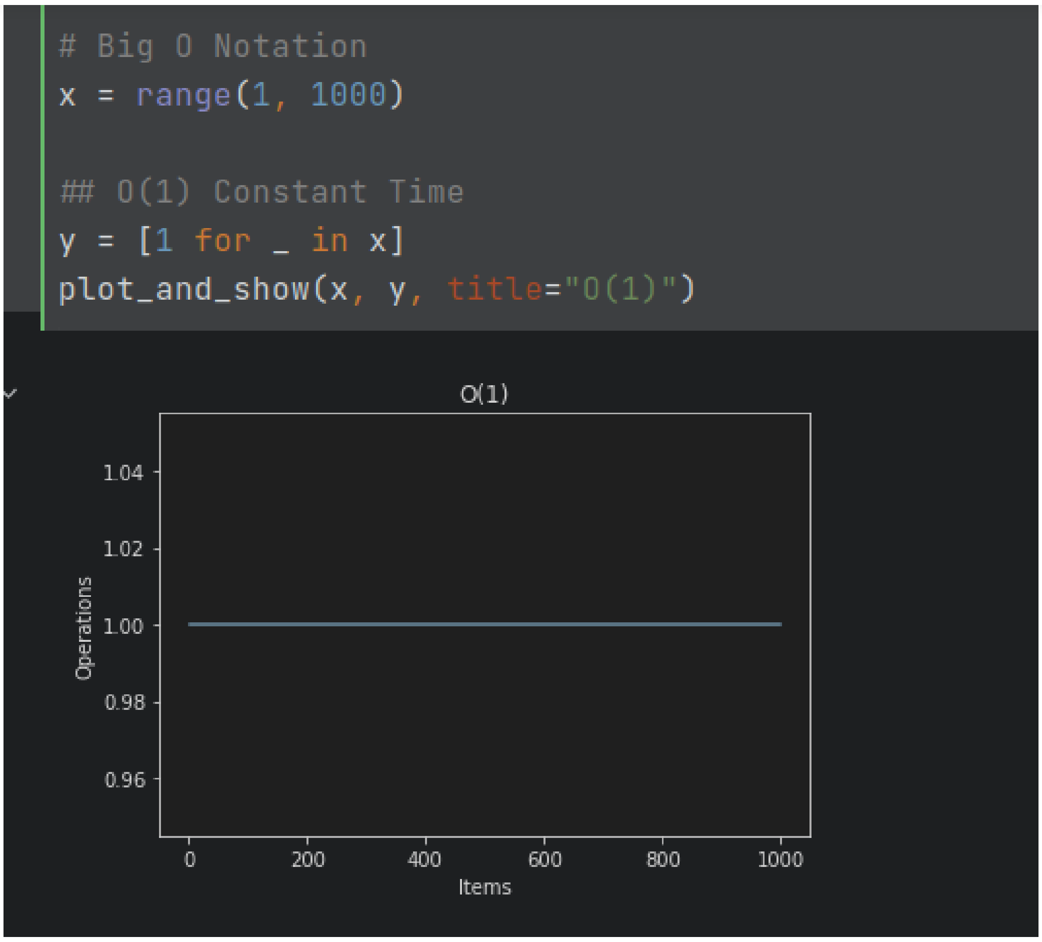
Here I want to stop for a moment on the term ***worst case***. If you think about it there is a chance that the required number to be at the beginning of the list, in this case, you will perform only 1 operation.

```
[5, 1, 2, 3, 4]  # you will perform only one operation in this case
```

The problem is that we can not take into consideration the best case and hope that it will happen most of the time because in this way we are not able to compare different algorithms to each other. In the context of the BigO notation, we are always interested in the worst case (with some exceptions like hash maps, more on that later).

I said before, that BigO gives you a math function that is focused on finding the limit of the number of operations. When we talk about the limits in math we can not only talk about them without any visualization. This helps a lot to understand the trend of the function as the size of the input goes to infinity. Let's start by analyzing one by one some very common BigO notations together with an example.

O(1) Constant Time



```
# Big O Notation
x = range(1, 1000)

## O(1) Constant Time
y = [1 for _ in x]
plot_and_show(x, y, title="O(1)")
```

O(1)

This is understandably the best BigO notation an algorithm can have. When you want to perform a certain action and you can perform it in only one operation. Let’s take a look at an example using python:

```
country_phone_code_map = {
    'Albania': '+355',
    'Algeria': '+213',
    'American Samoa': '+684',
}
country = 'Albania'
print(country_phone_code_map[country])  # 1 operation
>>> '+355'
```

In python, if you want to make a lookup for an item into a dict then the operation is O(1) Time Complexity. Dict in python is similar to HashMap in other languages.

To be exact, the worst-case scenario is O(N) and this is related to how well the data structure is implemented. The hashing function takes the key role here but in general, it is agreed that the BigO for dict lookups is O(1). If you are in a coding interview you can assume that it's O(1).

I want to stop on another very important topic when calculating the BigO. **The constants**. You may or may not be familiar that the constants are ignored when calculating the BigO. I don't want you to just accept this as a rule and don't think about the reasons behind it.

