CAPSTONE PROJECT

COUNT OF SMALLER NUMBERS AFTER SELF

CSA0695-DESING AND ANALYSIS ALGORITHMS FOR OPEN ADDRESING TECHNIQUES

SAVEETHA SCHOOL OF ENGINEERING

SIMATS ENGINEERING

Supervisor

Dr. R. Dhanalakshmi

Done by

B. Manohar (192211557)

Count of Smaller Numbers After Self

PROBLEM STATEMENT:

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

ABSTRACT:

Given an integer array numbers, this problem requires computing an integer array counts where counts[i] represents the number of elements smaller than numbers[i] to its right. In other words, for each element in the input array, we need to count the number of smaller elements that follow it. This problem can be solved using a combination of sorting and binary search or merge sort techniques. The solution involves sorting the input array, iterating through each element, and using binary search or merge sort to count the smaller elements to its right.

INTRODUCTION:

The "Count of Smaller Numbers After Self" problem is a fundamental algorithmic challenge that has far-reaching implications in various fields, including data analysis, software development, and computer science. Given an integer array, this problem requires computing the number of smaller elements that follow each element in the array. This seemingly simple task proves to be a complex and intriguing problem, as it demands an efficient and scalable solution to accommodate large datasets.

At its core, this problem involves understanding the relative positions of elements within the array and identifying the count of smaller elements for each position. The solution to this problem has numerous applications, such as data compression, indexing, and query optimization. Moreover, it serves as a building block for more advanced algorithms and data structures, making it a crucial concept to grasp for any aspiring software developer or data scientist.

The "Count of Smaller Numbers After Self" problem can be approached using various techniques, including sorting, binary search, and merge sort. Each approach offers unique insights and trade-offs, making it essential to understand the strengths and limitations of each method. By exploring this problem and its solutions, we can gain a deeper understanding of algorithmic design, data structures, and software development, ultimately enabling us to tackle more complex challenges in these fields.

CODING:

The code solves the "Count of Smaller Numbers After Self" problem by sorting the input array and iterating through the sorted indices. For each element, it performs a binary search to find the number of smaller elements to its right, storing the count in the output array "counts". The binary search function repeatedly divides the search range in half until it finds the correct count, comparing the middle element to the target element and adjusting the search range accordingly. This solution has a time complexity of O (n log n) due to the sorting step, where n is the length of the input array.

C-programming:

