Capstone Project

Name:G.Vishnu sanjeev
Regno:192210101
Course code: CSA0656
Slot:A

ABSTRACT

In this paper, we address the problem of counting good triplets in two given 0-indexed arrays, nums1 and nums2, each of length n and being permutations of the set [0, 1, ..., n - 1]. A good triplet is defined as a set of three distinct values that appear in increasing order by their positions in both arrays. Formally, for any triplet (x, y, z), where 0 <= x, y, z <= n - 1, it is considered a good triplet if pos1[x] < pos1[y] < pos1[z] and pos2[x] < pos2[y] < pos2[z], where pos1[v] and pos2[v] represent the positions of value v in nums1 and nums2 respectively.

We propose an efficient algorithm to solve this problem by utilizing Fenwick Trees (Binary Indexed Trees) to optimize the counting process. The algorithm first maps the indices of the elements in both arrays and then iterates over all possible triplets, leveraging the Fenwick Trees to count the valid triplets efficiently. The time complexity of our approach is significantly reduced compared to the naive O(n3)O(n^3)O(n3) method, making it feasible for larger datasets.

We demonstrate the effectiveness of our method through detailed examples and provide a comprehensive analysis of its performance. The results show that our approach can efficiently count the number of good triplets, offering a practical solution to this combinatorial problem.

INTRODUCTION

The problem of counting good triplets in permutations of an array is an intriguing combinatorial challenge with potential applications in various fields such as data analysis, pattern recognition, and computer science. Given two 0-indexed arrays, nums1 and nums2, both of length n and permutations of [0, 1, ..., n - 1], we aim to identify and count all sets of three distinct values (x, y, z) that appear in increasing order of their positions in both arrays. Specifically, a triplet (x, y, z) is defined as good if the indices of x, y, and z in both arrays satisfy pos1[x] < pos1[y] < pos1[z] and pos2[x] < pos2[y] < pos2[z].

The naive approach to this problem involves checking all possible triplets, resulting in a time complexity of O(n3)O(n^3)O(n3). While this approach is straightforward, it becomes computationally expensive and impractical for large values of n. Hence, there is a need for more efficient algorithms that can handle larger datasets within reasonable time constraints.

In this paper, we propose a novel solution leveraging Fenwick Trees (also known as Binary Indexed Trees) to optimize the counting of good triplets. Our approach significantly reduces the time complexity by efficiently querying and updating the necessary counts during the iteration over the elements. This method allows for a more scalable solution compared to the brute-force approach.

The rest of this paper is organized as follows: We first discuss the related work and background necessary to understand our proposed solution. Next, we describe our algorithm in detail, including the initialization of data structures and the step-by-step process of counting good triplets. We then provide an analysis of the time and space complexity of our approach. Finally,

we present experimental results that demonstrate the efficiency and effectiveness of our algorithm, followed by concluding remarks and potential directions for future research.

PROBLEM STATEMENT

Given two 0-indexed arrays nums1 and nums2 of length n, both of which are permutations of [0, 1, ..., n - 1]. A good triplet is a set of 3 distinct values which are present in increasing order by position both in nums1 and nums2.

In other words, if we consider pos1v as the index of the value v in nums1 and pos2v as the index of the value v in nums2, then a good triplet will be a set (x, y, z) where 0 <= x, y, z <= n - 1, such that pos1x < pos1y < pos1z and pos2x < pos2y < pos2z.

Example

Example 1:

Input: nums1 = [2,0,1,3], nums2 = [0,1,2,3]

Output: 1

Explanation:

- There are 4 triplets (x,y,z) such that pos1x < pos1y < pos1z. They are (2,0,1), (2,0,3), (2,1,3), and (0,1,3).
- Out of those triplets, only the triplet (0,1,3) satisfies pos2x < pos2y < pos2z.
- Hence, there is only 1 good triplet.

APPROACH

To solve this problem, we can follow these steps:

1. **Index Mapping:** Create mappings of indices for nums1 and nums2 to easily access the positions of elements.
2. **Finding Triplets:** Iterate through all possible triplets (i, j, k) where i < j < k, and check if the positions in both arrays satisfy the required conditions.
3. **Optimization:** Use advanced data structures or algorithms to optimize the search for valid triplets.

