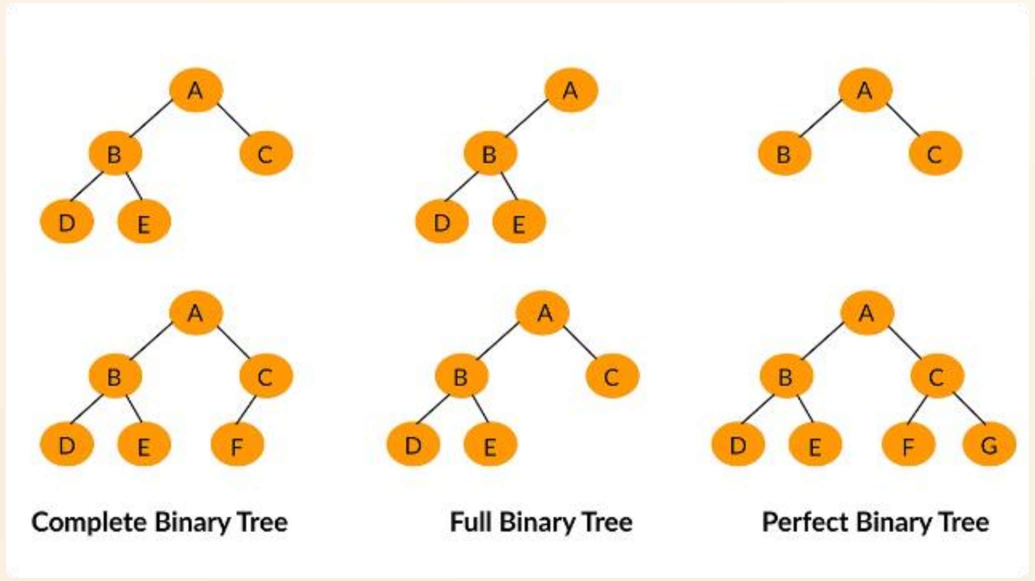
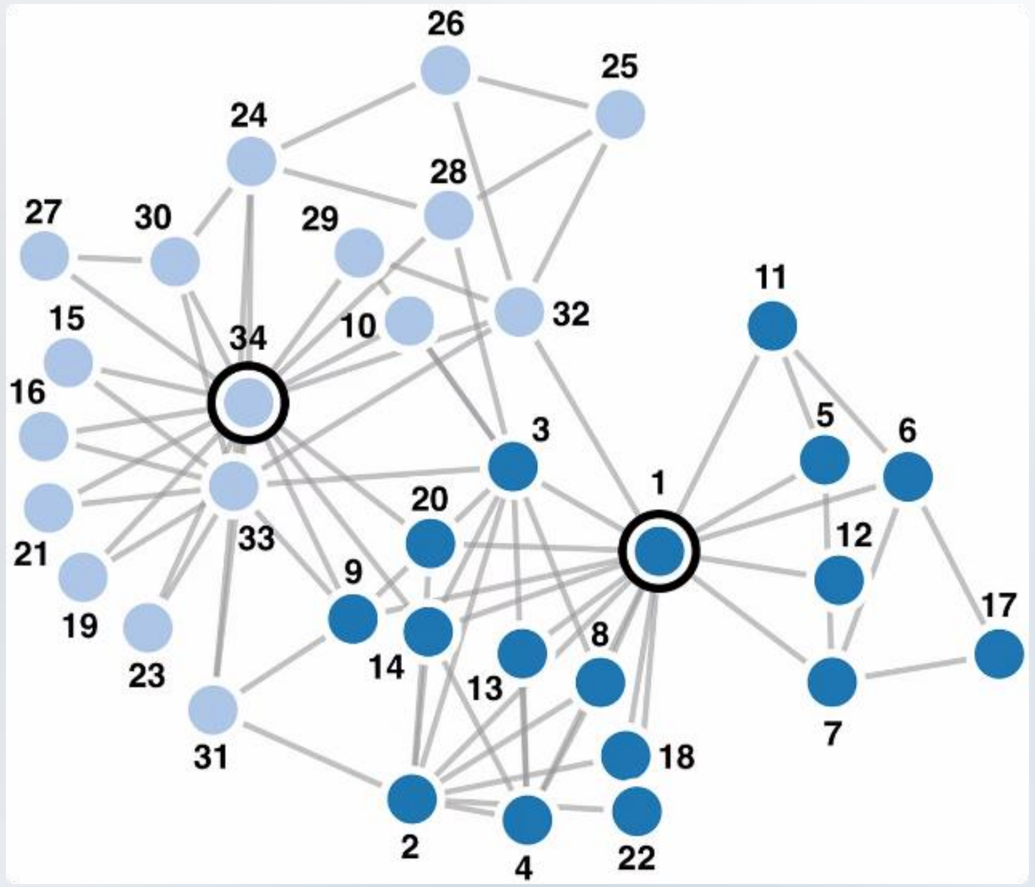
Count of Smaller Numbers After Self