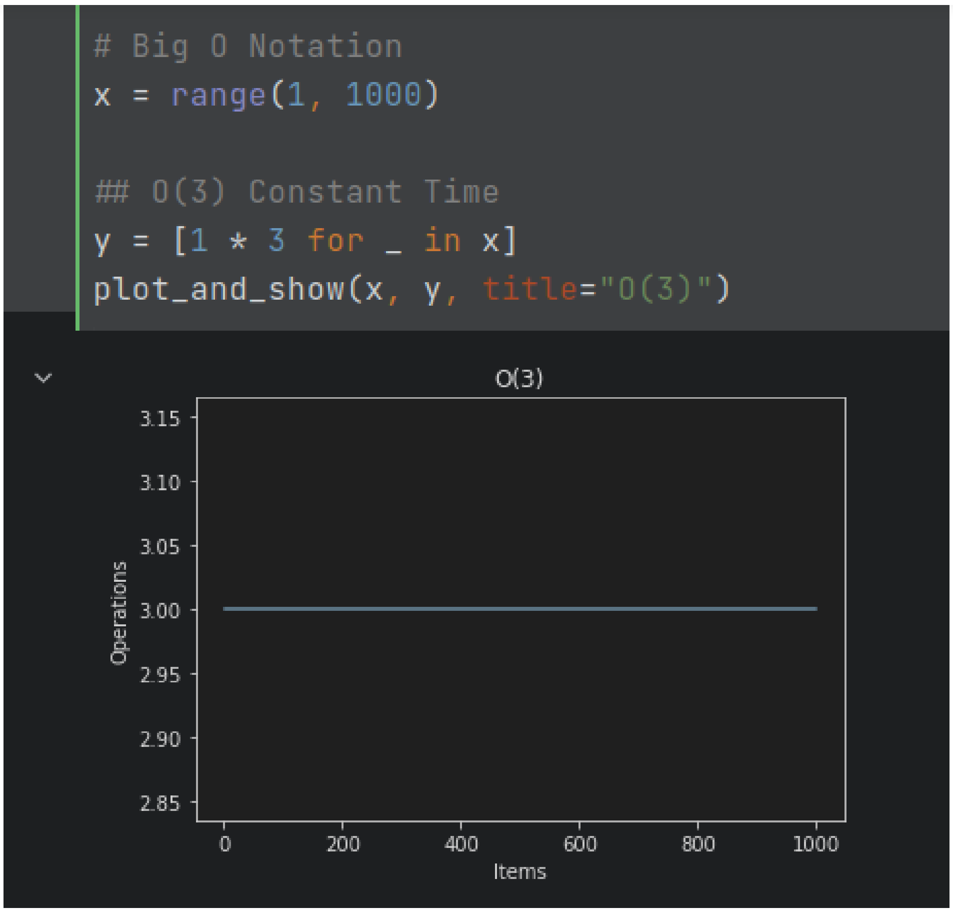
This is exactly why I am visualizing the BigO notation. So let's assume that for the above example we will also need to get the 3 and 2 letter country code by having the country name. This means that we have some other mapping for the 2 and 3 letter country code and we will just need to perform 2 more operations inside the same function.

```
country_phone_code_map = {
    'Albania': '+355',
    'Algeria': '+213',
}
country_2_letter_code_map = {
    'Albania': 'AL',
    'Algeria': 'DZ',
}
country_3_letter_code_map = {
    'Albania': 'ALB',
    'Algeria': 'DZA',
}
country = 'Albania'

phone_code = country_phone_code_map[country]  # 1 operation
two_letter_code = country_2_letter_code_map[country]  # 1 operation
three_letter_code = country_3_letter_code_map[country]  # 1 operation
```

If we continue to count the number of operations as we agreed to do before. Here we will have 3 operations that make the BigO = O(3) right?

Let's visualize this:



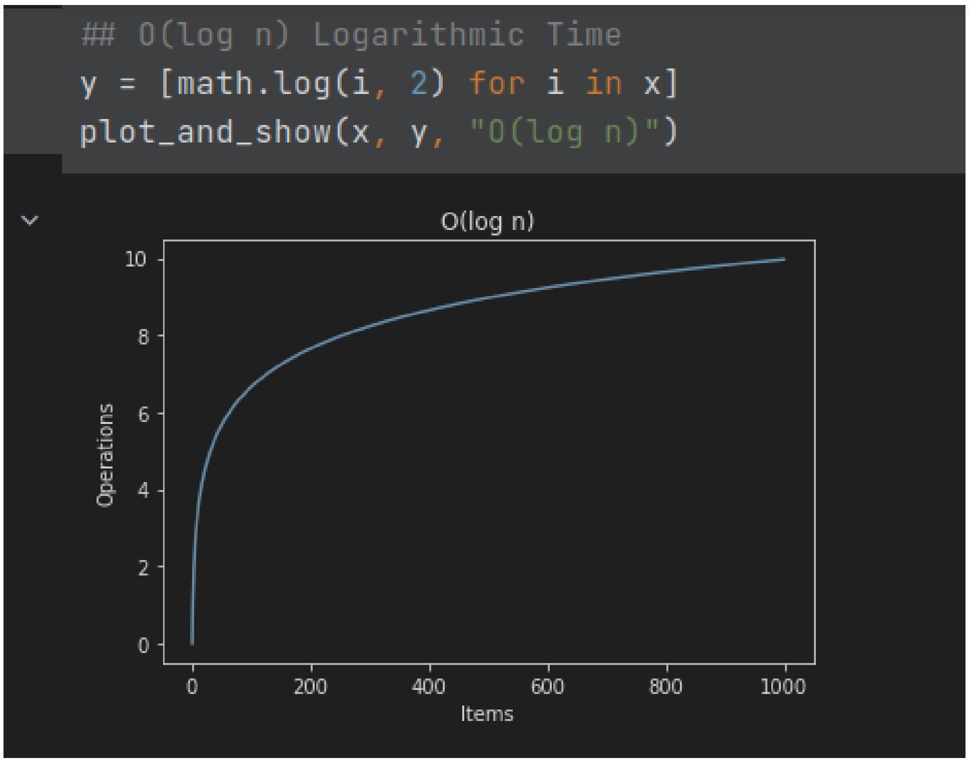
```
# Big O Notation
x = range(1, 1000)

## O(3) Constant Time
y = [1 * 3 for _ in x]
plot_and_show(x, y, title="O(3)")
```

