

April 25, 2023

Big O notations



**Presented by
Amanuel Sisay**



Housekeeping Rules

- Stay attentive, we do not want you to miss anything
- Share your questions on the zoom chat

Agenda

Item	Time (GMT)
Importance of understanding time and space complexity	5
Go through Time complexity and how to get time complexity	20
Time complexity of different algorithms	20
QA	5



Be aware of
Kimba



The Framework



Importance of understanding time and space complexity

- Understanding time and space complexity is crucial for designing and evaluating efficient algorithms.
- Inefficient algorithms can waste system resources and slow down program execution, leading to poor performance and user dissatisfaction.
- Optimizing time and space complexity can reduce the cost of running a program, make it more scalable, and allow it to handle larger inputs or datasets.
- Knowledge of time and space complexity can help developers choose the best algorithm for a given problem, and improve their problem-solving skills in general.
- In some fields, such as scientific computing and big data analytics, the ability to design algorithms with optimal time and space complexity can be critical for making meaningful discoveries or insights.
- Finally, understanding time and space complexity is a fundamental part of computer science and software engineering education, and is an essential skill for any developer or engineer.

To do

Write an algorithm that takes list of numbers and a target number and checks whether the number is found in that list or not? Returns a boolean value 1 if it is found else 0.

```
int Find (int *nums, int target);
```


Classifications of Time Complexity

Time complexity analysis is a way to measure the **performance of an algorithm in terms of the time** it takes to execute as a function of **its input size**. Three common classifications of time complexity are:

- Best-case
- Worst-case
- and average-case time complexity.

Best-case time complexity

Best-case time complexity refers to the minimum amount of time an algorithm can take to execute for any possible input of size n . It is often denoted as $\Omega(f(n))$.

For example: consider an algorithm that finds if a number exists in a list of numbers, and returns true or false, when is the best time complexity?

Average-case time complexity

Average-case time complexity refers to the expected amount of time an algorithm takes to execute on an input of size n , averaged over all possible inputs. It is often denoted as $\Theta(f(n))$.

For example: for the algorithm we just wrote, when Average time complexity?

Worst-case time complexity

Worst-case time complexity refers to the maximum amount of time an algorithm can take to execute for any possible input of size n . It is often denoted as $O(f(n))$.

For example: consider the same algorithm? What is the case for the worst time complexity?

```
def findNum(listNums, target):
```

```
    # iterate and find the target
```

```
    for num in listNums:
```

```
        if num == target:
```

```
            return True
```

```
    # the target was not found
```

```
    return False
```

```
nums = [1,2,3,4,5,6,7,8,9,0]
```

```
target = 1
```

```
foundNum = findNum(nums, target)
```

```
print(foundNum)
```

```
# n = 10 which is the size of the list
```

```
print("Best case is:  $O(1)$ , searching for element at the beginning")
```

```
print("Average Case is:  $O(n)$ , searching for element in the middle or in the list")
```

```
print("Worst Case:  $O(n)$ , searching for element not there")
```

Time complexity

- Time complexity measures **how long it takes for a program** to run as the amount of data it needs to **process gets bigger**.
- Big O notation is a way of expressing time complexity using math.
- Big O notation helps us describe how **the running time of a program changes as the amount of data it needs to process increases**.
- We use symbols like "O" to describe how the running time changes as the input data size increases.
- For example, if a program's time complexity is " $O(n)$ ", it means the running time increases linearly as the input data size (" n ") increases.
- Understanding time complexity and Big O notation is important for **writing efficient programs and choosing the best algorithm for a given problem**.

Common Big O notations

Big O notation is used to describe the upper bound of an algorithm's time complexity.

