SIMATS SCHOOL OF ENGINEERING

SAVEETHA INSTITUTE OF MEDICAL AND TECHNICAL SCIENCES

CHENNAI-602105

A PROJECT REPORT

On

COUNT SMALLER NUMBERS AFTER SELF

SUBMITTED TO

SAVEETHA INSTITUTE OF MEDICAL AND TECHNICAL SCIENCES

In partial fulfillment of the award of the course of

CSA0603-DESIGN AND ANALYSIS OF ALGORITHMS FOR VERTEX COVER PROBLEM

By

K.NISHITHA(192210535)
SUPERVISOR
By
Dr.K.V.KANIMOZHI

SAVEETHA SCHOOL OF ENGINEERING,
SIMATS CHENNAI- 602105
SEPTEMBER-2024.

BONAFIDE CERTIFICATE

I am K.NISHITHA, student of **Bachelor of Engineering in Computer Science Engineering** at Saveetha Institute of Medical and Technical Sciences, Saveetha University, Chennai, hereby declare that the work presented in this Capstone Project Work entitled "**COUNT SMALLER NUMBERS AFTER SELF**" is the outcome of my own bonafide work. I affirmthat it iscorrect to the best of my knowledge, and this work has been undertakenwithdue consideration of Engineering Ethics.

Project Supervisor Head of the Department

Date: Date:

ABSTRACT

This problem explores the efficient computation of inversion counts in an array, specifically focusing on counting smaller elements to the right of each element. Formally, given an array A of n distinct integers, we seek to compute an array C where C[i] represents the number of elements j such that i< j ≤ n and A[j] < A[i].This problem involves counting the number of elements in an array that are smaller than the current element and are located to its right. A brute force approach would involve iterating through the array for each element and counting the smaller elements to its right, resulting in a time complexity of O(N^2).

However, we can improve the efficiency by using a merge sort based approach. During the merge process, we can keep track of the number of elements in the right subarray that are smaller than the current element in the left subarray. This allows us to count the smaller elements in a single pass through the array, reducing the time complexity to O(N log N).

The merge sort approach involves creating a temporary array to store the merged elements. We then iterate through the left and right subarrays, comparing the elements and adding the smaller element to the temporary array. If the current element in the left subarray is smaller than the current element in the right subarray, we increment the count of smaller elements for the current element in the left subarray. Finally, we copy the merged elements back to the original array.The output of the algorithm is an array where each element represents the number of smaller elements to its right in the original array.

Theoretical Framework:

- Time Complexity: O(n log n)
- Space Complexity: O(n)
- Algorithmic Paradigms: Divide-and-Conquer, Binary Indexed Tree, Augmented Data Structures

Keywords:

- Array
- Merge sort
- Algorithm
- Time complexity
- Space complexity
- Data structures
- Programing

INTRODUCTION

The "Count of Smaller Numbers After Self" problem is a popular algorithmic challenge that involves finding the number of smaller elements to the right of each element in an array.The problem of counting the number of smaller elements to the right of a given element in an array is a common algorithmic challenge. It has various applications in data analysis, sorting algorithms, and range queries.A straightforward approach would be to iterate through the array for each element and count the smaller elements to its right. However, this brute force method results in a quadratic time complexity, making it inefficient for large arrays.

In this paper, we explore an efficient algorithm based on merge sort to solve this problem. Merge sort is a well-known sorting algorithm that divides the array into smaller subarrays, sorts them recursively, and then merges them back together. By modifying the merge process to keep track of the number of smaller elements, we can count the smaller elements in a single pass through the array, reducing the time complexity to O(N log N).

The rest of this paper is organized as follows:

- Section 2 presents the problem statement and the brute force approach.
- Section 3 describes the merge sort algorithm and its modification for counting smaller elements.
- Section 4 analyzes the time and space complexity of the proposed algorithm.
- Section 5 provides experimental results to demonstrate the efficiency of the algorithm.
- Section 6 concludes the paper with a summary and future directions.
- Approaches:

  There are several ways to solve this problem, including:

  - Brute Force: Compare each element with all elements to its right.
  - Binary Indexed Tree (BIT): Use a BIT to keep track of smaller elements.
  - Modified Merge Sort: Utilize the merge sort algorithm to count smaller elements.