As you can see the number of operations moved up by 3. Which means we are actually performing more than one operation to complete this task. But, BigO says that if there are constants just ignore them. So O(3) or O(2n) or O(2n + 1) will be respectively O(1), O(n), O(n).

This is because we are interested to know the limit of the function as N goes to infinity and not how many operations exactly it will perform. We are not calculating the number of operations but instead, we are interested to see how the number of operations will grow as N goes to infinity. You may be thinking that, yes but an algorithm with `O(1000n)` is slower than one with O(n) so we need to consider that 1000 we cannot ignore it. That's true but this number 1000 is a constant and it is not getting bigger as N. Even when N is 10 it will remain 1000, even when N is 1B it will remain 1000. So it does not provide us with any valuable information regarding the limits of the function. The only important part is O(n) which tells us that the more you increase, the more you are going to perform operations to complete the task.

O(logn) Logarithmic Time



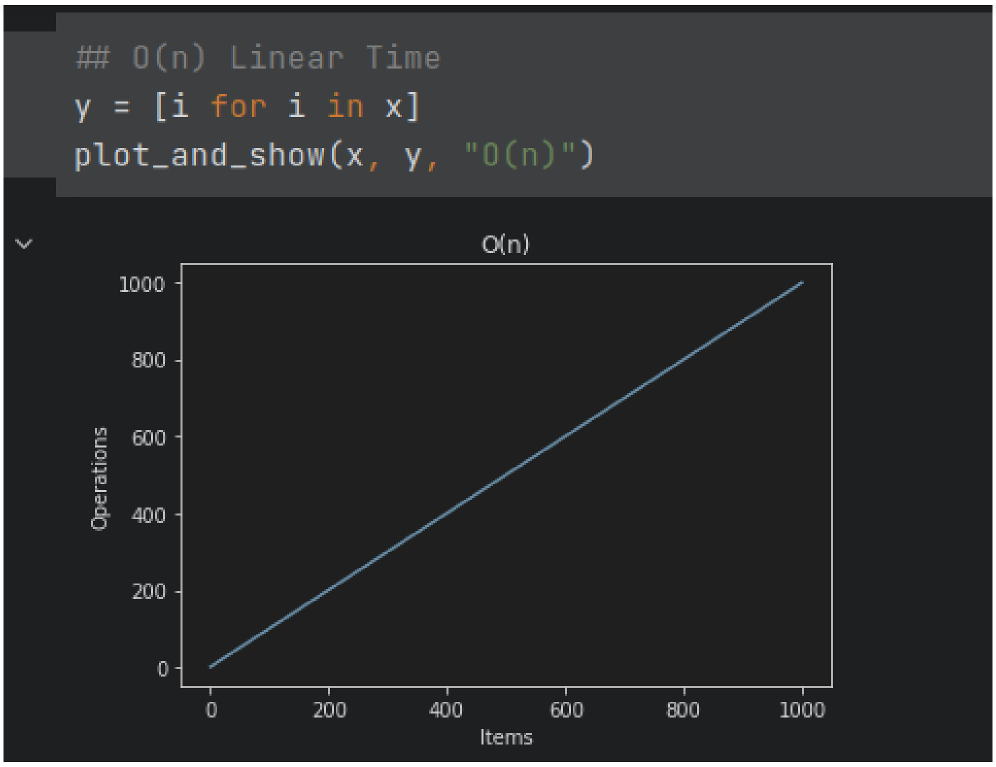
```
## O(log n) Logarithmic Time
y = [math.log(i, 2) for i in x]
plot_and_show(x, y, "O(log n)")
```

This notation usually comes together with searching algorithms with the divide and conquer approach. If we are searching for a number in a sorted array we can use the most basic algorithm, binary search. This algorithm will divide the array by half on every operation and it will take log(n) operations to find the number. Here is a nice tool to visualize this algorithm.