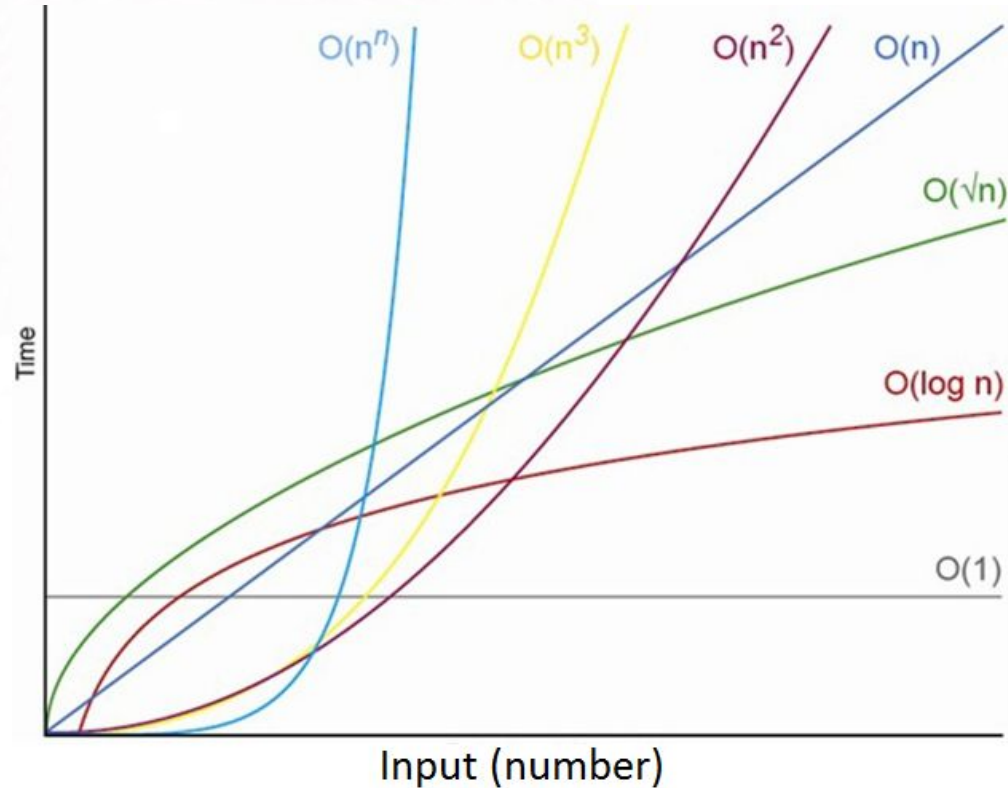
There are several common types of Big O notation that you may come across when studying algorithms and their time complexity.

Here are some of the most common types of Big O notation and what they mean:

- **$O(1)$: Constant time.** The running time of the algorithm does not change as the size of the input increases.
- **$O(\log n)$: Logarithmic time.** The running time of the algorithm increases slowly as the size of the input increases.
- **$O(n)$: Linear time.** The running time of the algorithm increases at the same rate as the size of the input.
- **$O(n \log n)$: Linearithmic time.** The running time of the algorithm increases at a rate between linear and logarithmic as the size of the input increases.
- **$O(n^2)$: Quadratic time.** The running time of the algorithm increases at a rate proportional to the square of the size of the input.

I hope this helps you understand the different types of Big O notation better!

Time Vs Input Number



How to determine time complexity of an algorithm

Determining the time complexity of an algorithm involves analyzing how the algorithm's running time grows with respect to the size of the input. Here is a step-by-step process to determine the time complexity of an algorithm using an example:

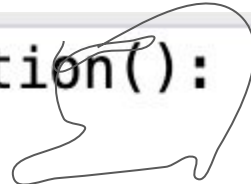
1. **Identify the input size:** The input size is usually denoted by "n" and represents the size of the input to the algorithm. For example, in the case of an array, the input size would be the length of the array.
2. **Identify the basic operations:** Identify the basic operations that are performed in the algorithm, and count how many times each operation is executed. Examples of basic operations include comparisons, assignments, arithmetic operations, and function calls.
3. **Express the number of operations as a function of the input size:** Express the total number of operations as a function of the input size "n". This function represents the running time of the algorithm.
4. **Simplify the function:** Simplify the function by removing any constant factors or lower order terms that do not significantly contribute to the overall running time.
5. **Determine the time complexity:** Determine the time complexity of the algorithm based on the simplified function. Common time complexity classes include $O(1)$, $O(\log n)$, $O(n)$, $O(n \log n)$, $O(n^2)$, and $O(2^n)$, among others.