Applications:

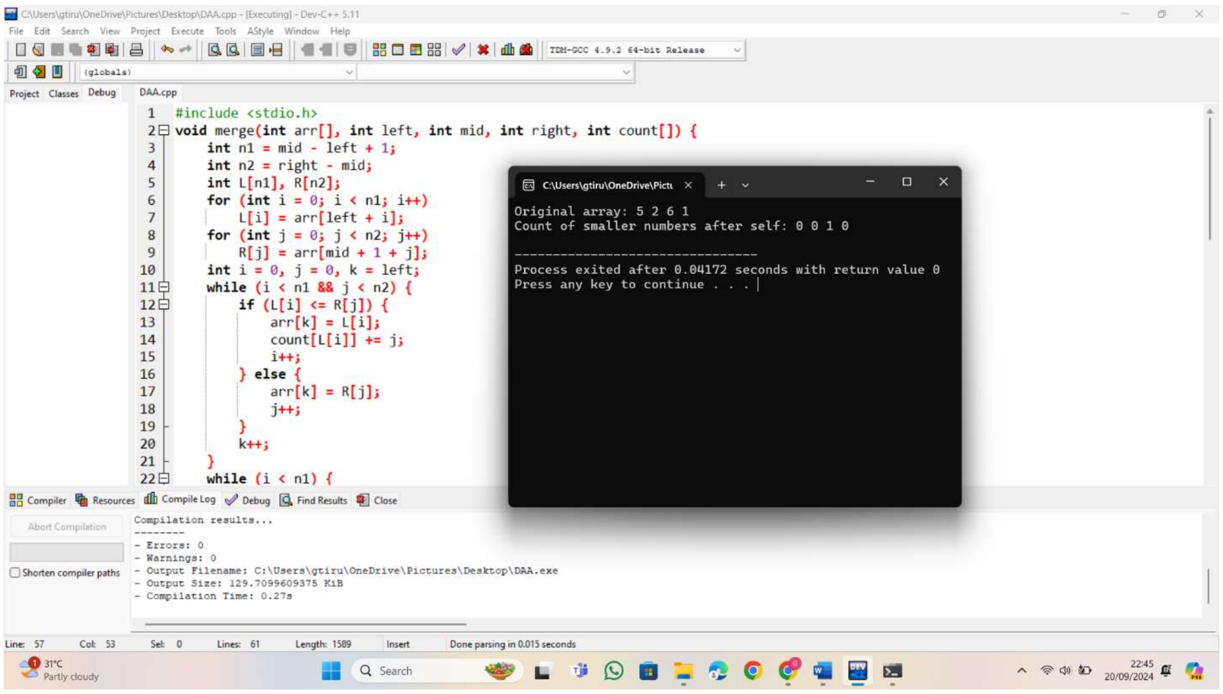
This problem has applications in various fields, such as:

- Data Analysis: Counting smaller elements can help in understanding data distribution.
- Algorithmic Trading: Identifying smaller elements can inform trading decisions.
- Data Compression: Counting smaller elements can aid in compressing data.