> *Something important here is that when we talk about logarithm in computer science without specifying the base we* ***always*** *talk about the* ***logarithm with base 2.*** *In math, we are used to a logarithm with base 10 in this case but it is different in computer science. Just keep that in mind when dealing with complexity analysis.*

As you can see from the image above, this is actually a very good time complexity. To have a complexity of O(logn) in plain English means that every time the size of the input doubles we only need to make one more iteration to complete the task. When N is about a million we need to perform only 20 operations, and when it gets around 1billion we need to perform only 30 operations. You can see the power of an algorithm with O(logn) time complexity. For such a huge increase in N, we only need 10 more operations to perform.

O(N) Linear Time



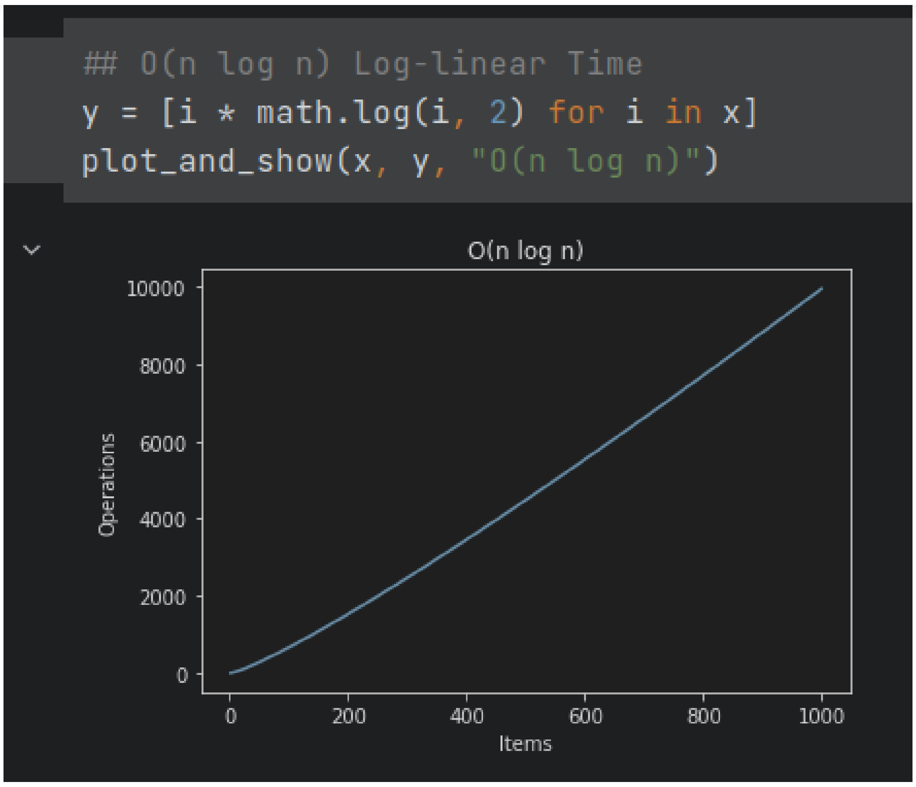
```
## O(n) Linear Time
y = [i for i in x]
plot_and_show(x, y, "O(n)")
```

In this case when N goes to infinity the number of operations goes to infinity as well with the same rate as N. An example is a linear search as we discussed before.

```
array = [1, 2, 3, 4, 5]
number = 5
for index, item in enumerate(array):  # loop n times
    if item == number:  # check for equality
        print(f'Found item at {index=}')
        break

>>> Found item at index=4
```

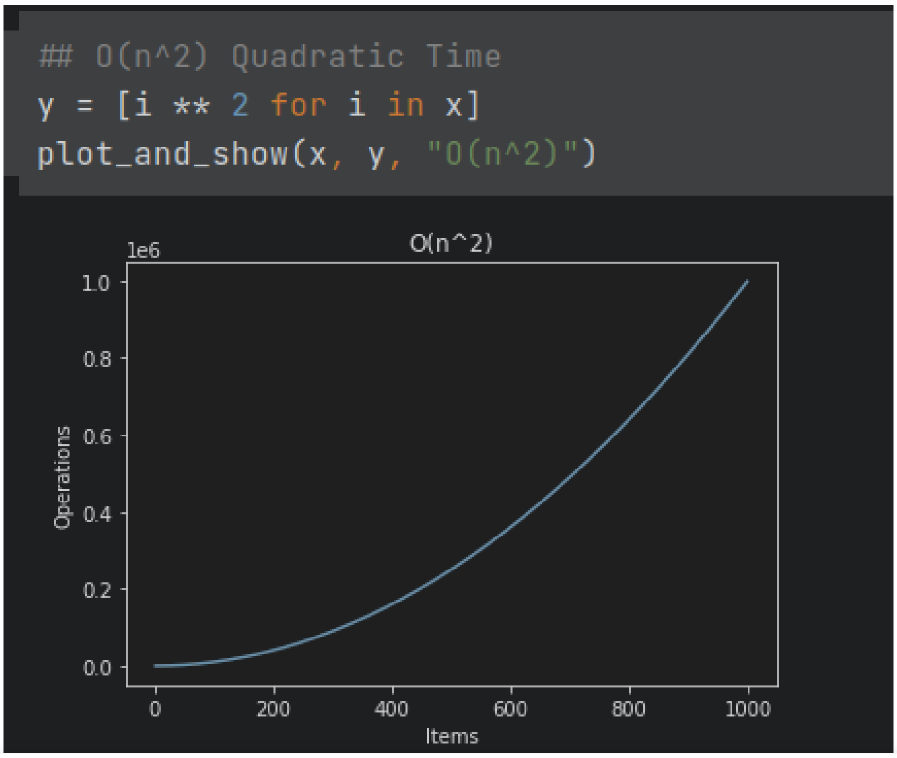
O(NlogN) Log-Linear Time



```
## O(n log n) Log-linear Time
y = [i * math.log(i, 2) for i in x]
plot_and_show(x, y, "O(n log n)")
```