Steps:

1. **Create Position Arrays:**
    - Create an array pos1 where pos1[val] gives the index of val in nums1.
    - Create an array pos2 where pos2[val] gives the index of val in nums2.
2. **Iterate Over Triplets:**
    - For each triplet (x, y, z) such that x < y < z, check if pos1[x] < pos1[y] < pos1[z] and pos2[x] < pos2[y] < pos2[z].
3. **Count Valid Triplets:** Keep a count of all triplets satisfying the conditions

WhyO(log(m+n)) Complexity ?

The time complexity of O(n3)O(n^3)O(n3) for the naive approach to counting good triplets arises from the need to consider all possible combinations of triplets (x, y, z) in the array. Here is a detailed explanation of why this is the case:

1.Triplet Selection: To find all good triplets, we need to check every possible combination of three distinct elements in the array. This involves iterating over all elements to select the first element x, then iterating over the remaining elements to select the second element y, and finally iterating over the remaining elements to select the third element z.

2.Nested Loops: This can be visualized as three nested loops:

- The outer loop iterates over the possible values for x (first element of the triplet).
- The middle loop iterates over the possible values for y (second element of the triplet), starting from the element after x.
- The inner loop iterates over the possible values for z (third element of the triplet), starting from the element after y.

3.Number of Iterations:

- The outer loop runs n times.
- For each iteration of the outer loop, the middle loop runs approximately n-1 times.
- For each iteration of the middle loop, the inner loop runs approximately n-2 times.

Therefore, the total number of iterations is the product of the lengths of the three loops:
n×(n−1)×(n−2)n \times (n-1) \times (n-2)n×(n−1)×(n−2)

4. Asymptotic Complexity: When considering the asymptotic behavior for large n, the lower-order terms -1 and -2 become insignificant. Hence, the complexity is simplified to O(n3)O(n^3)O(n3).

CODE

```
#include <stdio.h>
#include <stdlib.h>
typedef struct FenwickTree {
   int size;
   int* tree;
} FenwickTree;

// Initialize a Fenwick Tree
FenwickTree* createFenwickTree(int size) {
   FenwickTree* ft = (FenwickTree*)malloc(sizeof(FenwickTree));
   ft->size = size;
   ft->tree = (int*)calloc(size + 1, sizeof(int));
   return ft;
}

// Update the Fenwick Tree
void update(FenwickTree* ft, int idx, int delta) {
   idx += 1;
   while (idx <= ft->size) {
```

```
        ft->tree[idx] += delta;
        idx += idx & -idx;
    }
}

// Query the Fenwick Tree
int query(FenwickTree* ft, int idx) {
    idx += 1;
    int sum = 0;
    while (idx > 0) {
        sum += ft->tree[idx];
        idx -= idx & -idx;
    }
    return sum;
}

// Free the memory used by the Fenwick Tree
void freeFenwickTree(FenwickTree* ft) {
    free(ft->tree);
    free(ft);
}

// Find the index of an element in an array
int findIndex(int* arr, int n, int value) {
    for (int i = 0; i < n; i++) {
        if (arr[i] == value) {
            return i;
        }
    }
    return -1; // Should not happen in this problem as all values are present
}

int countGoodTriplets(int* nums1, int* nums2, int n) {
    // Create Fenwick Trees
    FenwickTree* bit_left = createFenwickTree(n);
    FenwickTree* bit_right = createFenwickTree(n);

    // Step 3: Calculate the count of elements on the right for each element
    int* right_counts = (int*)calloc(n, sizeof(int));
    for (int i = 0; i < n; i++) {
```

