Understanding Big O notation

Master the basics of Big O notation

Dinesh Kumar K B Follow
Nov 8, 2020 · 8 min read

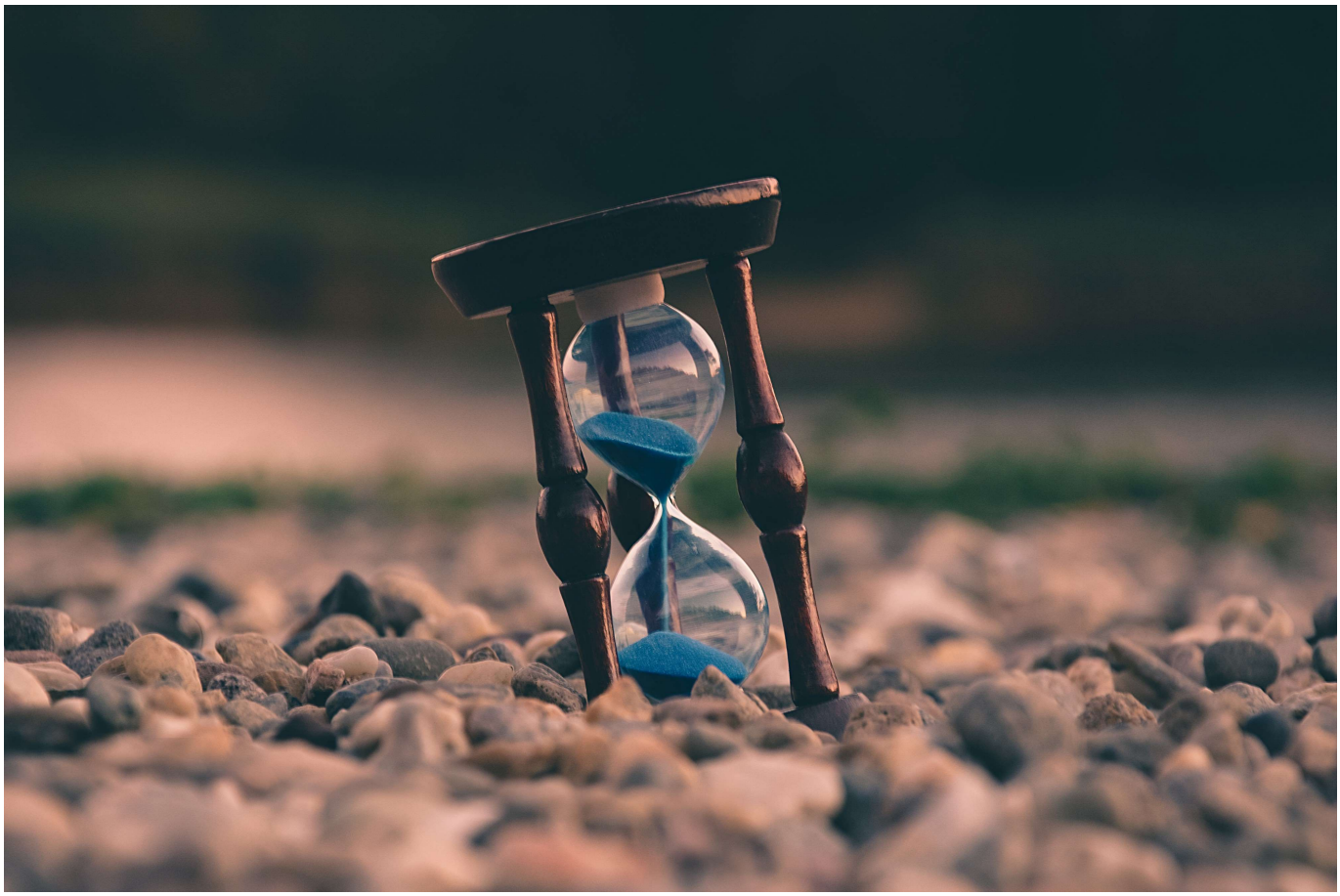


Photo by Aron Visuals on Unsplash

Note: For non-members, this article is also available at https://dineshkumarkb.com/tech/master-the-basics-of-big-o-notation/

What is Big O?