Time: $O(1)$
Space: $O(1)$

alx

```
1 def constant_time_operation():  
2     x = 10  
3     y = x + 5  
4     return y  
5
```



Time: $O(n)$

Space: $O(n)$ if the input is counted, $O(1)$

```
1 def find_max(numbers):  
2     maximum = numbers[0]  
3     for number in numbers:  
4         if number > maximum:  
5             maximum = number  
6     return maximum
```

7

Time: $O(n)$

Space: $O(n)$ considering the recursive stack other wise $O(1)$

```
1 def factorial(n):  
2     if n == 0:  
3         return 1  
4     else:  
5         return n * factorial(n-1)  
6
```

Time: $O(\sqrt{n})$
Space: $O(1)$

main.py

+

```
1 def is_prime(n):  
2     if n <= 1:  
3         return False  
4     for i in range(2, int(n**0.5) + 1):  
5         if n % i == 0:  
6             return False  
7     return True  
8
```

Time: $O(2^n)$

**Space: $O(2^n)$ recursion stack
 $O(1)$**


```
1 def fibonacci(n):  
2     if n <= 1:  
3         return n  
4     else:  
5         return fibonacci(n-1) + fibonacci(n-2)  
6 |
```



Time: $O(n)$
Space: $O(1)$

alx

```
1 def compute_sum(n):  
2     total = 0  
3     for i in range(n):  
4         total += i  
5     return total  
6
```



Which one is better?

```
def find_element(elements, target):  
    for element in elements:  
        if element == target:  
            return True  
    return False
```

```
1 def binary_search(arr, target):  
2     left, right = 0, len(arr) - 1  
3     while left <= right:  
4         mid = (left + right) // 2  
5         if arr[mid] == target:  
6             return True  
7         elif arr[mid] < target:  
8             left = mid + 1  
9         else:  
10            right = mid - 1  
11    return False  
12
```


**Determine time complexity for
the algorithm we wrote**

Bubble sort

Works by repeatedly swapping the adjacent elements if they are in the wrong order.

Insertion sort algorithm

Insertion sort is a simple sorting algorithm that sorts an array by comparing each element with the previous elements and inserting it into the correct position in the sorted subarray.



Code snippet and What is time complexity?

```
def insertionSort(nums): #  $O(n^2)$ 
    # n -> is the length nums (number of elements in nums)

    for i in range(len(nums)): #  $O(n) \rightarrow O(n * 1 * n) \rightarrow O(n^2)$ 
        curr = i #  $O(1)$ 
        while curr > 0 and nums[curr] < nums[curr - 1]: #  $O(n)$ 
            nums[curr], nums[curr - 1] = nums[curr - 1], nums[curr]
            curr -= 1
        print(nums)
```

```
worstCaseNum = [9,8,7,6,5,4,3,2,1] #  $n * (n + 1) / 2$ 
                                     #  $n * (n + 1)$ 
                                     #  $n^2 + n$ 
                                     #  $O(n^2)$ 
```

```
bestCaseNum = [1,2,3,4,5,6,7,8,9] #  $O(n)$ 
```

```
averageCaseNums = [6,2,3,9,9,1,8,1,5,1,25,1] #  $O(n^2)$ 
insertionSort(worstCaseNum) #  $n^2$ 
print(num)
```

Analysis of time complexity of Insertion sort.

Best-Case Time Complexity:

- Occurs when the input array is already sorted.
- Only need to compare each element with its previous element and determine that it is already in the correct position.
- Time complexity of best-case is $O(n)$, where n is the size of the input array.

Average-Case Time Complexity:

- Occurs when the input array is randomly ordered.
- Time complexity of average-case is $O(n^2)$, where n is the size of the input array.

Worst-Case Time Complexity:

- Occurs when the input array is in reverse order.
- Each element needs to be compared with all the previous elements in the sorted subarray and shifted to its correct position.
- Time complexity of worst-case is $O(n^2)$, where n is the size of the input array.

Counting Sort

Counting Sort... N=10 , K=5

Input Array A.

3	4	2	1	0	0	4	3	4	2
0	1	2	3	4	5	6	7	8	9

Count Array C.

0	0	0	0	0
0	1	2	3	4

Result Array B.