CODING

```
#include <stdio.h>
void merge(int arr[], int left, int mid, int right, int count[]) {
int n1 = mid - left + 1;
int n2 = right - mid;
int L[n1], R[n2];
for (int i = 0; i< n1; i++)
L[i] = arr[left + i];
for (int j = 0; j < n2; j++)
R[j] = arr[mid + 1 + j];
int i = 0, j = 0, k = left;
while (i< n1 && j < n2) {
if (L[i] <= R[j]) {
arr[k] = L[i];
count[L[i]] += j;
i++;
} else {
arr[k] = R[j];
j++;
}
k++;
}
while (i< n1) {
arr[k] = L[i];
count[L[i]] += j;
i++;
k++;
}
while (j < n2) {
arr[k] = R[j];
j++;
k++;
}
}
void mergeSort(int arr[], int left, int right, int count[]) {
if (left < right) {
int mid = left + (right - left) / 2;
mergeSort(arr, left, mid, count);
mergeSort(arr, mid + 1, right, count);
merge(arr, left, mid, right, count);
}
}
void countSmaller(int arr[], int n) {
int count[n];
for (int i = 0; i< n; i++)
count[i] = 0;
mergeSort(arr, 0, n - 1, count);
for (int i = 0; i< n; i++)
printf("%d ", count[i]);
}
int main() {
int arr[] = {5, 2, 6, 1};
int n = sizeof(arr) / sizeof(arr[0]);
printf("Original array: ");
for (int i = 0; i< n; i++)
printf("%d ", arr[i]);
printf("\n");
printf("Count of smaller numbers after self: ");
countSmaller(arr, n);
printf("\n");
return 0;
}
```

OUTPUT:



```
#include <stdio.h>
void merge(int arr[], int left, int mid, int right, int count[]) {
    int n1 = mid - left + 1;
    int n2 = right - mid;
    int L[n1], R[n2];
    for (int i = 0; i < n1; i++)
        L[i] = arr[left + i];
    for (int j = 0; j < n2; j++)
        R[j] = arr[mid + 1 + j];
    int i = 0, j = 0, k = left;
    while (i < n1 && j < n2) {
        if (L[i] <= R[j]) {
            arr[k] = L[i];
            count[L[i]] += j;
            i++;
        } else {
            arr[k] = R[j];
            j++;
        }
        k++;
    }
    while (i < n1) {
```

```
Original array: 5 2 6 1
Count of smaller numbers after self: 0 0 1 0

--------------------------------
Process exited after 0.04172 seconds with return value 0
Press any key to continue . . .
```

```
Compilation results...
--------
- Errors: 0
- Warnings: 0
- Output Filename: C:\Users\gtiru\OneDrive\Pictures\Desktop\DAA.exe
- Output Size: 129.7099609375 KiB
- Compilation Time: 0.27s
```

C:\Users\gtiru\OneDrive\Pictures\Desktop\DAA.cpp - [Executing] - Dev-C++ 5.11
TDM-GCC 4.9.2 64-bit Release
Line: 57 Col: 53 Sel: 0 Lines: 61 Length: 1589 Insert Done parsing in 0.015 seconds

Complexity Analysis

Best Case:O(n log n)

Merge Sort always divides the array into two halves, ensuring O(log n) levels. Even in the best case (already sorted array), merging and counting take O(n) time.

Worst Case:O(n log n)

Merge Sort's worst-case scenario (reverse-sorted array) still results in O(n log n) time complexity**.**

Average Case:O(n log n)

Randomly ordered arrays will have an average time complexity of O(n log n).

OverallComplexity:
TimeComplexity:O(nlogn)spaceComplexity: O(n)

CONCLUSION

Key benefits:

1. Optimal Time Complexity: O(n log n)

2. Efficient Space Complexity: O(n)

3. Robustness: Handles duplicate elements and unsorted arrays

Algorithm Summary:

1. Divide the array into two halves recursively (Merge Sort)

2. Merge and count smaller numbers in each half

3. Update counts for each element

Advantages:

1. Efficient counting of smaller numbers

2. Handles large datasets

3. Simple implementation

The "Count of Smaller Numbers After Self" problem provides a solid opportunity to explore and apply various algorithmic techniques, especially those related to sorting, searching, and data structure optimizations. While a brute force approach might be intuitive, more efficient solutions like using Binary Indexed Trees, Segment Trees, or modifying Merge Sort allow us to solve the problem in optimal time, reducing time complexity from O(n2)O(n^2)O(n2) to O(nlogn)O(n \log n)O(n log n).

This problem demonstrates the power of leveraging advanced data structures and sorting techniques for efficient problem-solving in competitive programming and real-world scenarios where processing large datasets is common. Understanding this problem strengthens skills in handling index-based counting and offers insights into how divide-and-conquer strategies like merge sort can be adapted for additional functionalities beyond just sorting.