Big O is a term used to represent the **efficiency of an algorithm**. It is mandatory for a programmer to master the basics of Big O to clearly specify how fast or slow his algorithm could perform.

Big O is also used to represent the space taken by a program in-memory during the program execution.

So, in short, Big O is used to represent both the time and space which are popularly identified as

1. **Time-complexity**
2. **space-complexity**

This article predominantly focuses on the time-complexity though we may slightly touch-base on the space complexity too. The examples provided here are in `python`, however this applies to any programming language.

What is a time-complexity of an algorithm?

Time-complexity is the time taken by an algorithm to complete its execution.

Let's look at an example of printing an array of n elements.

Example — 1

```
# A program to print n elements of an array

def display_number(n):
    for i in range(n):
        print(i)


display_number(10)
```

So when asked what is the time-complexity of this algorithm, how do we answer ? Well, for starters,

1. Check the variables in the program(Here variables mean values that vary).

2. Try asking yourself, varying what value would impact the run time of the program

Let's try applying this concept for the `display_number` program. We have 2 variables here speaking literally. One is `i` and the other one is `n.` Well, it doesn't make sense to consider i because i will just iterate through n. So, let's forget about i.

Will the run time vary depending upon n? Of course yes. In the example the value of n is 10.

What if the value of n is 1000 or 100000 ?

That would definitely impact the runtime of the program wouldn't it?

So now, what is the time-complexity of the above program? It's definitely O(n). As the value of n linearly increases, so will the time complexity. This is just a basic example.

We have certain ground rules while calculating the Big O. The Big O is not meant to calculate the accurate time steps or precise time taken in milliseconds. Rather, it is meant to describe **the rate of the increase of the time taken for an algorithm.**

Ground rules while calculating the Big O:

1. Drop the constant values
2. Drop the non-dominant terms

1.Drop the constant values.

Why drop the constant values?

Remember,the Big O is used to calculate the rate of increase of the time-taken of an algorithm as the size of the input varies? The constant factor neither impacts nor contributes to the rate of increase of time-taken to execute a program. Arguably, it does contribute. However,the contribution by a constant is predictable and "constant" when it comes to the rate of increase.

Well, because it is a constant irrespective of the size of input.

Our objective is to calculate the rate of increase of the time-taken by an algorithm as the size of the input varies.Therefore, the contributing factor will be the variable and not the

constant. So we can very well drop the constants while calculating the Big O.

Let's look at an example for this.

Example — 2

```
1
2  def display_number(n):
3
4      for i in range(n):
5          print(i, end="")
6
7      print()
8
9      for i in range(n-1, -1, -1):
10         print(i, end="")
11
12
13 display_number(10)
```

display_numbers.py hosted with ♡ by GitHub view raw

```
Output:

0123456789
9876543210
```

The above program will print n numbers both in ascending and descending orders. Let's calculate the time complexity for this.

The program has 2 `for` loops. So, time complexity will be `O(n + n) = O(2n)` ideally. Why? Because it traverses through n elements twice. Once to print elements in ascending and second time to print elements in reverse. If we drop the constant value 2, this narrows down to O(n).

So, how does it differ from the example-1 of printing n elements in an array? The first one has only 1 for loop whereas the second has 2.But all we need is the rate at which the execution time would increase, not the exact value.So it's O(n) for both the programs.

As and when n increases or decreases, the time complexity is slower or faster respectively.Period.

> *Note : One important thing to remember is, in example-2 both the for loops are iterating through the same array n. I repeat, both the for loops are iterating through the same array.*

2.Drop the non-dominant terms

Dominant terms are basically values that will have a bigger impact on the algorithm. For example, n^2 is dominant compared to n.n is dominant compared to log n and so on.