This notation usually comes together with sorting algorithms. Take a look at this visualization for merge sort. Merge sort divides the array into two halves O(logn) and takes O(n) linear time to merge divided arrays.

O(N²) Quadratic Time



```
## O(n^2) Quadratic Time
y = [i ** 2 for i in x]
plot_and_show(x, y, "O(n^2)")
```

Get unlimited access Open in app

Usually algorithms with nested loops. For example a sorting algorithm with brute force, that is looping all over the array in two **nested** for loops. Bubble sort is an example:

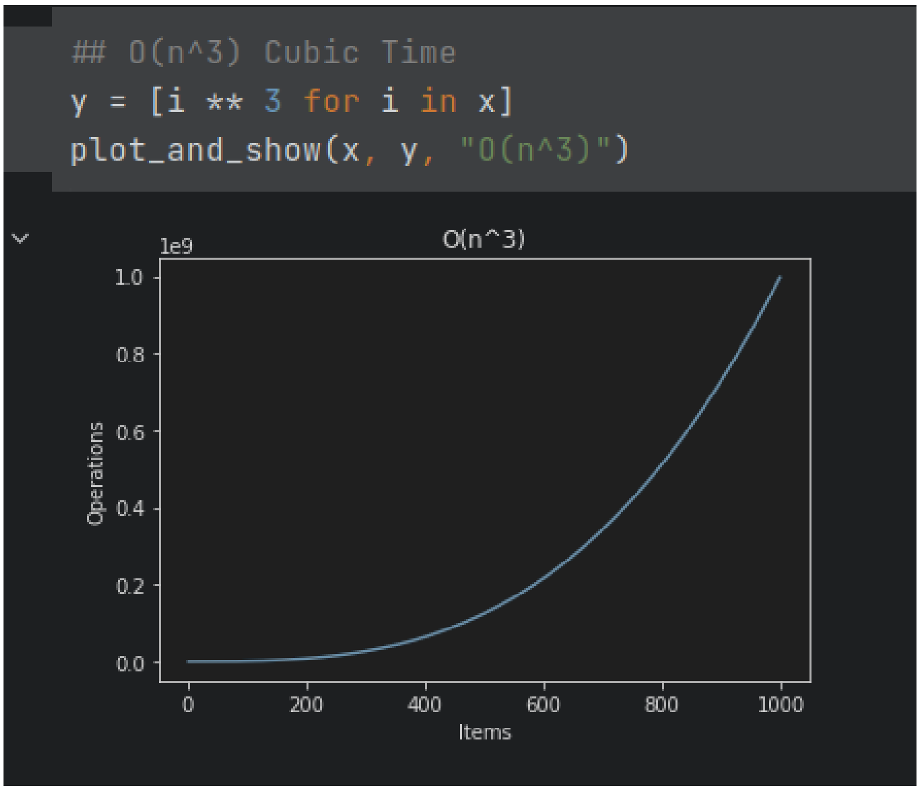
```
def bubble_sort(data):
    for _ in range(len(data)):  # O(n)
        for i in range(len(data) - 1):  # nested O(n)

            if data[i] > data[i + 1]:
                data[i], data[i + 1] = data[i + 1], data[i]
    return data
```

Since the second loop is nested we will multiply the complexity of it with the complexity of the first loop. Which is O(n) * O(n) = $O(n^2)$

> *If the second loop were outside of the first one, we would sum them instead of multiplying because in this case the second loop will not be repeated as many times as the first loop.*