The problem involves determining the count of smaller elements to the right of each element in an integer array. This provides a comprehensive solution, explains the approach in detail, and analyzes the complexity of the implemented algorithm. The chosen method efficiently tackles the problem using a modified merge sort technique, ensuring optimal performance.

V by Vivek kolla





Introduction

1

Understanding the Problem

The goal is to determine the count of smaller numbers after after each element in a given array.

2

Approach

We will explore different algorithms and their time complexities complexities to solve this problem efficiently.

3

Applications

This problem has applications in data analysis, stock trading, and other areas where finding the relative position of elements is important.

Problem Statement

Given an integer array nums , we need to return an integer array counts where counts[i] is the number of smaller elements to the right of nums [i].

Input

An array of integers.

Output

An array containing the count of smaller numbers after each element in the input array.

Example

Input: [5, 2, 6, 1] Output: [2, 1, 1, 0]

Explanation:
- For 5, there are 2 smaller elements to its right (2 and 1).
- For 2, there is 1 smaller element to its right (1).
- For 6, there is 1 smaller element to its right (1).
- For 1, there are no smaller elements to its right.

- **Divide**: Recursively divide the array into two halves until each half contains a single element.
- **Conquer**: During the merge process, count the elements in the right half that are smaller than the elements in the left half.
- **Merge**: Combine the two halves while maintaining the count of smaller elements.