So, when the run time is $O(n^2 + n)$ we can drop the non-dominant n and define the run time as $O(n^2)$.

```
1
2  def print_coord(n):
3
4      for i in range(n):
5          for j in range(n):
6              print(i,j)
7
8      for i in range(n):
9          print(i)
10
11 print_coord(10)
```

print-coordin.py hosted with ♡ by GitHub view raw

We have 2 for loops here. The first one is a nested for loop for which the runtime is $O(n^2)$ and the other is $O(n)$. Together, execution time is $O(n^2+n)$.

As the value of n increases, $O(n)$ becomes negligible while compared to $O(n^2)$. Therefore, we drop the non-dominant terms.

Note : $O(n^2)$ is not always slower than n. For specific smaller values of n, $O(n^2)$ could be faster than $O(n)$. Likewise $O(n)$ could very well be faster than $O(1)$.

Now that we have established the rules, we will look at the most predominant run times.

O(1):

This means that the time-complexity is always constant. Whatever, the size of input is, the execution time is always constant.

```
def get_x(x):
  return x+1
```

The time-complexity of the above method is O(1). why not O(x)? Irrespective of the value of x, the program just adds 1 to the value of x and returns the same. So the execution time is always constant.

O(n):

The time complexity increases linearly with the value of n. We have already seen examples for this like printing all n numbers in example — 1.

Here is another algorithm to calculate the sum of n numbers.

```
"""
Calculate the sum of n numbers
"""

def add_values(n):

    summ = 0

    for i in range(n):
        summ += i

    return summ


print(add_values(1000))
```

summ.py hosted with ♡ by GitHub view raw

The execution time increases linearly as the value of n increases.

O(n²):

Let's look at an algorithm to sort n numbers.

```
"""
This algorithm sorts an array using bubble sort

"""


def bubble_sort(lst):

    for i in range(len(lst)):
        for j in range(len(lst)):
            if lst[i] < lst[j]:
                lst[i], lst[j] = lst[j], lst[i]

    return lst


print(bubble_sort(lst=[98,43,21,13,55,76,88,33]))

```

bubble_sort.py hosted with ♡ by GitHub view raw

```
Output:

[13, 21, 33, 43, 55, 76, 88, 98]
```

This is a bubble sorting algorithm,which compares every adjacent element and then swaps their places. Let's assume the length of the list is n.Every element is compared with every other element in the array. Let's say if the size of the `list` is 10, every element is compared with every element in the array, which results in the array being traversed 10*10=100 times.

To summarise, whatever we discussed so far was traversing through the same `list`. What if we have to traverse through 2 different `lists` in a nested for loop?

```
def merge_lists(x,y):

    for i in x:
        for j in y:
```

```
            print(i,j)


merge_lists([1,2,3,4,5],[6,7,8,9,10,11,12])
```

So, is the time-complexity $O(n^2)$ for the above program? No.

Please note that x and y are 2 different lists of 2 different lengths and we are trying the pair the elements of x and y. Therefore the time-complexity for this program is O(x*y).

O (log n):

This is a binary search algorithm to search an element recursively.

```
"""
This is a program for a binary search of an element.
In every iteration, the search list is halved.

For example in the below program
Iteration 1 : The list is [2,3,4,5,6,7,8,9,10,11]
Iteration 2 : [7, 8, 9, 10, 11]
Iteration 3 : [9, 10, 11]
Iteration 4 : [10, 11]


"""

def search(n, lst):

    print(f" The list is {lst} ")

    pivot = len(lst)//2

    print(f" Pivot element : {pivot}")

    if n == lst[pivot]:
        return "Element found"
    elif n < lst[pivot]:
        lst = lst[:pivot]
        return search(n, lst)
    elif n > lst[pivot]:
        lst = lst[pivot:]
        return search(n, lst)

```

```
31
32  print(search(11, lst = [2,3,4,5,6,7,8,9,10,11]))
```

bsearch.py hosted with ♡ by GitHub view raw

If we observe the pattern here, on every iteration, the input list to be searched is halved.

```
For example in the below program
Iteration 1 : The list is [2,3,4,5,6,7,8,9,10,11] -- 10 elements
Iteration 2 : [7, 8, 9, 10, 11] -- 5
Iteration 3 : [9, 10, 11] -- 3 elements
Iteration 4 : [10, 11] -- 2 elements
```

This is repeated until the element to be found is equal to pivot. The total runtime is equal to the execution time of number of steps until the element is equal to pivot.

