AlvBarros
Posted on 13 nov.

❤️ 122 🦄 9 🤯 11 🙌 10 🔥 8

Learn Big O Notation once and for all

#interview #beginners #algorithms

Introduction

Recently I was doing a job interview to a position that I really wanted in a very cool company, and one of the steps was the dreaded code interview where we solve LeetCode problems live.

I got the solution, and when asked the big O function for my solution, I answered correctly, but I was very confused and probably stumbled my way into it by simply counting the loops.

In order to not fail anymore job interviews in the future, I'm revisiting this topic some years after first learning about it in college.

The main objective behind this post is to provide a quick summary and a refresher for me to read before a coding interview. While I learn by writing, it is also important to store this somewhere I can always revisit when I need. And hey, maybe it can work for you too.

Big thanks to NeetCode for providing so much material and teaching all of this stuff for free.

What is Big O time complexity?

> In computer science, big O notation is used to classify algorithms according to how their run time or space requirements grow as the input size grows. [...] [It] characterizes functions according to their growth rates: different functions with the same asymptotic growth rate may be represented using the same O notation.
>
> - Source: Wikipedia

Or, in other words, it's a way to analyze the amount of time of our algorithm takes to run as the input grows. **O** is meant to be the whole operation, and **n** the input.

Let's look at some examples and it will make more sense.

O(n) - Sure, give me an example

Perhaps the easiest example to understand is of O(n), where the growth rate is linear.

- Given an unsorted array **n**, write a function that will return the biggest value.

To solve this, we need to go through every item in the array **n** and store it whenever we find a value bigger than the previous found.

```python
n = [3, 1, 4, 1, 5, 9, 2, 6, 5, 3, 5] # Initialize the array of n
def find_max_value(arr):
    # Initialize the maximum value with the first element of the array
    max_value = arr[0]

    # Iterate through the array to find the maximum value
    for num in arr:
        if num > max_value:
            max_value = num

    return max_value
print(find_max_value(n)) # Output: 9
```

The previous algorithm will always run through every item inside **n** at least once - it has to, because the array is unsorted.

Because of this, we say this algorithm has time complexity of **O(n)**, because as the array size (n) grows, the runtime grows in a linear fashion.

It also does not care about the non-constant attributes of your algorithm. Imagine that your algorithm iterates through every item in your **n** exactly twice, resulting in your time complexity of **O(2n)**. We simplify it by saying it is O(n), because the priority of the Big O notation is to **convey the shape of the growth in run time**.

O(1) - First and only exception

After telling you that we shouldn't care about non-constant values in the notation, We have to discuss the **O(1)**, where the **n** is not even present in this classification. It is perhaps the most desirable rate, where the time does not grow with the input, staying constant. For example:

- Given a non-empty array, return the first element.

```
n = [3, 1, 4, 1, 5, 9, 2, 6, 5, 3, 5] # Initialize the array of n
def first_element(arr):
    # Return the first element of the array
    return arr[0]
print(first_element(n)) # Output: 3
```

Because we don't actually iterate through any items in the array, the notation for this operation would be **O(1)**.

Some other examples of this include appending items to an array, removing (`pop`), or when using Hash maps (or Dictionaries) where we simply lookup using an index - like the algorithm above.

O(n^2) - This seems easy enough

The simplest case for this notation is when you have **nested loops**, or a two-dimensional array and you have to go through them to find what you're looking for.

- Given a number of sides in a dice, calculate every possible combination when using two dices of the given size

```
def dice_combinations(sides):
    # Initialize combinations array
    combinations = []
    # Iterate through first side
    for i in range(1, sides + 1):
        # Add every combination possible
        for j in range(1, sides + 1):
            combinations.append((i, j))
    return combinations

sides = 6  # Example for a 6-sided dice
print(dice_combinations(sides))
# Output: An array with 36 items (6 * 6)
# [(1, 1), (1, 2), (1, 3), (1, 4), (1, 5), (1, 6), (2, 1), (2, 2), (2, 3), (2
```

But what if the dice have different sides?

