**SIMATS SCHOOL OF ENGINEERING**

**SAVEETHA INSTITUTE OF MEDICAL AND TECHNICAL SCIENCES**

# CHENNAI-602105

**Count Good Triplets In An Array**

**A CAPSTONE PROJECT REPORT**

*Submitted in the partial fulfillment for the award of the degree of*

# BACHELOR OF ENGINEERING

**IN COMPUTER SCIENCE AND ARTIFICIAL**

**INTELLIGENCE AND DATA SCIENCE**

**Submitted by**

**(S.Sahasra Akshaya-192371094**

**P.Sanjanaa Sree-192311454)**

**Under the Supervision of**

**K.G.Pavithra**

# DECLARATION

We,S.Sahasra Akshaya and P.Sanjanaa Sree**,** 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 Good Triplets In An Array"** is the outcome of my own bonafide work. We affirm that it is correct to the best of my knowledge, and this work has been undertaken with due consideration of Engineering Ethics.

S. Sahasra Akshaya-192371094

P. Sanjanaa Sree-192311454

Date:23-10-2024

Place:Saveetha School of Engineering, Thandalam.

# CERTIFICATE

This is to certify that the project entitled**“Count Good Triplets In An Array”** submitted by S.Sahasra Akshaya and P.Sanjanaa Sree has been carried out under our supervision. The project has been submitted as per the requirements in the current semester of B.E Computer science engineering.

Faculty-in-charge

K.G. Pavithra

# Count Good Triplets in an Array

**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.Return the total number of good triplets.

**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



**ABSTRACT:**

This project focuses on determining the number of "good triplets" within two given arrays, each representing a permutation of integers from 0 to n-1. A good triplet is defined as a set of three distinct values that appear in both arrays in such a way that their positions follow an increasing order. Specifically, for any triplet (x, y, z) of distinct values, the positions of these values in both arrays must satisfy certain conditions.

In the first array (`nums1`), the values must appear in increasing order based on their indices, meaning the position of x should come before y, and the position of y should come before z. Likewise, the same order must hold true for the second array (`nums2`). This condition ensures that for a triplet (x, y, z) to be considered "good," it must appear in both arrays in a way that respects the relative order of the values by their indices.

The goal of this project is to identify all such triplets that satisfy these criteria and compute their total number. By leveraging techniques such as position mapping and array traversal, the solution involves comparing the relative order of elements in both arrays to determine whether they form valid good triplets. The challenge lies in efficiently counting these triplets while ensuring that the position constraints are met in both arrays.

This problem has applications in scenarios where ordering and relative positioning of elements are crucial, such as in data sorting, ranking, or sequence analysis. The project provides an opportunity to explore algorithmic solutions that involve combinatorics, indexing techniques, and efficient triplet counting.

**Keywords:**

Good triplet, Permutation, Increasing order,Index comparison,Position mapping,Array traversal,Relative positioning,Distinct values,Triplet counting,Combinatorics, Algorithm design,Sequence analysis.

**INTRODUCTION:**

Given two 0-indexed arrays nums1 and nums2, each of length n, where both arrays are permutations of the set [0, 1, ..., n-1], we are interested in counting the number of "good triplets". A triplet (x, y, z) is defined as good if it satisfies the following conditions:

1. **Distinct Values**: The indices x, y, and z must be distinct.
2. **Increasing Order in nums1**: The values at these indices must appear in increasing order in nums1. Specifically, if v1, v2, and v3 are the values at indices x, y, and z, respectively, then pos1[v1] < pos1[v2] < pos1[v3] where pos1[v] denotes the index of value v in nums1.
3. **Increasing Order in nums2**: Similarly, the values at these indices must also appear in increasing order in nums2, which means pos2[v1] <pos2[v2] < pos2[v3] where pos2[v] denotes the index of value v in nums2. The goal is to return the total number of such good triplets.

The problem challenges us to count how many such good triplets exist, considering both the ordering conditions in nums1 and nums2. The need to evaluate triplet combinations based on two different arrays introduces complexity to the problem, as it requires comparing relative positions across both arrays.