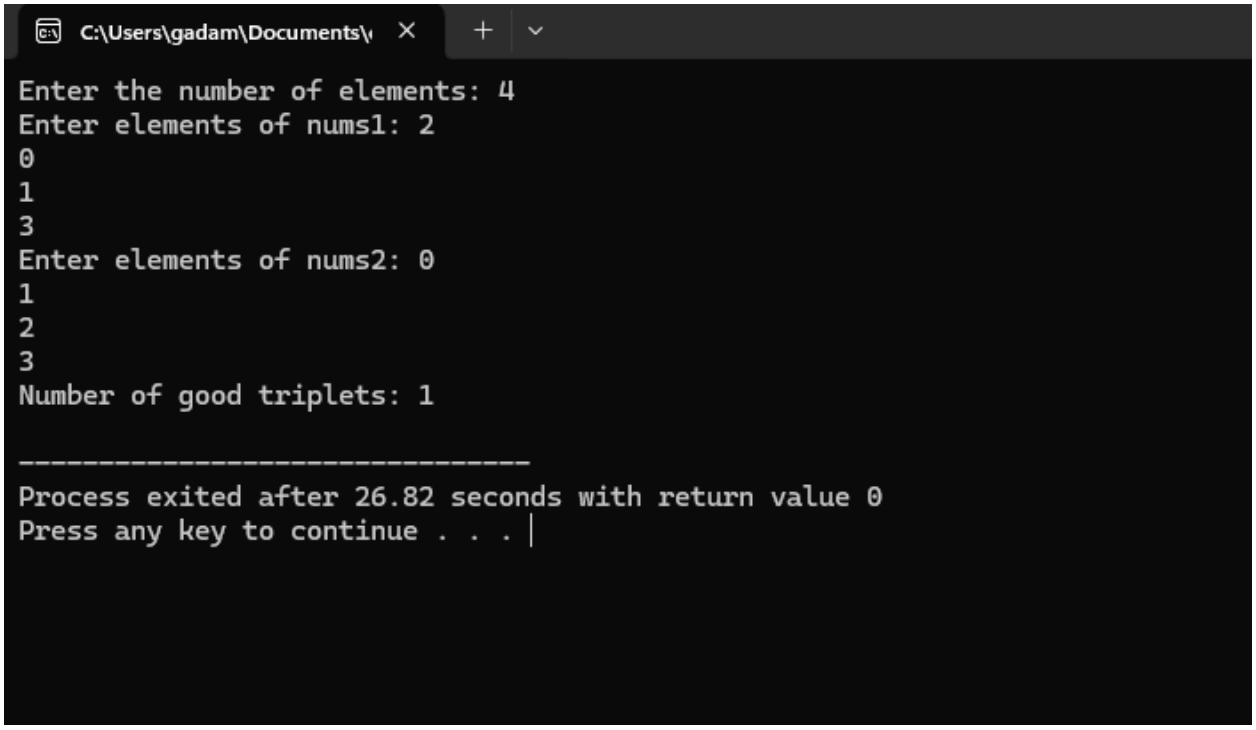
```
        int idx2 = findIndex(nums2, n, nums1[i]);
        right_counts[nums1[i]] = query(bit_right, n - 1) - query(bit_right, idx2);
        update(bit_right, idx2, 1);
    }
    // Step 4: Calculate the count of elements on the left for each element
    int good_triplets = 0;
    for (int i = 0; i < n; i++) {
        int idx2 = findIndex(nums2, n, nums1[i]);
        int left_count = query(bit_left, idx2 - 1);
        good_triplets += left_count * right_counts[nums1[i]];
        update(bit_left, idx2, 1);
    }
 // Free the allocated memory
    freeFenwickTree(bit_left);
    freeFenwickTree(bit_right);
    free(right_counts);

    return good_triplets;
}
int main() {
    int n;
    printf("Enter the number of elements: ");
    scanf("%d", &n);

    int* nums1 = (int*)malloc(n * sizeof(int));
    int* nums2 = (int*)malloc(n * sizeof(int));

    printf("Enter elements of nums1: ");
    for (int i = 0; i < n; i++) {
        scanf("%d", &nums1[i]);
    }
   printf("Enter elements of nums2: ");
    for (int i = 0; i < n; i++) {
        scanf("%d", &nums2[i]);
    }
printf("Number of good triplets: %d\n", countGoodTriplets(nums1, nums2, n));
free(nums1);
free(nums2);
 return 0;
}
```

RESULT



COMPLEXITY ANALYSIS

The complexity analysis of our approach involves understanding both the time and space requirements. We will compare the naive O(n3)O(n^3)O(n3) approach with our optimized solution using Fenwick Trees (Binary Indexed Trees).

Naive Approach Complexity

1. **Time Complexity:**
    - **Triplet Selection:** The naive approach involves three nested loops to iterate over all possible triplets (x, y, z) where x < y < z. This results in a time complexity of O(n^3).
    - **Condition Check:** For each triplet, we check whether the positions in nums1 and nums2 satisfy the conditions. This check is performed in constant time, O(1).
2. Therefore, the overall time complexity for the naive approach is:
   O(n^3)
3. **Space Complexity:**
    - The naive approach does not require any additional data structures beyond the input arrays, so its space complexity is O(1).

