CAPSTONE PROJECT REPORT

Reg NO: 192210103

Name: K.Vivek Reddy

Course Code: CSA0656

Course Name: DAA

SLOT: A

ABSTRACT:

The problem involves determining the count of smaller elements to the right of each element in an integer array. This document 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.

INTRODUCTION:

Given an integer array `nums`, the objective is to return an integer array `counts` where `counts[i]` represents the number of smaller elements to the right of `nums[i]`. This problem is a common interview question and serves as a good exercise for understanding divide-and-conquer algorithms. While a naive approach would involve nested loops resulting in O(n^2) time complexity, more efficient algorithms like the modified merge sort can achieve O(n log n) time complexity.

Problem Statement:

Given an integer array `nums`, return an integer array `counts` where `counts[i]` is the number of smaller elements to the right of `nums[i]`. The challenge is to achieve this in an efficient manner, ideally with a time complexity better than O(n^2).

Example

For the array:

nums = [5, 2, 6, 1]

1. Understanding the Output:

- For each element in `nums`, determine how many of the elements to its right are smaller than it.

- Example:

  - 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.

2. Initial Approach (Inefficient):

- A naive solution involves iterating over each element and counting the smaller elements to its right, resulting in O(n^2) time complexity.

- This approach is simple but inefficient for large arrays.

3. Efficient Approach Using Modified Merge Sort:

- Instead of the naive O(n^2) method, we can use a modified merge sort to count the smaller elements efficiently.

- This approach leverages the divide-and-conquer strategy of merge sort to maintain the sorted order and count smaller elements during the merge process

4. Merge Sort Technique:

- 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.

5. Detailed Steps:

- Partition: Divide the array into left and right halves recursively.

- Count and Merge:

- Maintain an array `smaller` to keep track of the count of smaller elements for each index.

- During the merge step, compare elements from both halves and update the `smaller` array accordingly

6. Determine the Counts:

- After the merge sort completes, the `smaller` array contains the count of smaller elements to the right for each element in the original array

CODING:

```
#include <stdio.h>
#include <stdlib.h>

// Function to merge two halves and count the smaller elements
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 (j < rightSize)


{
        indices[k++] = rightIndices[j++];
    }

    free(leftIndices);
    free(rightIndices);
}

// Recursive function to perform merge sort and count smaller elements
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);
    }
}
```

```
// Function to count smaller numbers after self
int* countSmaller(int* nums, int numsSize, int* returnSize)

{
    *returnSize = numsSize;
    int* counts = (int*)calloc(numsSize, sizeof(int));
    int* indices = (int*)malloc(numsSize * sizeof(int));

    for (int i = 0; i < numsSize; ++i)
        indices[i] = i;

    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: [2, 1, 1, 0]

--------------------------------
Process exited after 0.06865 seconds with return value 0
Press any key to continue . . .
```

COMPLEXITY ANALYSIS:

Time Complexity:

1. **Merge Sort Based Counting:**
   - The algorithm leverages a modified merge sort to count the number of smaller elements to the right of each element.
   - Merge sort operates by recursively dividing the array into halves, sorting, and merging them.
2. **Recursive Division:**
   - The array is divided into two halves recursively until each half contains a single element.
   - The depth of recursion is determined by the logarithm of the array size, leading to O(log n) levels of recursion.
3. **Merging and Counting:**
   - During the merge step, the algorithm merges the two halves while counting the smaller elements to the right.
   - Each merge operation processes the entire array in O(n) time for each level of recursion.
4. **Overall Time Complexity:**
   - Combining the recursive division and the merge steps, the overall time complexity is O(n log n).
   - This is significantly more efficient than a naive O(n^2) approach, making it suitable for larger arrays.

Space Complexity:

1. **Auxiliary Space:**
   - The algorithm requires additional space for auxiliary arrays to store indices and temporary values during the merge process.
   - These auxiliary arrays are proportional to the size of the input array.
2. **Overall Space Complexity:**
   - The space complexity is O(n), where n is the size of the input array.
   - This is due to the extra storage required for the auxiliary arrays used in the merge step.

BEST CASE:

Scenario:

- The array is sorted in descending order.

Example:

- Input: nums = [6, 5, 4, 3, 2, 1]
- Output: [5, 4, 3, 2, 1, 0]

Explanation:

- For 6, there are 5 smaller elements to the right (5, 4, 3, 2, 1).
- For 5, there are 4 smaller elements to the right (4, 3, 2, 1).
- For 4, there are 3 smaller elements to the right (3, 2, 1).
- For 3, there are 2 smaller elements to the right (2, 1).
- For 2, there is 1 smaller element to the right (1).
- For 1, there are no smaller elements to the right.

Complexity:

- The algorithm will still perform the merge sort operations.

- Time Complexity: O(n log n)

Reason:

- Despite the order of elements, the merge sort will recursively divide the array and merge it back, ensuring the same performance as any other case.

WORST CASE:

Scenario:

- The array is sorted in ascending order.

Example:

- Input: nums = [1, 2, 3, 4, 5, 6]
- Output: [0, 0, 0, 0, 0, 0]

Explanation:

- For 1, there are no smaller elements to the right.
- For 2, there are no smaller elements to the right.
- For 3, there are no smaller elements to the right.
- For 4, there are no smaller elements to the right.
- For 5, there are no smaller elements to the right.
- For 6, there are no smaller elements to the right.

Complexity:

- The algorithm needs to count each element's smaller elements on the right.

- Time Complexity: O(n log n)

Reason:

- The merge sort approach ensures that even in the worst case, the time complexity remains efficient compared to a naive O(n^2) approach.

AVERAGE CASE:

Scenario:

- The array elements are in random order.

Example:

- Input: nums = [5, 2, 6, 1]
- Output: [2, 1, 1, 0]

Explanation:

- For 5, there are 2 smaller elements to the right (2, 1).
- For 2, there is 1 smaller element to the right (1).
- For 6, there is 1 smaller element to the right (1).
- For 1, there are no smaller elements to the right.

Complexity:

- The performance remains consistent with the merge sort characteristics.

- Time Complexity: O(n log n)

Reason:

- Randomly ordered elements do not affect the divide-and-conquer strategy of merge sort, ensuring consistent performance.

CONCLUSION:

The problem of counting smaller numbers after self can be effectively solved using a modified merge sort algorithm. This approach provides a balance between simplicity and efficiency, ensuring an O(n log n) complexity for all cases. The provided implementation in Python demonstrates the solution, and the analysis confirms its effectiveness across different scenarios. The modified merge sort is a powerful technique for this class of problems, offering significant performance improvements over naive methods.