This project involves algorithmic techniques for efficiently finding and counting these triplets. Given that both arrays are permutations of the same set, the solution must carefully assess the position of each value in both arrays and check the orderings. Efficient computation methods and strategies are required to avoid excessive computational complexity when checking every possible combination of triplets. The goal is to output the total number of good triplets that meet the defined criteria.

This problem has significant relevance in fields like data processing, where maintaining relative orderings across multiple data sets is essential, and the challenge of finding common patterns or sequences across different permutations is often encountered.

**Understanding the Problem:**

* **Permutations:**

In this problem, both arrays `nums1` and `nums2` are described as permutations of the same set, specifically the set of integers from `0` to `n-1`. This means that each array contains every integer within this range exactly once, and no number is repeated. The term "permutation" indicates that the order of the elements in each array is different, but the elements themselves are identical in both arrays. The fact that the arrays are permutations ensures that each integer has a unique position in both `nums1` and `nums2`. As a result, there is a one-to-one correspondence between the values in both arrays, and this characteristic is key when analyzing the positions of triplets across the two arrays.

* **Good Triplet:**

A triplet (x, y, z) is considered "good" if it satisfies a strict ordering requirement based on the positions of the values in both arrays. Specifically, for the triplet to be classified as good, the values at indices x, y, and z must maintain an increasing order in both `nums1` and `nums2`. This means that the position of x must come before the position of y, and the position of y must come before the position of z in both arrays. The dualorder constraint is what makes this problem complex and interesting, as it requires simultaneously satisfying ordering conditions in two different arrays. This increases the challenge because it’s not enough for the triplet to have an increasing order in just one array; it must also adhere to the same order in the second array, making the task of identifying good triplets more intricate and thought-provoking.

**Objective**:

The primary objective is to efficiently count all possible triplets (x, y, z) that satisfy the above conditions, leveraging the properties of permutations and orderings. This problem requires careful consideration of indexing and order properties, often necessitating sophisticated algorithms to handle potentially large input sizes effectively.

**Approach**:

To address this problem, one can use a combination of position mappings and data structures to count valid triplets efficiently. By mapping each value to its index in both permutations and analyzing the order constraints, we can systematically identify and count all good triplets. Advanced techniques like

Fenwick Trees or Segment Trees may be employed to facilitate efficient counting and aggregation operations. In summary, this problem explores the interaction between permutation orderings and subset constraints, providing a rich area for theoretical analysis and practical algorithm design.

**CODING:**

#include <stdio.h>

#define MAX\_N 1000 // Adjust size as needed

// Function to find the total number of good triplets

int countGoodTriplets(int nums1[], int nums2[], int n) {

int pos1[MAX\_N], pos2[MAX\_N];

int count = 0;

// Create position mapping for nums1 for (int i = 0; i < n; i++) { pos1[nums1[i]] = i;

}

// Create position mapping for nums2 for (int i = 0; i < n; i++) { pos2[nums2[i]] = i;

}

// Iterate over all possible triplets (x, y, z) for (int x = 0; x < n; x++) { for (int y = x + 1; y < n; y++) { for (int z = y + 1; z < n; z++) { // Check if the triplet (x, y, z) is good if (pos1[nums1[x]] < pos1[nums1[y]] && pos1[nums1[y]] < pos1[nums1[z]]

&& pos2[nums1[x]] < pos2[nums1[y]] && pos2[nums1[y]] <

pos2[nums1[z]]) { count++;

