Table of Content

Understand the Complexity Metrics
× Analyze the Time Complexity
× Evaluate the Best Case, Worst Case, and Average Case
× Consider the Space Complexity
× Identify Bottlenecks
× Compare with Known Complexity Classes
× Consider Trade-offs
× Discuss Limitations
× Provide Recommendations for Improvement

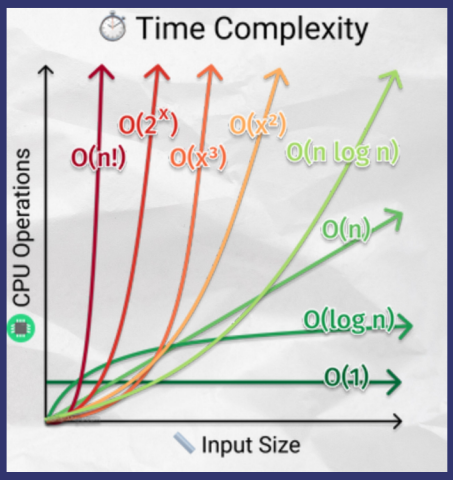
Understand the Complexity Metrics

Complexity metrics are tools used to evaluate the efficiency and performance of an algorithm. They primarily focus on two key aspects:

1. Time Complexity: Measures the time an algorithm takes to complete as a function of the input size.
2. Space Complexity: Measures the amount of memory an algorithm uses relative to the input size.

Analyze the Time Complexity

Time complexity is expressed using Big O notation, which provides an upper bound on the growth rate of the runtime relative to the input size (n). It helps categorize algorithms based on how they scale as the input grows.



Common Time Complexities:

01

O(1) - Constant Time

The runtime is independent of the input size. E.g., accessing an element in an array.

02

O(log n) - Logarithmic Time

The runtime grows logarithmically with input size. E.g., binary search.

03

O(n) - Linear Time

The runtime grows linearly with the input size. E.g., iterating through an array.

04

O(n log n) - Linearithmic Time

Common in efficient sorting algorithms like merge sort and quicksort.

05

$O(n^2)$ - Quadratic Time

The runtime grows quadratically with the input size. E.g., bubble sort, insertion sort.

Purpose of Time Complexity

01

To predict the scalability of an algorithm.

02

To compare the efficiency of different algorithms.

03

To identify potential bottlenecks or inefficiencies in the algorithm.

Evaluate the Best Case, Worst Case, and Average Case

Merge Sort

- The merge function takes two sorted lists (left and right) and merges them into a single sorted list.
- Initialization: An empty list merged is created to store the merged result. Two pointers i and j are used to traverse the left and right lists, respectively.

```
public void sortStudents() {
    if (students == null || students.size() <= 1) {
        return; // No need to sort if the list is null or has only one element
    }
    students = mergeSort(students);
}

private List<Student> mergeSort(List<Student> studentList) {
    if (studentList.size() <= 1) {
        return studentList;
    }

    int mid = studentList.size() / 2;
    List<Student> left = mergeSort(new ArrayList<>(studentList.subList(0, mid)));
    List<Student> right = mergeSort(new ArrayList<>(studentList.subList(mid, studentList.size()

    return merge(left, right);
}

private List<Student> merge(List<Student> left, List<Student> right) {
    List<Student> merged = new ArrayList<>();
    int i = 0, j = 0;
```

Evaluate the Best Case, Worst Case, and Average Case

Merge Sort

The Algorithms I use in my code is Merge Sort, so I will present about The Best Case ,Worst Case and Average Case

```
    while (i < left.size() && j < right.size()) {
        if (left.get(i).getMarks() >= right.get(j).getMarks()) {
            merged.add(left.get(i));
            i++;
        } else {
            merged.add(right.get(j));
            j++;
        }
    }

    while (i < left.size()) {
        merged.add(left.get(i));
        i++;
    }

    while (j < right.size()) {
        merged.add(right.get(j));
        j++;
    }

    return merged;
}
```

Merge Sort

Best Case: O(n log n)

Explanation: Merge Sort always divides the array into halves and merges them, regardless of whether the array is already sorted. The process involves breaking down the array into single elements and then merging them back together in sorted order, requiring O(nlogn)O(n \log n)O(nlogn) operations in all cases.

Worst Case: O(n log n)

Explanation: The worst-case time complexity is also O(nlogn)O(n \log n)O(nlogn) because, similar to the best case, the array is always split and merged regardless of the initial order of elements. Merge Sort has consistent performance across all cases.

Average Case: O(n log n)

Explanation: Since the process of splitting and merging is consistent regardless of the input distribution, the average time complexity remains O(nlogn)O(n \log n)O(nlogn).

Auxiliary Space Complexity: O(n)O(n)O(n)

Explanation: Merge Sort requires additional space to store the subarrays during the merging process. At each level of the recursion, temporary arrays (or lists) are created to hold the left and right halves of the array being sorted. Since the entire array is ultimately copied into these temporary arrays at various stages of the sort, the total auxiliary space required is O(n)O(n)O(n).

Recursion Stack Space: O(logn)O(\log n)O(logn)

Explanation: Merge Sort involves recursive calls that divide the array until the base case of single-element subarrays is reached. The depth of the recursion tree is logn\log nlogn, which corresponds to the number of levels in the recursive calls. Each recursive call consumes stack space, leading to an additional space complexity of O(logn)O(\log n)O(logn) for the recursion stack.