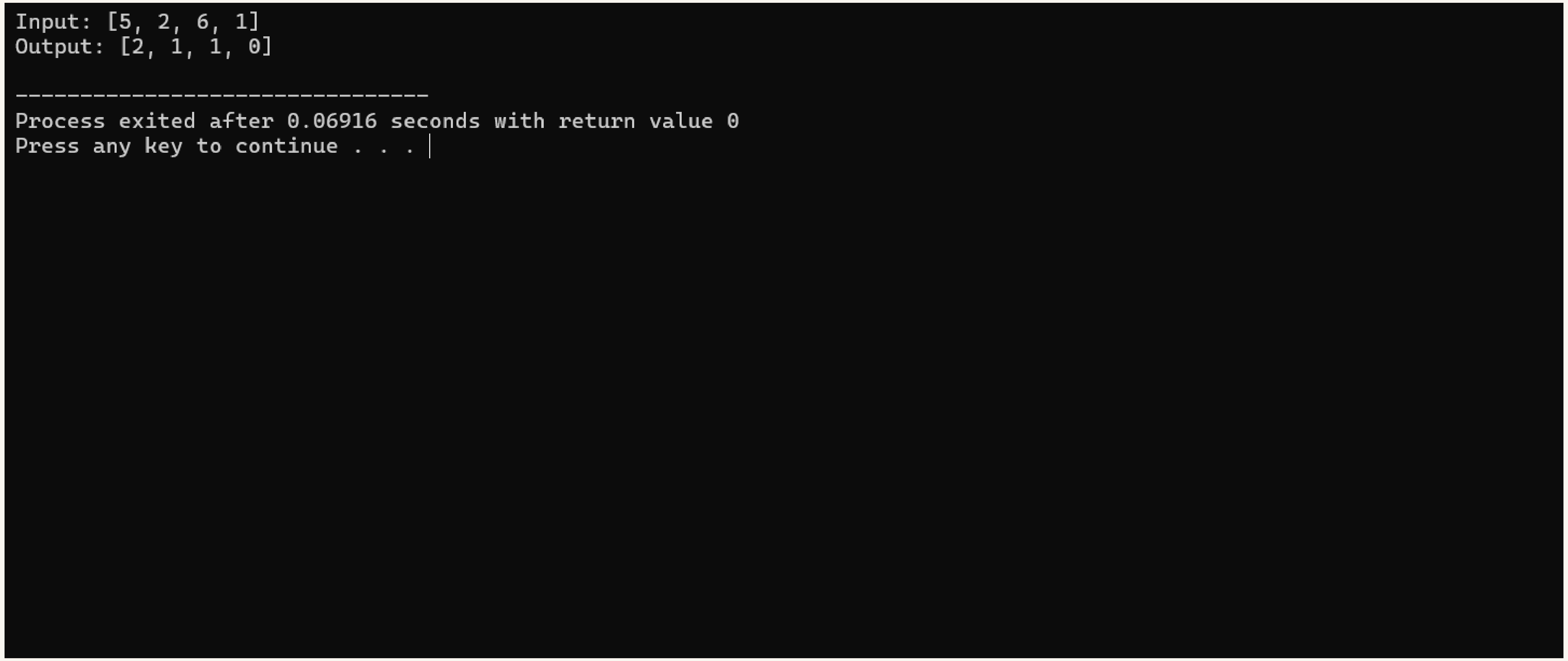
Count and Merge:
- Maintain an array counts to keep track of the count of smaller elements for each index.
- During the merge step, compare elements from both halves and update the counts array accordingly.

CODING

```
#include <stdio.h>
#include <stdlib.h>
void merge(int* nums, int* indices, int* counts, int left, int
mid, int right) {
    int leftSize = mid - left + 1;
    int rightSize = right - mid;
    int* leftIndices = (int*)malloc(leftSize * sizeof(int));
    int* rightIndices = (int*)malloc(rightSize * sizeof(int));

    for (int i = 0; i < leftSize; ++i)
        leftIndices[i] = indices[left + i];
    for (int i = 0; i < rightSize; ++i)
        rightIndices[i] = indices[mid + 1 + i];

    int i = 0, j = 0, k = left, rightCounter = 0;

    while (i < leftSize && j < rightSize) {
        if (nums[leftIndices[i]] <= nums[rightIndices[j]]) {
            counts[leftIndices[i]] += rightCounter;
            indices[k++] = leftIndices[i++];
        } else {
            rightCounter++;
            indices[k++] = rightIndices[j++]; } }
     while (i < leftSize) {
        counts[leftIndices[i]] += rightCounter;
        indices[k++] = leftIndices[i++]; }
```