O(n*m) - Okay, now you're just adding letters

If instead you had to calculate the possible combinations using two dice of different sides, they would work as follows.

- Given two numbers of sides in a dice, calculate every possible combination when rolling these two dices.

```
def two_dice_combinations(sides1, sides2):
    # Initialize combinations array
    combinations = []
    # Iterate through first side
    for i in range(1, sides1 + 1):
        # Add every combination possible
        for j in range(1, sides2 + 1):
            combinations.append((i, j))
    return combinations

sides1 = 6  # Example for a 6-sided dice
sides2 = 8  # Example for an 8-sided dice
print(two_dice_combinations(sides1, sides2))
# Output: An array with 48 items (6 * 8)
# [(1, 1), (1, 2), (1, 3), (1, 4), (1, 5), (1, 6), (1, 7), (1, 8), (2, 1), (2
```

Keep in mind that these can work indefinitely. You can have an **O(n^3)** algorithm if we had three dice, or even **O(n^5)** - math does not impose a limit.

O(log n) - What?

Most people don't even understand what *log* means, and simply memorize that this notation is used when doing some sort of **binary search**.

This is the case when for every iteration of the loop, we divide the loop in half for the next iteration. The **log n** part then becomes **how many times can we divide n by 2 to get the result** - which is kind of the definition of this log notation when the base is 2.

When working with **binary threes** we have to traverse the nodes, and on each node we have to make a decision - go "left" or go "right". This is already splitting the amount of operations in half since we're only going into one direction (don't mind that the nodes may have a different amount of child nodes).

This is one of the best algorithms since the run time grows very slowly. For really big input sizes, the time is basically a flat line.

The most common example of O(log n) is when we're doing a **binary search**.

I won't get into too much detail, but basically a binary search can be used when we have a **sorted array** where we want to find the index of a specific value.

- Given a sorted array, find the index of a target value

```
def binary_search(arr, target):
    # Initialize the left and right pointers as the first and last
    left, right = 0, len(arr) - 1

    # Continue searching while the left pointer is less than or equal to the
    while left <= right:
        # Calculate the middle index
        mid = (left + right) // 2

        # Check if the middle element is the target
        if arr[mid] == target:
            return mid
        # If the middle element is less than the target, adjust the left poin
        elif arr[mid] < target:
            left = mid + 1
        # If the middle element is greater than the target, adjust the right
        else:
            right = mid - 1

    # Return -1 if the target is not found in the array
    return -1

# Example usage:
sorted_array = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
target = 7
print(binary_search(sorted_array, target)) # Output: 6 - sorted_array[6] == 7
```

Note that the loop is not in the familiar `for i in range(1, n)`, but rather the middle between the `left` and `right` indexes.

O(n log n) - Now you're just making stuff up

The only reason this is here is because it is very hard to intuitively figure this out.

This notation is commonly found in sorting algorithms and in fact is the most common for built-in sorting functions in modern languages.

Take, for example, the **Merge Sort**. Just so we do not get into too much detail, it basically works by dividing the array into two halves recursively (log n divisions) and then merges the halves back together in linear time (O(n) for each merge). By combining these two steps, we have **O(n * log n)**.

- Given an unsorted array, sort it by using merge sort.

```python
def merge_sort(arr):
    if len(arr) <= 1:
        return arr

    # Find the middle point and divide the array into two halves
    mid = len(arr) // 2
    left_half = arr[:mid]
    right_half = arr[mid:]

    # Recursively sort the two halves
    left_sorted = merge_sort(left_half)
    right_sorted = merge_sort(right_half)

    # Merge the sorted halves
    return merge(left_sorted, right_sorted)

def merge(left, right):
    sorted_array = []
    left_index, right_index = 0, 0

    # Merge the two arrays while maintaining order
    while left_index < len(left) and right_index < len(right):
        if left[left_index] < right[right_index]:
            sorted_array.append(left[left_index])
            left_index += 1
        else:
            sorted_array.append(right[right_index])
            right_index += 1

    # Append any remaining elements from the left or right array
    sorted_array.extend(left[left_index:])
    sorted_array.extend(right[right_index:])

    return sorted_array

# Example usage:
unsorted_array = [3, 1, 4, 1, 5, 9, 2, 6, 5, 3, 5]
print(merge_sort(unsorted_array)) # Output: [1, 1, 2, 3, 3, 4, 5, 5, 5, 6, 9]
```

We can see in the code above that every time `merge_sort` is called, it ends by calling `merge`. And every time merge_sort is called, it also calls itself twice, one for each half of the array.