0	0	0	0	0	0	0	0	0	0
0	1	2	3	4	5	6	7	8	9

Code snippet and What is time complexity?

```
def countSort(nums): #  $O(n + (n + k)) \rightarrow O(2n + k) \rightarrow O(n + k)$ 
    n = max(nums)
    count = [0] * (n + 1)
    # n -> length of nums
    # k -> length of count or the maximum number

    for num in nums: #  $O(n)$ 
        count[num] += 1

    print("count array", count)
    ptrCnt = 0
    for i in range(len(nums)): #  $O(n + k)$ 
        while count[ptrCnt] == 0:
            ptrCnt += 1
        count[ptrCnt] -= 1
        nums[i] = ptrCnt
        print(count)

count = [0, 4, 1, 1, 0, 1, 1, 0, 1, 2, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1]
nums = [6, 2, 3, 9, 9, 1, 8, 1, 5, 1, 25, 1]
countSort(count)
print(count)
```

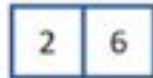
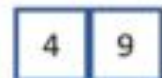
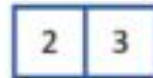
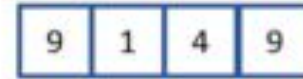
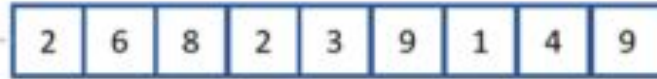
Time complexity of Counting Sort

- Should be used when
 - The range of the input integers is known and relatively small compared to the size of the input array.
 - The input array is relatively small or has few distinct elements.
 - The input array can be modified, or additional space can be allocated for the output array.

Time complexity is: $O(n + k)$ where n is the number of elements in the array and k is the largest number, **why?**

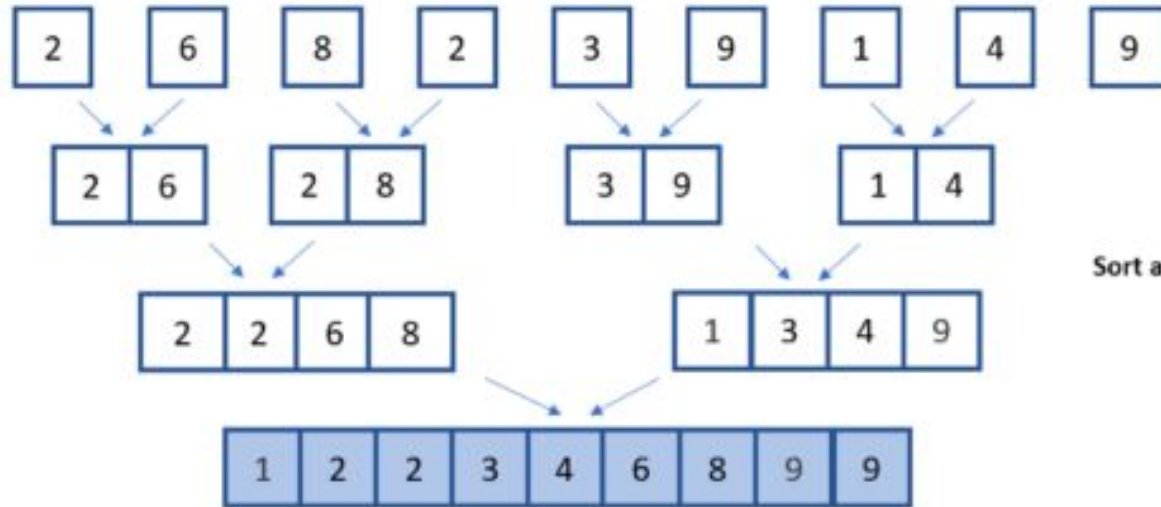
Merge Sort time complexity analysis

Starting array



Recursively divide the array into smaller parts, until each subpart can be divided no further.

Merge continued...



Sort and combine each subpart

Code snippet and what is time complexity?



Time complexity Analysis for Insertion Sort

Steps:

1. Divide the array in half, resulting in two subarrays of size $n/2$.
2. Recursively sort each subarray using merge sort.
3. Merge the sorted subarrays back together in $O(n)$ time.

Each recursive call to merge sort will divide the subarray in half again, resulting in a total of $\log n$ levels of recursion. At each level, the algorithm performs $O(n)$ work to merge the sorted subarrays back together. Therefore, the total time complexity of merge sort is $O(n \log n)$.

The $O(n \log n)$ time complexity of merge sort makes it one of the most efficient sorting algorithms, especially for large input sizes.

Questions

1. Why is Time complexity important?
2. Does a nested loop necessarily, imply n^2 time complexity? **Why?**
3. What is the sorting algorithm in Python? Timsort?
What is the time complexity?

Q/A



**See you at
the next
session!**