O(N³) Cubic Time

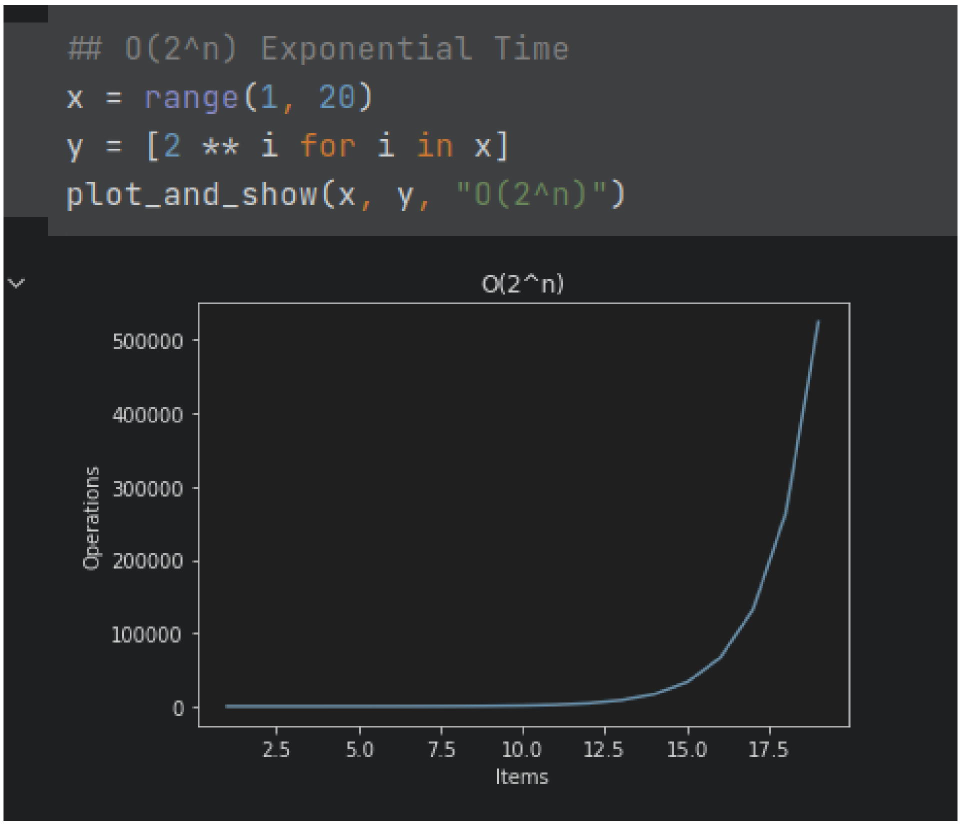


```
## O(n^3) Cubic Time
y = [i ** 3 for i in x]
plot_and_show(x, y, "O(n^3)")
```

The simplest example can be an algorithm with 3 nested for loops.

If you directly apply the mathematical definition of matrix multiplication then you will end up with an algorithm with cubic time. There are some improved algorithms for this task, take a look here.

O(2^N) Exponential Time



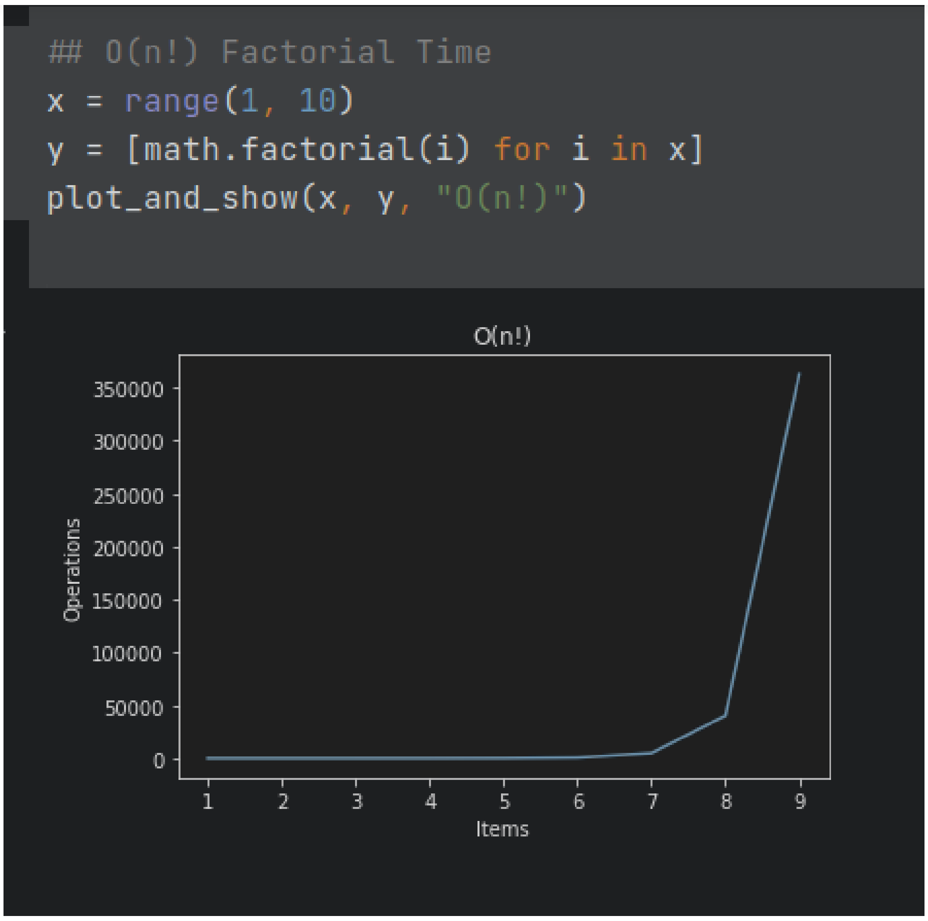
```
## O(2^n) Exponential Time
x = range(1, 20)
y = [2 ** i for i in x]
plot_and_show(x, y, "O(2^n)")
```