O(2^n) - I should have seen it coming

This notation is usually found when we have a recursion algorithm that branches out in two ways.

We can easily see these complexities in **bubble sort**, but as we can't talk about algorithms without talking about Fibonacci, let's finally do it. But remember - there are more efficient ways to solve this problem.

- Given an index, find the number in the Fibonacci sequence.

```python
def fibonacci(n):
    if n <= 1:
        return n
    branch1 = fibonacci(n-1)
    branch2 = fibonacci(n-2)
    return branch1 + branch2

# Example usage:
print(fibonacci(5))  # Output: 5
```

It is clear in the implementation above that two branches are created for every iteration of the recursive loop.

For example, for `n = 5`, we would have `fibonacci(4)` and `fibonacci(3)` be called, which would generate `fibonacci(3)` **again** (we have not implemented memoization in the above algorithm), `fibonacci(2)` **twice** and `fibonacci(1)`. You can visualize it as an "upside down binary tree", where the height of the tree is **n**.

Theoretically we could have any number being raised to the power of **n**, such as **O(3^n)** and **O(5^n)**.

O(n!) - Make it end, please!

If you don't know what the **!** means, we simply multiply the number by every number - 1 until we get to 1 (which we can ignore).

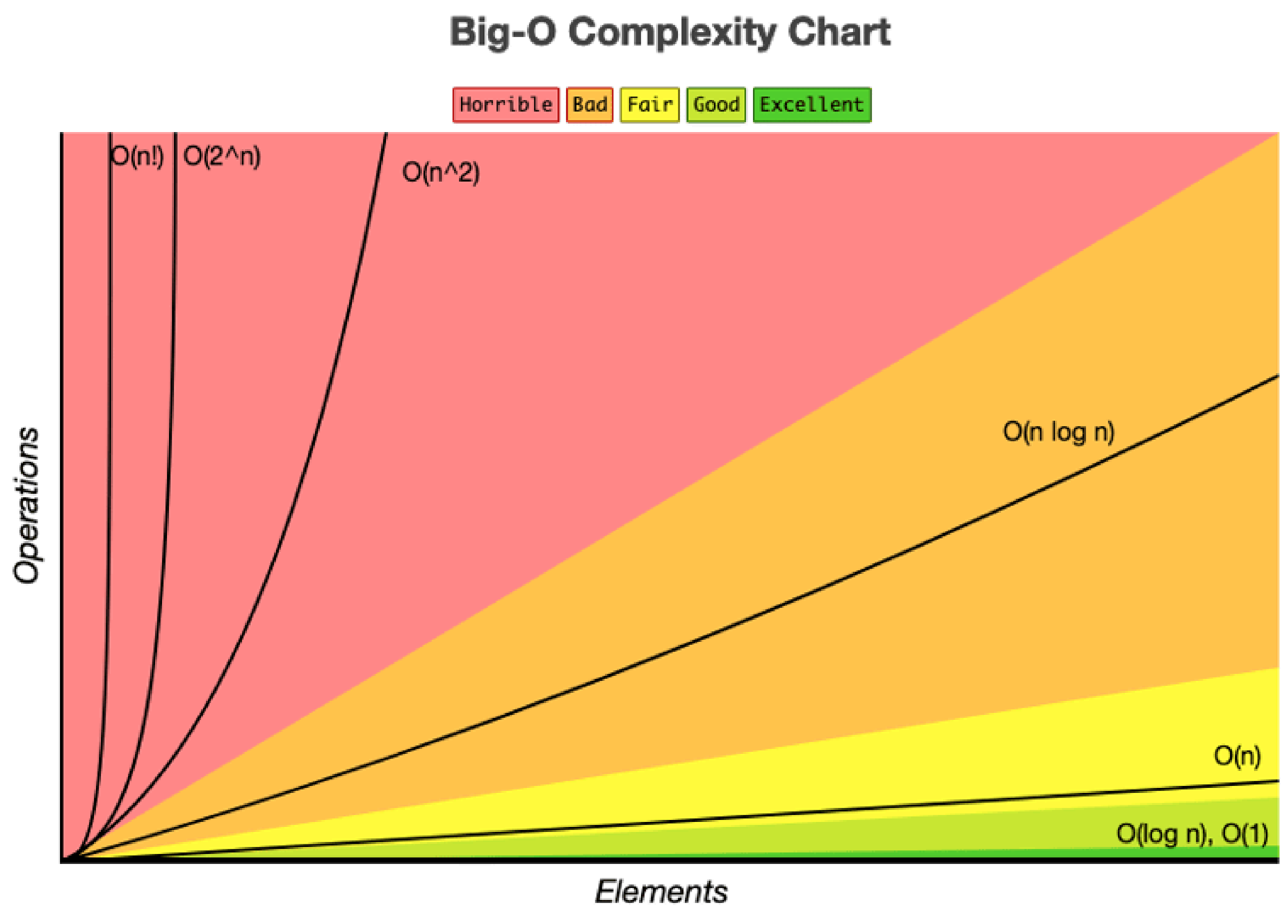
> For example:
> 5! = 5 * 4 * 3 * 2 = 120