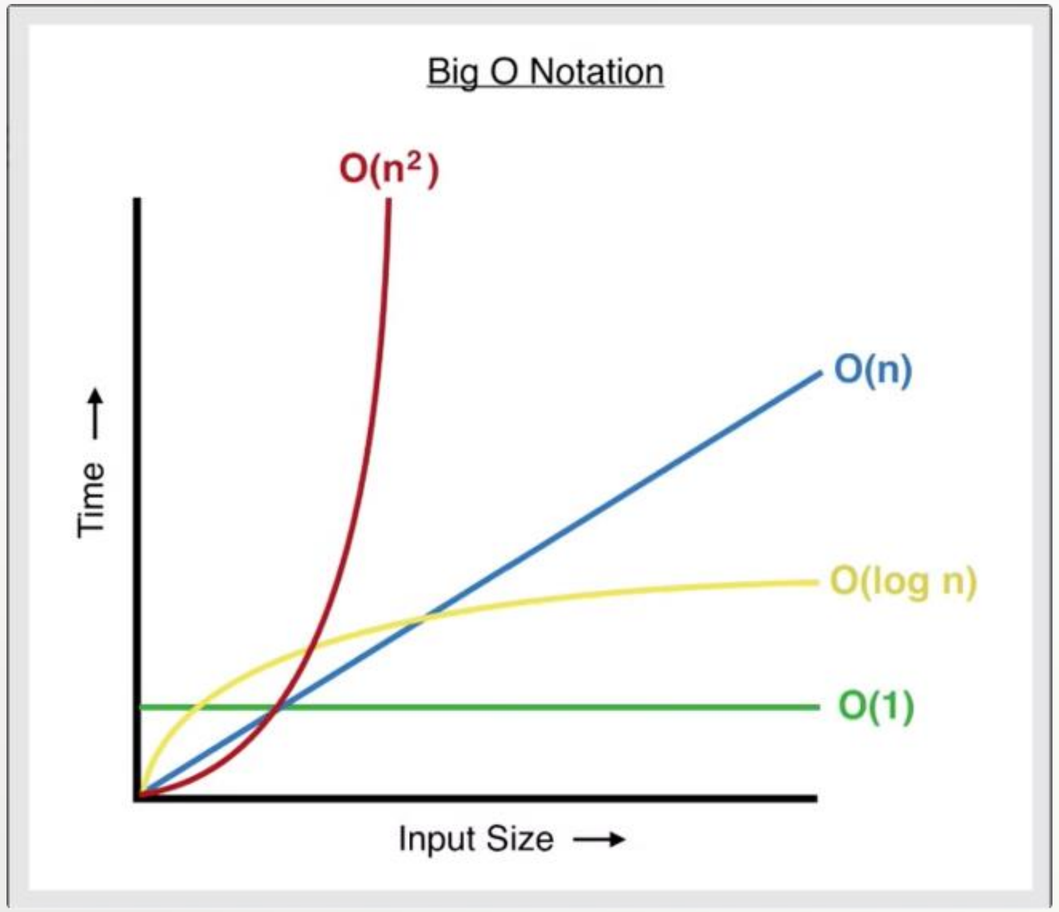
```
 while (i < leftSize) {
        counts[leftIndices[i]] += rightCounter;
        indices[k++] = leftIndices[i++];}
    while (j < rightSize) {
        indices[k++] = rightIndices[j++];}
     free(leftIndices);
    free(rightIndices);}
void mergeSort(int* nums, int* indices, int* counts, int left, int right) {
    if (left < right) {
        int mid = left + (right - left) / 2;
        mergeSort(nums, indices, counts, left, mid);
        mergeSort(nums, indices, counts, mid + 1, right);
        merge(nums, indices, counts, left, mid, right); } }
mergeSort(nums, indices, counts, 0, numsSize - 1);
    free(indices);
    return counts; }
int main() {
    int nums[] = {5, 2, 6, 1};
    int numsSize = sizeof(nums) / sizeof(nums[0]);
    int returnSize;
    int* result = countSmaller(nums, numsSize, &returnSize);
    printf("Output: [");
    for (int i = 0; i < returnSize; ++i) {
        printf("%d", result[i]);
        if (i < returnSize - 1)
            printf(", ");}
    printf("]\n");
    free(result);
    return 0; }
```

OUTPUT



```
Input: [5, 2, 6, 1]
Output: [2, 1, 1, 0]

--------------------------------
Process exited after 0.06916 seconds with return value 0
Press any key to continue . . .
```

- The merge function merges two halves and counts the smaller elements on the right.
- The mergeSort function recursively divides the array and sorts while counting smaller elements
- The countSmaller function initializes the necessary arrays and calls mergeSort.
- The main function demonstrates the use of countSmaller and prints the input and output arrays.