The most known example for this notation is finding the nth Fibonacci number with a recursive solution.

```
def nth_fibonacci(n: int) -> int:
    if n in [1, 2]:
        return 1

    return nth_fibonacci(n - 1) + nth_fibonacci(n - 2)
```

O(N!) Factorial Time



```
## O(n!) Factorial Time
x = range(1, 10)
y = [math.factorial(i) for i in x]
plot_and_show(x, y, "O(n!)")
```

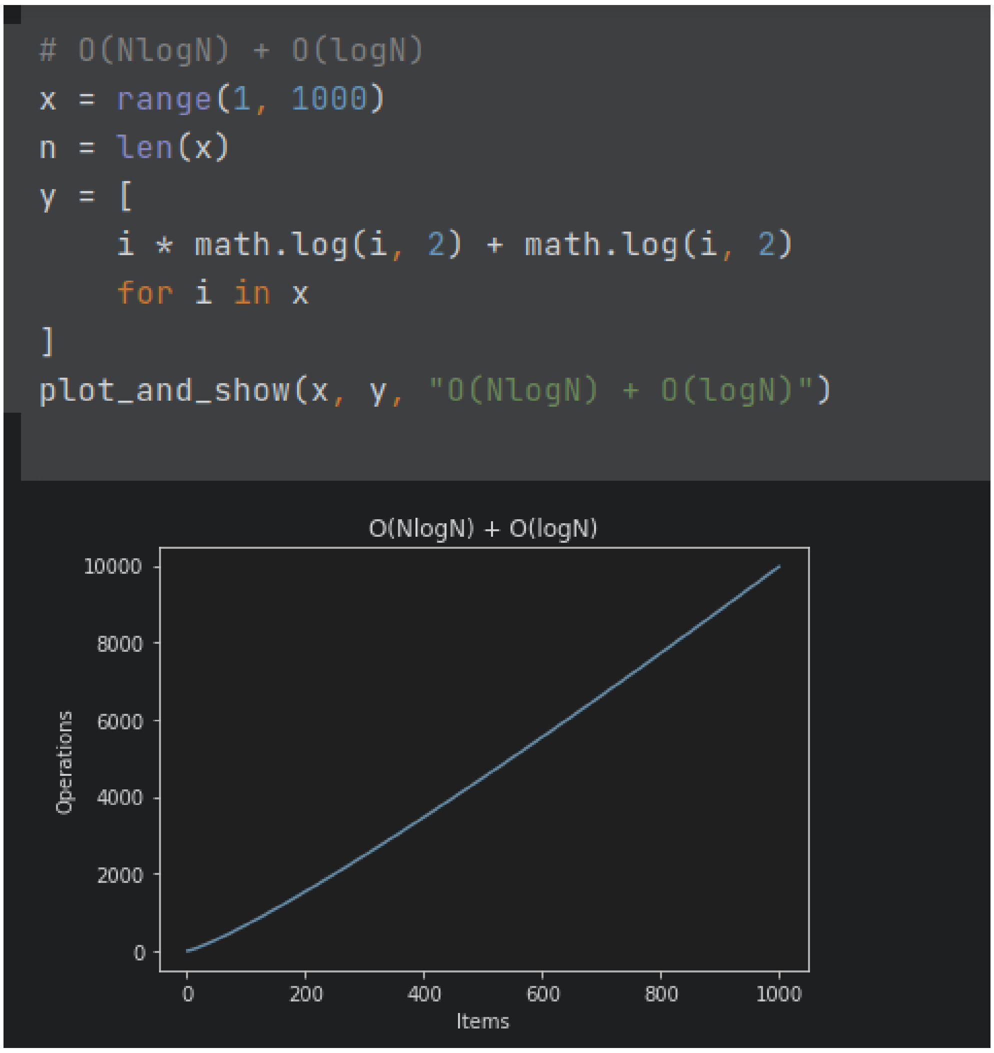
An example of this would be to generate all the permutations of a list. Take a look at the Traveling Salesman Problem.

Take the most important factor

We've already talked about dropping the constants when calculating the complexity of an algorithm because they don't provide us any value. There is something more regarding that rule. When performing complexity analysis we can end up with an algorithm that performs more than 1 type of operation to the input given. For example, we may need some function to initially sort an array then search on it. Let's assume that this will be one operation to sort with complexity O(NlogN) plus another one to search with complexity O(logN).