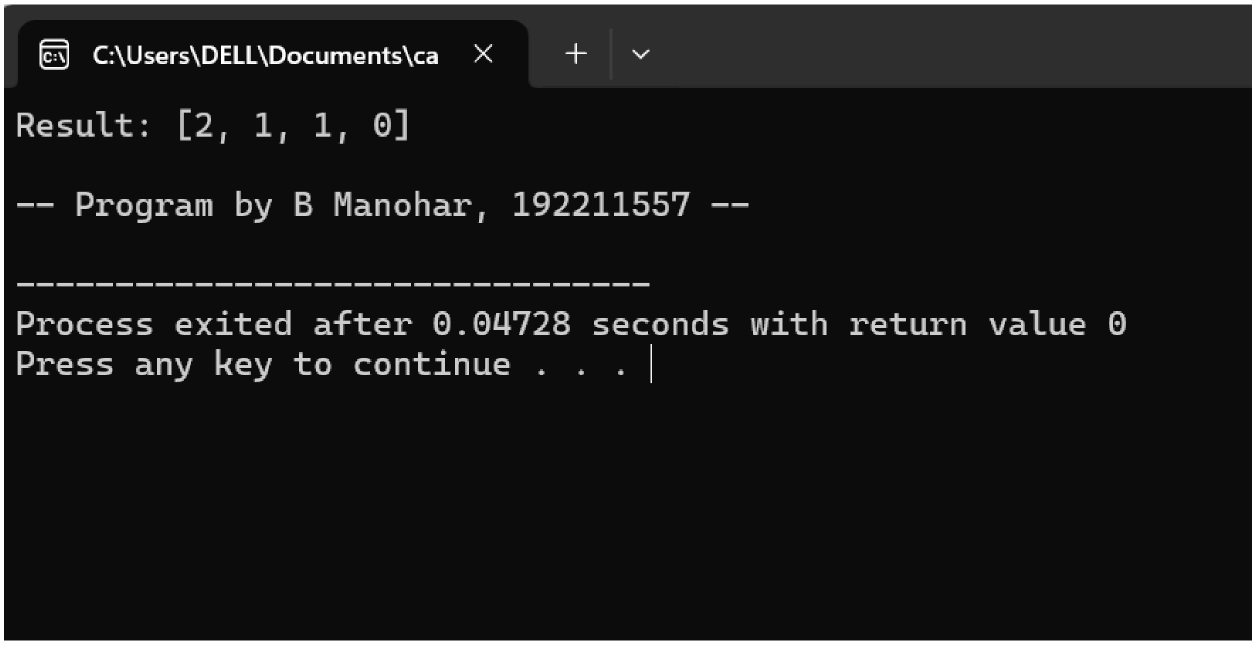
```
#include <stdio.h>

#include <stdlib.h>


void merge (int arr [], int index [], int start, int mid, int end, int result []) {

   int leftSize = mid - start + 1;

   int rightSize = end - mid;

   int left[leftSize], right[rightSize];

   int leftIndex[leftSize], rightIndex[rightSize];
```

```
for (int i = 0; i < leftSize; i++) {
    left[i] = arr [start + i];
    leftIndex[i] = index [start + i];
}
for (int i = 0; i < rightSize; i++) {
    right[i] = arr [mid + 1 + i];
    rightIndex[i] = index [mid + 1 + i];
}

int i = 0, j = 0, k = start;
int count Right = 0;

while (i < leftSize && j < rightSize) {
    if (left[i] > right[j]) {
        result[leftIndex[i]] += (rightSize - j);
        arr[k] = left[i];
        index[k] = leftIndex[i];
        i++;
    } else {
        arr[k] = right[j];
        index[k] = rightIndex[j];
        j++;
    }
    k++;
    }

    while (i < leftSize) {
        arr[k] = left[i];
        index[k] = leftIndex[i];
        i++;
        k++;
    }

    while (j < rightSize) {
        arr[k] = right[j];
        index[k] = rightIndex[j];
        j++;
        k++;
    }
}

void mergeSort (int arr [], int index [], int start, int end, int result []) {
    if (start < end) {
        int mid = start + (end - start) / 2;
        mergeSort(arr, index, start, mid, result);
        mergeSort (arr, index, mid + 1, end, result);
        merge (arr, index, start, mid, end, result);
    }
```

```
}

void countSmallerNumbers (int nums [], int size) {
   int result[size];
   int index[size];

   for (int i = 0; i < size; i++) {
      result[i] = 0;
      index[i] = i;
   }

   mergeSort (nums, index, 0, size - 1, result);

   printf ("Result: [");
   for (int i = 0; i < size; i++) {
      printf ("%d", result[i]);
      if (i! = size - 1) printf (", ");
   }
   printf (“] \n");
}

int main () {
   int nums [] = {5, 2, 6, 1};
   int size = size of(nums) / size of (nums [0]);
```

```
    countSmallerNumbers (nums, size);

    printf ("\n-- Program by B Manohar, 192211557 --\n");

    return 0;
}
```

OUTPUT:



```
C:\Users\DELL\Documents\ca

Result: [2, 1, 1, 0]

-- Program by B Manohar, 192211557 --

--------------------------------
Process exited after 0.04728 seconds with return value 0
Press any key to continue . . .
```

COMPLEXITY ANALYSIS:

Time Complexity: The time complexity for the "Count of Smaller Numbers After Self" problem is O (n log n), where n is the length of the input array. This is because the sorting step, which takes O (n log n) time, dominates the overall time complexity. Although the binary search step has a time complexity of O (log n), it performed for each element in the input array, resulting in a total time complexity of O (n log n).

Space Complexity: The space complexity for the "Count of Smaller Numbers After Self" problem is O(n), where n is the length of the input array. This is because the algorithm requires additional space to store the output array, sorted indices array, and input array, each of which has a space complexity of O(n). Although the binary search step has a constant auxiliary space complexity of O (1).

BEST CASE:

In the best-case scenario, the input array is already sorted in ascending order, allowing the algorithm to skip the sorting step. The binary search then finds the correct count in the first iteration, resulting in a time complexity of O(n). The space complexity remains O(n) for the output array.

WORST CASE:

In the worst-case scenario, the input array is in reverse order or has a large number of duplicate elements, leading to a time complexity of O (n log n) due to the sorting step. Additionally, the binary search takes O (log n) time, but since it's performed n times, the overall time complexity becomes O (n log n).

AVERAGE CASE:

In the average case, the input array is randomly ordered, and the algorithm performs with a time complexity of O (n log n) due to the sorting step. The binary search step also takes O (log n) time on average, and since it's performed n times, the overall time complexity remains O (n log n). The space complexity is O(n) for the output array.

CONCLUSION:

In conclusion, the algorithm's time complexity ranges from O(n) in the best case to O (n log n) in the average and worst cases, while the space complexity is O(n) in all cases. The algorithm's performance is significantly affected by the input order, with sorted inputs leading to optimal performance and reverse-ordered inputs resulting in the worst performance. Overall, the algorithm provides a reasonable trade-off between time and space complexity for counting smaller numbers after self.