So if N has 16 elements, it is divided as 8,4, 2 and 1.If we represent this in powers of 2,

```
1 -- 2 power 0
2 -- 2 power 1
4 -- 2 power 2
8 -- 2 power 3
16 -- 2 power 4
```

So, 2^x = N ==> $2^4 = 16$

Representing this in logarithm, if N is 16,

log2 16 = x. Here the base is 2 and argument is 16. What value of x would bring 16. 2^4 is 16. Therefore, log 16 = 4. We don't need the base either as it wouldn't matter. So we have dropped the base value 2.

Therefore this is represented as O(log n).If this is too much,remember that any algorithm where the input list is halved, we can predict the algorithm to be O(log n).Same execution time is applied to searching an element in a binary search tree.

O(A+B):

```python
def multi_part_add(a,b):

    add1 = 0
    add2 = 0

    # sum all the elements in a
    for i in range(a):
        add1 += i

    # sum all the elements in b
    for j in range(b):
        add2 += j

    print(add1,add2)


multi_part_add(5,10)
```

multi-part.py hosted with ♡ by GitHub view raw

The time complexity of the above algorithm is O(a+b). Here a and b are two different lists. Therefore this cannot not be O(2n) or O(2a). We cannot derive similarities between a and b as they are completely independent.

So, the time complexity is O(a+b) the sum of time taken to traverse through a and time taken to traverse through b. The run time is impacted when either a or b varies.

Run time for Recursive algorithms:

Let's look at a recursive program to calculate the sum of n numbers.

```python
"""
Return the sum of given n numbers using recursion
"""



def calc_sum(n):
    """
    This function calculates the sum using time complexity : O(n) and space complexity O(n).
    At any given point of time the stack size is equal to n as we use recursion
    :param n:
    :return:
    """
```

```python
    if n <= 0:
        return 0
    else:
        print(f" Calculating {n} + calc_sum({n-1})")
        return n + calc_sum(n-1)

print(calc_sum(5))
```

sumn.py hosted with ♡ by GitHub view raw

The time complexity of this algorithm is O(n). Let's quickly brush upon space complexity.

Space Complexity:

Space complexity is the amount of in-memory space taken by the algorithm during its execution. During a program execution, every function is executed and placed on a stack with its arguments and once the execution is complete, the stack is popped. That's why we call the errors a stack trace and the entire stack trace is printed during errors.

For the above program, the space complexity is O(n). The reason is at any given point of time, there are n calls in the stack unlike the linear execution.

The stack trace roughly looks like below.

Search this file…

sum(5)	Function call
sum(5-1)	Iteration 1
sum(4-1)	Iteration 2
sum(3-1)	Iteration 3
sum(2-1)	Iteration 4

stacktrace.csv hosted with ♡ by GitHub view raw

Only after it hits the base case (n=1), the program starts returning. Therefore, both the time and space complexities are O(n).

The same program can be rewritten for O(n) time complexity and O(1) space complexity like below.

```python
def calc_sum_wo_recursion(n):
```