Complexity Analysis

1

Brute Force

A simple approach is to compare each element with all the elements that come after it. This has a time complexity of O(n^2).

2

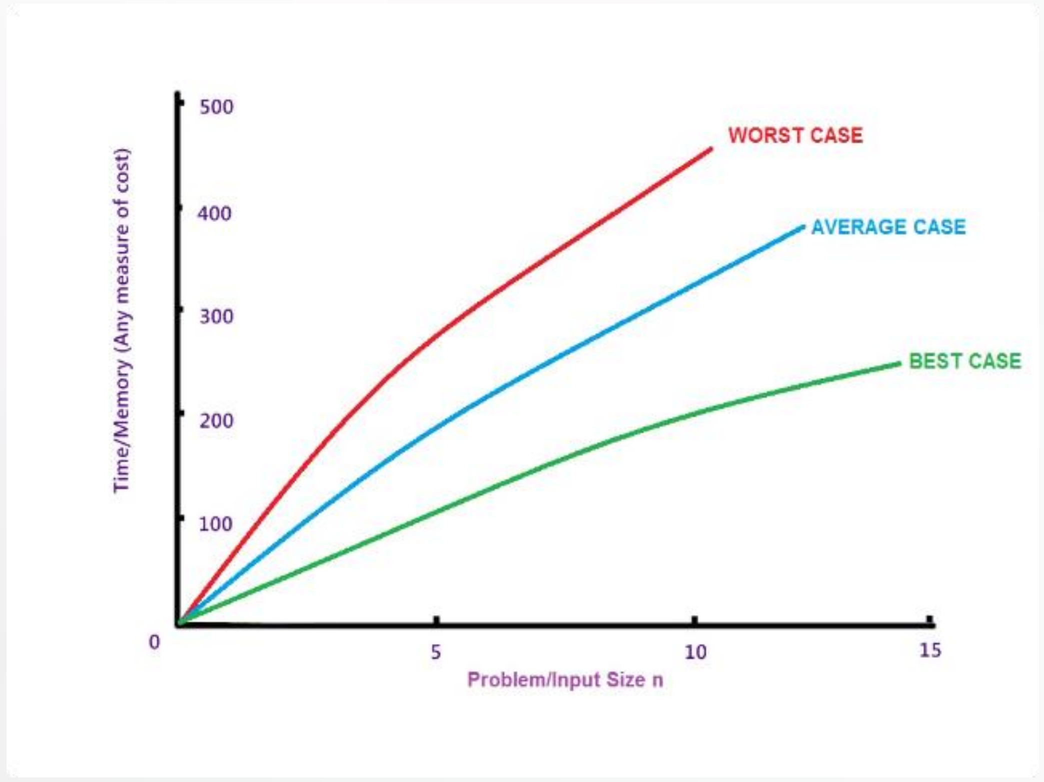
Segment Tree

We can use a Segment Tree data structure to keep track of the smaller elements. This has a time complexity of O(n log n).

3

Binary Indexed Tree

Another efficient approach is to use a Binary Indexed Tree (Fenwick Tree) to count the smaller elements. This also has a time complexity of O(n log n).