You can think of this as an algorithm that for every iteration, remove an item and run again. It mainly comes up in permutations or, perhaps more famously, in the Traveling Salesman Problem.

For this one, I won't be adding any piece of code since this can get very complicated and this is extremely rare anyways, because if you have an algorithm of **O(n!)** you most definitely don't have the optimal solution.

So there you have it!

You can refer to the graph below to see how the algorithms compare. The vertical axis means the number of operations (or also the time) and the horizontal axis means the amount of elements (or the value of **n**). Special thanks to Eric Rowell for the cheatsheet!



Available at https://www.bigocheatsheet.com/

I hope you've found this post useful, and good luck in your future studies! 🤞

Your step-by-step guide to building from scratch

Experience hands-on learning resources without setup or configuration, and learn to develop quicker than ever before.

Get started

Top comments (13)

Purvesh Panchal • 18 nov.

Nice explanations!!
Ps. I did the 69th Like

AlvBarros • 18 nov.

hehe Nice!

KMohZaid • 29 nov.

💎

Efren Marin • 13 nov.

This is a nice condensed version of learning Big O. Neetcode and Frontend Masters have amazing resources too.

Shayan Ansari • 20 nov.

great explanation, i now finally understand this

Hammad • 19 nov.

Very Nice and easy understanding! great job

ANIRUDDHA ADAK • 21 nov.

just wow ♥.

Marcelly Paiva • 13 nov.

Amazing, thank you for this! I specially love the examples.

ThisTish • 13 nov.

This helped me so much. I will return to this often. Thank you!

Joel • 15 nov.

I recently watched CS50’s lesson about how complexity works and finally got the idea on my mind. Your article is gold man, thank you!

Aravind • 17 nov.

Too good and Too precise with examples. Thanks for the share dude.

Mohamed Barakat • 22 nov.

Nice

View full discussion (13 comments)

Code of Conduct • Report abuse

Auth0 PROMOTED

We had a glow-up.

Late nights troubleshooting? Let us help you upgrade your auth. Check out how we improved our plans for devs for free today.

Learn more

AlvBarros

Started as a hobby, now a full-time software developer. Enthusiastic about anything tech-related. Brazilian.

LOCATION
Brazil

WORK
Software Developer

JOINED
15 août 2023

More from AlvBarros

Dependency Injection in Flutter
#flutter #architecture #development #beginners

AWS PROMOTED

[Innovation streaming live from AWS re:Invent]

Join hosts from AWS and AWS Partners as they discuss the latest news, trends, and strategies for driving success on the cloud. Streaming live from AWS re:Invent in Las Vegas.

Register now