The time complexity for such a function will be O(NlogN) + O(logN). Let's visualize this:

Get unlimited access Open in app



```python
# O(NlogN) + O(logN)
x = range(1, 1000)
n = len(x)
y = [
    i * math.log(i, 2) + math.log(i, 2)
    for i in x
]
plot_and_show(x, y, "O(NlogN) + O(logN)")
```

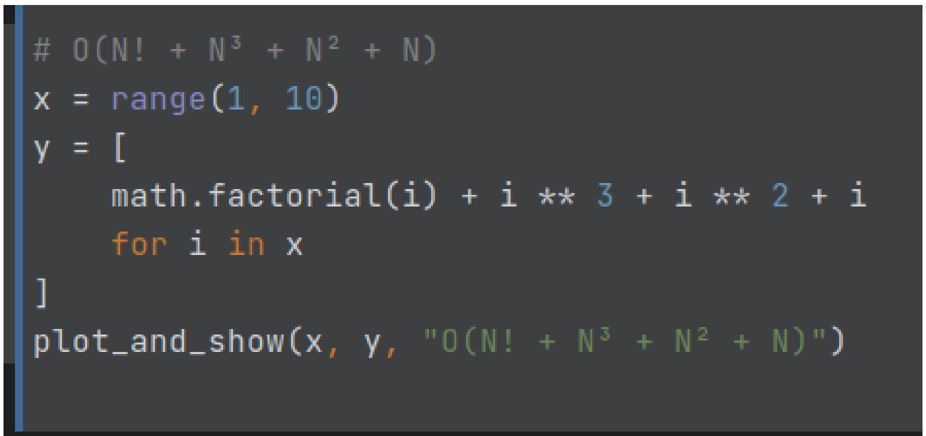
If you take a look at this graph, you will notice that the impact of O(NlogN) is bigger than the impact of O(logN) since the graph is more similar to the one of O(NlogN) compared to O(logN). We can even mathematically show that by doing so.

```
O(NlogN) + O(logN) = O((N+1)logN)  # factorize
O((N+1)logN) = O(NlogN)  # drop constant 1
```

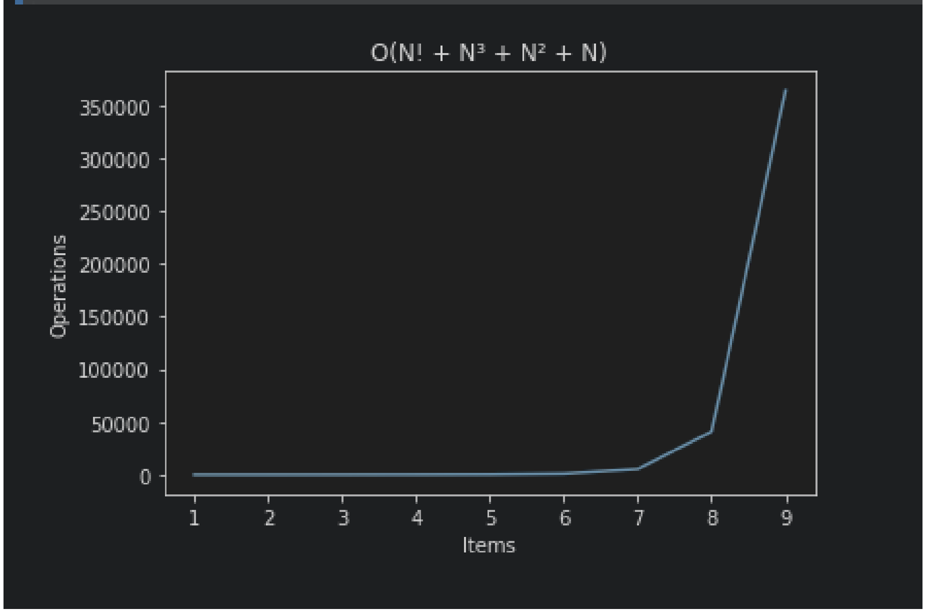
In this case, they are relatively close to each other and the difference is not obvious but if we take another example like $O(N! + N^3 + N^2 + N)$ we will notice that the impact of the notations except N! is so small compared to N! when N gets too large!

We can easily compute 1 000 000 ^ 3 but try the same for 1 000 000 factorial.

Get unlimited access
Open in app



```
# O(N! + N³ + N² + N)
x = range(1, 10)
y = [
    math.factorial(i) + i ** 3 + i ** 2 + i
    for i in x
]
plot_and_show(x, y, "O(N! + N³ + N² + N)")
```



> *The factorial of 10 is 3 628 800 whereas 10^3 is only 1000. As you can see the impact of N^3 is so small compared to the N! that we can actually ignore it. That is why when we have multiple notations summed up we take the most important factor.*

Something very important to know when taking the most important factor is that we group factors by the inputs. This means if we have an algorithm that operates on 2 different arrays one of size N and one of size M and the complexity of the algorithm is $O(N^2 + N + M^3 + M^2)$ then we cannot just say that the highest factor is M^3 so the complexity is $O(M^3)$. This is not true, because they are completely separate variables in our function. Our algorithm depends on both of them to work so what is correct is to take out the highest factors for both variables. We eliminate N since N^2 is higher, and we eliminate M^2 since M^3 is higher and the result is $O(N^2 + M^3)$.

Conclusion

If you want to learn algorithms and data structures deeply then you need to question everything. Do not take as granted any of the rules. Question them and try to find answers. Visualization makes a

Get unlimited access Open in app

References

- My Jupyter notebook for visualizations
- My Github repo on algorithms
- Binary search visualization
- Merge sort visualization
- Matrix multiplication
- Travelling Salesman problem

Sign up for Coffee Bytes

By Better Programming

A newsletter covering the best programming articles published across Medium Take a look.

Emails will be sent to a.v.nesterovich@gmail.com. Not you?

Get this newsletter