Best Case

Optimal Solution

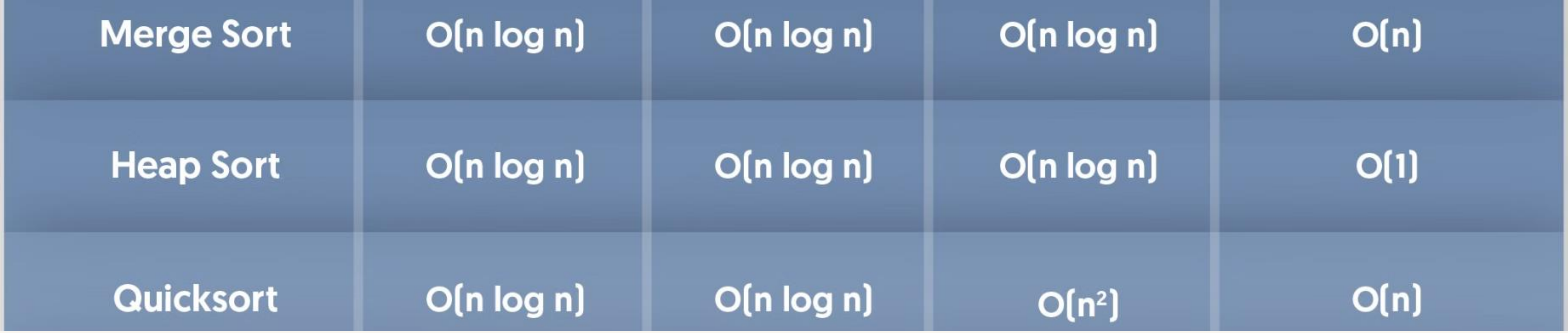
The optimal solution for this problem has a time complexity of O(n log n), which can be achieved using a Segment Tree or Binary Indexed Tree.

Explanation

These data structures allow for efficient range queries and updates, which are crucial for solving this problem.

Space Complexity

The space complexity of these solutions is also O(n), as we need to store the intermediate data structures.



Merge Sort	O(n log n)	O(n log n)	O(n log n)	O(n)
Heap Sort	O(n log n)	O(n log n)	O(n log n)	O(1)
Quicksort	O(n log n)	O(n log n)	$O(n^2)$	O(n)

Worst Case

Input Array

The worst-case input for this problem is an array in descending order, as this will require the most comparisons.

Time Complexity

In the worst case, the brute force approach has a time complexity of O(n^2), which is significantly slower than the optimal O(n log n) solution.

Space Complexity

The space complexity remains the same, O(n), even in the worst case, as we still need to store the intermediate data structures.

Average Case

1

Input Distribution

For the average case, we assume that the input array elements are randomly distributed, without any particular order.