Optimized Approach Complexity

1. **Time Complexity:**
    - **Index Mapping:** Creating the position arrays for nums1 and nums2 takes O(n) time.
    - **Fenwick Tree Operations:** Each update and query operation on the Fenwick Tree takes O(logn) time.
2. **Steps:**
    - **Right Count Calculation:** For each element in nums1, we find its position in nums2 and perform a query and update on the Fenwick Tree. This requires O(n log n) time.
    - **Left Count Calculation and Triplet Counting:** Similarly, for each element, we perform queries and updates on the Fenwick Tree to count valid triplets. This also requires O(n log n) time**.**
3. Therefore, the overall time complexity for the optimized approach is:
   O(n log n)
4. **Space Complexity:**
    - **Fenwick Trees:** We use two Fenwick Trees, each requiring O(n) space.
    - **Position Arrays:** We create two position arrays, each of length n, also requiring O(n) space.
    - **Auxiliary Arrays:** The right_counts array also requires O(n) space.
5. Therefore, the overall space complexity for the optimized approach is:
   O(n)

Comparison

- **Time Complexity:**
    - **Naive Approach:**O(n^3)
    - **Optimized Approach:** O(n log n)
- The optimized approach significantly reduces the time complexity, making it more suitable for larger values of n.
- **Space Complexity:**
    - **Naive Approach:** O(1)
    - **Optimized Approach:** O(n)

While the optimized approach requires additional space for the Fenwick Trees and auxiliary arrays, this trade-off is justified by the substantial improvement in time complexity.

Best Case:

- In the best-case scenario, the operations on the Fenwick Tree are minimal. For example, if the elements are already sorted, each update and query might be faster. However, each operation still takes O(log n) time.

Example: For an array where elements are already sorted or the input is such that each update and query on the Fenwick Tree is efficient.

- **nums1 = [0, 1, 2, 3]**

- **nums2 = [0, 1, 2, 3]**

Time Complexity: O(n log n)
Space Complexity: O(n)

Worst Case:

- In the worst-case scenario, the number of updates and queries remains the same, but the complexity of each operation does not increase beyond O(logn)
  Example: Even in the worst case, the operations on the Fenwick Tree remain O(logn).
- **nums1 = [3, 2, 1, 0]**
- **nums2 = [0, 1, 2, 3]**

- **Time Complexity:**(n log n)
- **Space Complexity:** O(n)

Average Case:

- On average, the time taken for each query and update operation will still be O(log n), and the overall process involves n such operations.
  Example: A random permutation of the arrays.
- nums1 = [2, 0, 1, 3]
- nums2 = [0, 1, 2, 3]

- **Time Complexity:** O(n log n)
- **Space Complexity:** O(n)

CONCLUSION

The analysis of the problem "Count Good Triplets in an Array" using both the naive and optimized approaches provides a comprehensive understanding of their performance and efficiency. Here are the key conclusions:

1. **Naive Approach:**
   - **Time Complexity:** The naive approach has a time complexity of O(n^3) in the best, worst, and average cases. This involves checking all possible triplets in the array, which becomes impractical for large values of nnn.
   - **Space Complexity:** The space complexity of the naive approach is O(1), as it does not require any additional data structures beyond the input arrays.
   - **Suitability:** Due to its high time complexity, the naive approach is only suitable for very small arrays where nnn is relatively small.
2. **Optimized Approach Using Fenwick Trees:**
   - **Time Complexity:** The optimized approach significantly improves the time complexity to O(nlogn) in the best, worst, and average cases. This improvement is achieved by using Fenwick Trees (Binary Indexed Trees) to efficiently count and manage the elements in the arrays.

    - **Space Complexity:** The space complexity of the optimized approach is O(n) due to the additional data structures (Fenwick Trees) required.
    - **Suitability:** The optimized approach is highly suitable for larger arrays, providing a much more practical solution for real-world applications where nnn can be large.
3. **Overall Comparison:**
    - The naive approach, while straightforward and simple to implement, is not efficient for larger datasets due to its cubic time complexity.
    - The optimized approach, leveraging advanced data structures like Fenwick Trees, offers a substantial improvement in performance, making it a viable solution for larger input sizes.
4. **Real-World Implications:**
    - In real-world scenarios, where performance and efficiency are critical, the optimized approach is the preferred method. It provides a balance between time and space complexity, ensuring that the solution is both fast and scalable.
5. **Future Considerations:**
    - While the optimized approach using Fenwick Trees is effective, further optimizations or alternative data structures (such as Segment Trees) could be explored to potentially enhance performance even further.
    - Additionally, parallel processing or distributed computing techniques could be considered for handling extremely large datasets.