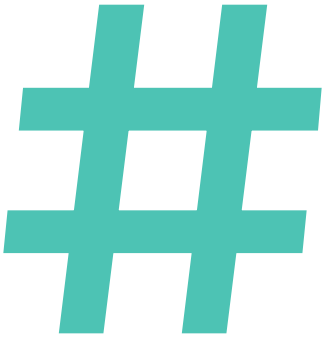


Introduction to Time and Space Complexity

Step 01 What is Algorithm Efficiency

When comparing algorithms, we evaluate their efficiency, which is primarily influenced by time and space.

> **NOTE**
> Time refers to the amount of time an algorithm takes to complete a task.
> Space indicates the amount of memory it consumes to complete a task.

Different computers may run the same algorithm at varying speeds due to differing hardware capabilities. Therefore, it is essential to have a standardized method for measuring time and space that is independent of hardware differences.

We can achieve this by using Big O notation to measure the performance of each algorithm.

Step 02 What is Big O Notation

Big O notation is a mathematical notation used to describe the worst-case performance of an algorithm and express how its running time or space requirements grow with the input size.

It primarily assumes that the algorithm will take the maximum amount of time or resources for any given input.

It is expressed as O(f(n)), where “O” stands for order of magnitude and “f(n)” indicates the algorithm’s growth rate based on input size “n”.

Step 03 Common Big O Notation Classes

Big O Notation provides various expected results that are crucial for understanding algorithm efficiency.

01 O(1) Constant

The algorithm’s growth rate remains constant, regardless of the input size. An example of a constant space complexity is a method that takes a number as input and returns true if it is even.

02 O(log n) Logarithmic

The growth rate increases logarithmically with the input size. An example of time complexity is the Binary search on a sorted array of recursive calls that split the problem in half, reducing the time required.

03 O(n) Linear

The growth rate increases linearly with the input size. If the input size doubles, the growth rate will also roughly double. An example of linear space complexity is a method that creates a new array copy from the input array.

04 O(n log n) Linearithmic

The growth rate increases in proportion to the number of elements multiplied by the logarithm of the number of elements. For example the time complexity of merge sort or quicksort algorithms.

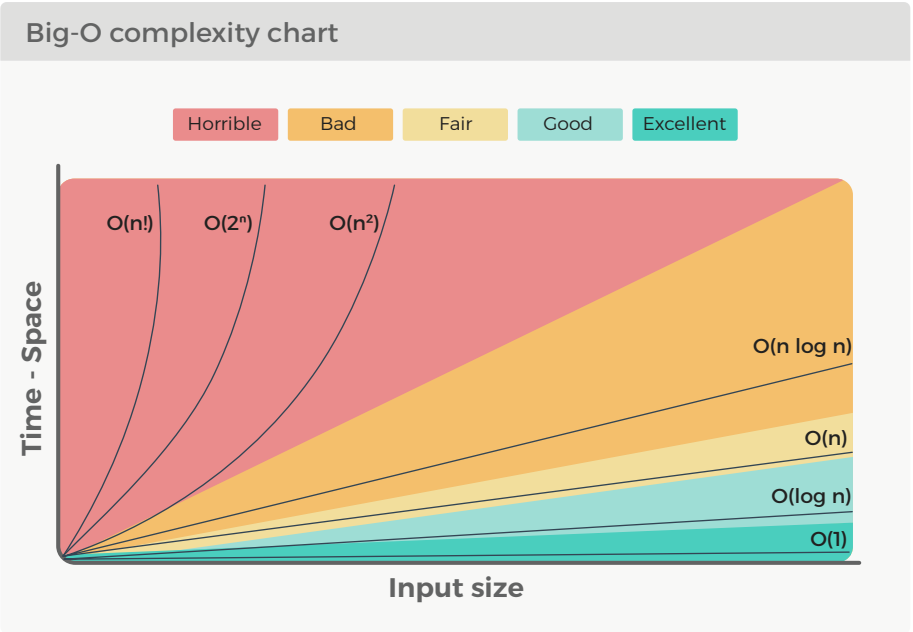
05 O(n^2) Quadratic

The growth rate increases quadratically with the input size. Examples of time complexity include nested loop.

06 O(2^n) Exponential

The growth rate increases exponentially with the input size. This represents algorithms with very slow performance as the input size increases. For example, the time complexity of the Fibonacci function makes two recursive calls for each level of recursion. This means that the number of recursive calls doubles with each increase in the input size.

The image below describes the growth rates of different complexities in Big O nottion



Step 04 Sorting an Array using Different Algorithms

Below we have two sorting algorithms, but they have different time complexity and space complexity.

01 Bubble Sort Implementation

```
01  public static void bubbleSort(int[] arr) {
02      int n = arr.length;
03      for (int i = 0; i < n - 1; i++) {
04          for (int j = 0; j < n - i - 1; j++) {
05              if (arr[j] > arr[j + 1]) {
06                  int temp = arr[j];
07                  arr[j] = arr[j + 1];
08                  arr[j + 1] = temp;
09              }
10          }
11      }
12  }
```

- **Time Complexity**

  In the worst case, when the array is completely reversed, the algorithm requires iterating over the array multiple times, comparing and swapping adjacent elements until the array is sorted.

  As the input size (n) increases, the number of iterations increases quadratically.

  Therefore, the time complexity is O(n^2), where n is the size of the array.

- **Space Complexity**

  The algorithm uses a constant amount of additional space to store variables for the sorting process.

  Regardless of the input size, the space complexity remains O(1), indicating constant space usage.

02 Quick Sort Implementation

```java
public static void quickSort(int[] arr, int low, int high) {
    if (low < high) {
        int pivotIdx = partition(arr, low, high);
        quickSort(arr, low, pivotIdx - 1);
        quickSort(arr, pivotIdx + 1, high);
    }
}

private static int partition(int[] arr, int low, int high) {
    int pivot = arr[high];
    int i = low - 1;
    for (int j = low; j < high; j++) {
        if (arr[j] < pivot) {
            i++;
            int temp = arr[i];
            arr[i] = arr[j];
            arr[j] = temp;
        }
    }
    int temp = arr[i + 1];
    arr[i + 1] = arr[high];
    arr[high] = temp;
    return i + 1;
}
```

- **Time Complexity**

The algorithm selects a pivot element, partitions the array, and recursively sorts the two resulting subarrays.
In the worst case, where the pivot always selects the smallest or largest element, the time complexity becomes O(n^2).

- **Space Complexity**

The algorithm uses a small amount of additional space for the recursive function calls.
In the worst case, the recursion depth can reach O(n).