2

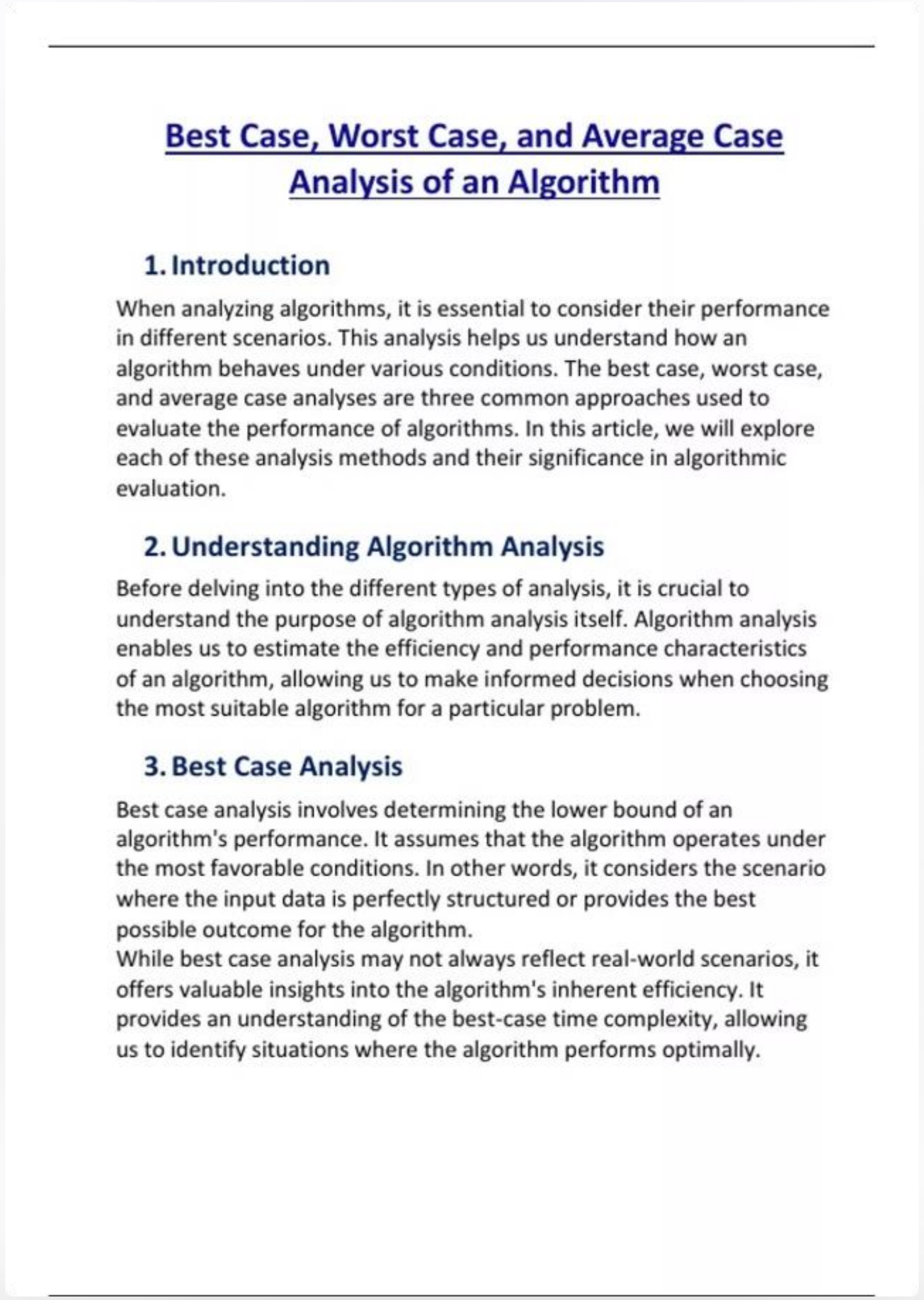
Time Complexity

In the average case, the optimal solutions with O(n log n) time complexity are expected to perform well.

3

Comparison

The brute force approach, with its O(n^2) time complexity, will be significantly slower in the average case compared to the optimal solutions.