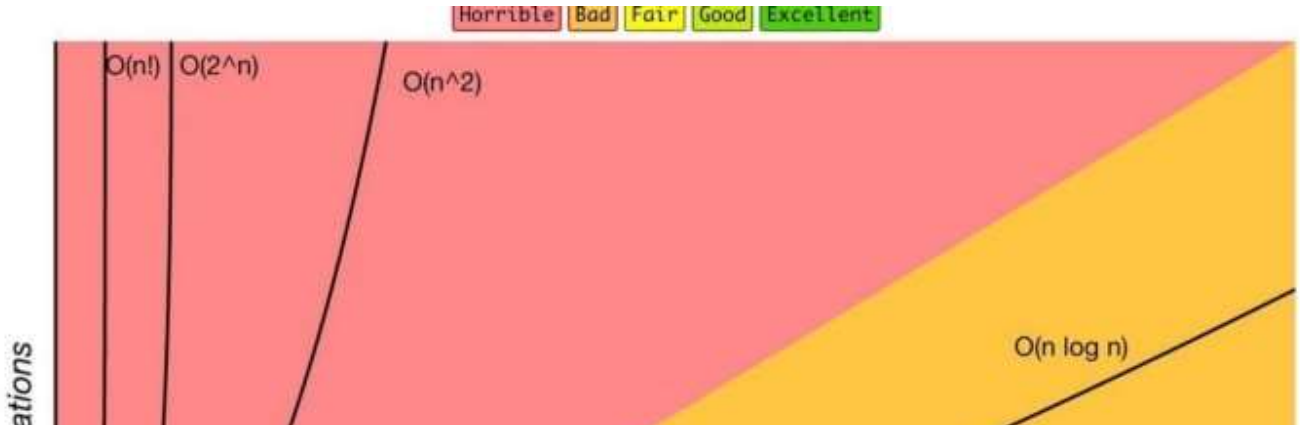
```python
def calc_sum_wo_recursion(n):
    """
    Here the time complexity is O(n) and the space complexity is O(1) unlike the previous one
    :param n:
    :return:
    """

    add = 0
    for i in range(n+1):
        print(f" Adding {i} ")
        add += add_num(i)
    return add


def add_num(a):
    return a


print(calc_sum_wo_recursion(5))
```

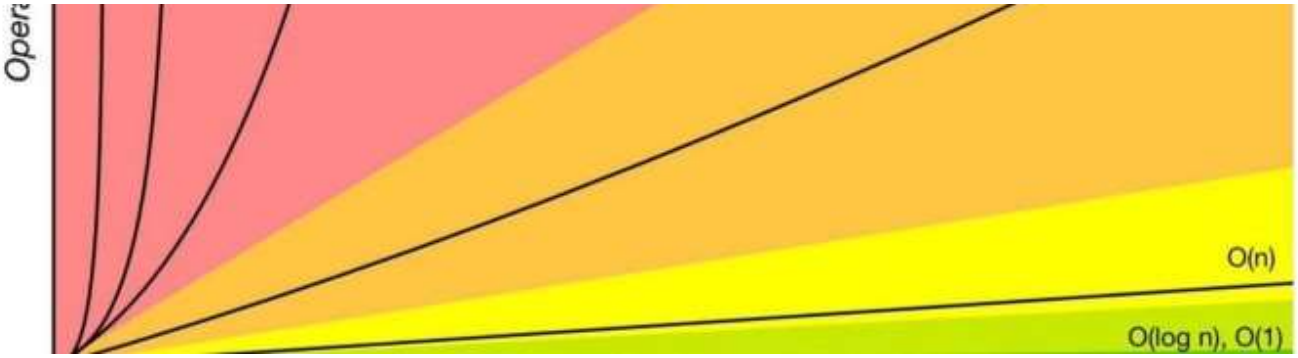
sumworecursion.py hosted with ♡ by GitHub view raw

This program has the same time complexity of O(n) as the previous one. However, this is not a recursive addition.During the execution, `add_num` is called every time and the moment it returns a, the call is not available in memory anymore. So the space complexity is O(1) at any given point of time.

> *Note: This is not an exhaustive list of algorithm run times. There are many more different run times like O(2^n), O(n!) etc which is not covered here. However, this should be good to start with.*

Below is an image of common Big O run times.





Summary:

1. Big O is used to represent the time taken by an algorithm to complete its execution
2. Big O is also used to represent memory taken by a program
3. Some common run times are O(n), O(n^2), O(log n) etc.

Programming Python Python3 Big O Notation Algorithms