Best Case, Worst Case, and Average Case Analysis of an Algorithm

1. Introduction

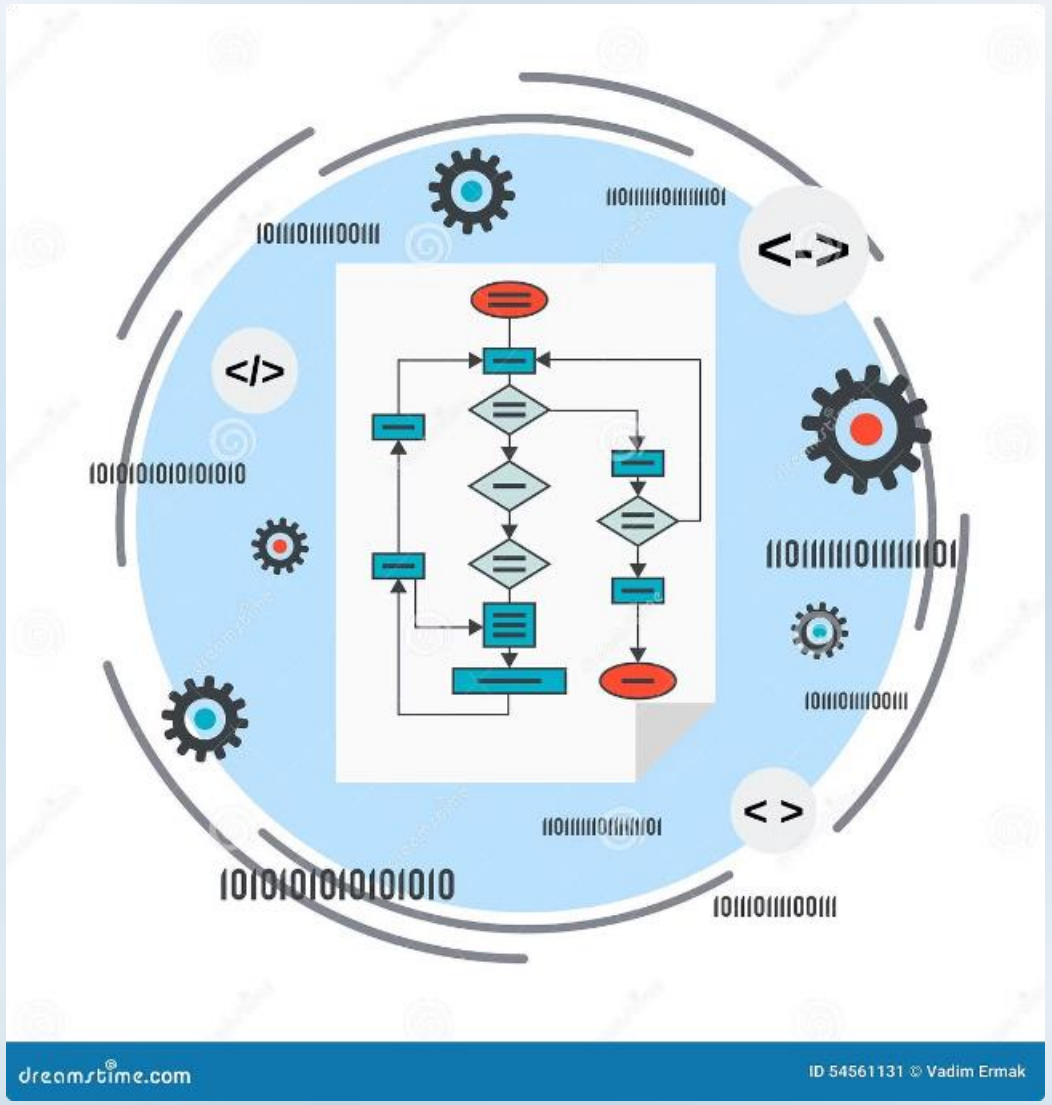
When analyzing algorithms, it is essential to consider their performance in different scenarios. This analysis helps us understand how an algorithm behaves under various conditions. The best case, worst case, and average case analyses are three common approaches used to evaluate the performance of algorithms. In this article, we will explore each of these analysis methods and their significance in algorithmic evaluation.

2. Understanding Algorithm Analysis

Before delving into the different types of analysis, it is crucial to understand the purpose of algorithm analysis itself. Algorithm analysis enables us to estimate the efficiency and performance characteristics of an algorithm, allowing us to make informed decisions when choosing the most suitable algorithm for a particular problem.

3. Best Case Analysis

Best case analysis involves determining the lower bound of an algorithm's performance. It assumes that the algorithm operates under the most favorable conditions. In other words, it considers the scenario where the input data is perfectly structured or provides the best possible outcome for the algorithm.
While best case analysis may not always reflect real-world scenarios, it offers valuable insights into the algorithm's inherent efficiency. It provides an understanding of the best-case time complexity, allowing us to identify situations where the algorithm performs optimally.



Conclusion

1 Efficient Solutions

The optimal solutions for the "Count of Smaller Numbers After Self" problem have a time complexity of O(n log n), achieved using data structures like Segment Trees or Binary Indexed Trees.

2 Tradeoffs

While the optimal solutions are more efficient, they may require more complex implementation and additional memory usage compared to the brute force approach.

3 Real-World Considerations

The choice of algorithm should consider factors such as input size, memory constraints, and the specific requirements of the problem domain.