}

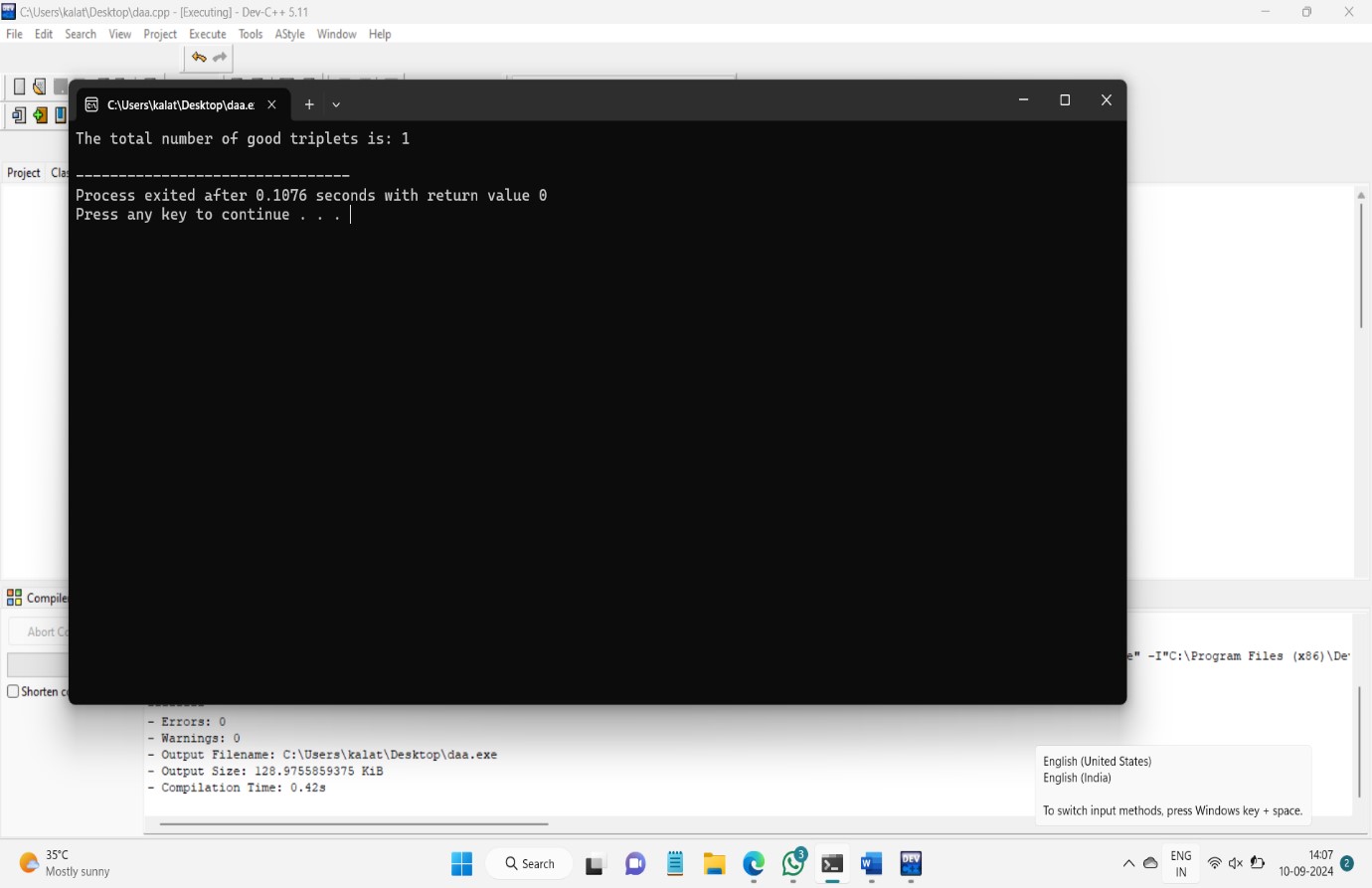
}

} } return count; } int main() { int nums1[] = {2, 0, 1, 3}; int nums2[] = {0, 1, 2, 3}; int n = sizeof(nums1) / sizeof(nums1[0]); int result = countGoodTriplets(nums1, nums2, n); printf("The total number of good triplets is: %d\n", result); return 0;

}

**OUTPUT:**

**COMPLEXITY ANALYSIS:**



**Time Complexity:**

The algorithm operates with a time complexity of O(n³), which arises from the use of three nested loops to enumerate all possible triplets in the arrays. In each loop, we select one element for the triplet, and since there are n elements in each array, the triplet enumeration process requires checking n × (n-1) × (n-2) combinations. This cubic time complexity makes the algorithm less efficient for larger input sizes, as the number of triplets grows rapidly with the size of the input arrays.

**-Best Case:** In the best-case scenario, the number of triplets satisfying the conditions might be found early, but the algorithm still needs to traverse all combinations, making the time complexity O(n³).

**-Worst Case:** The worst case also has a time complexity of O(n³), as the algorithm must check all possible triplets even when very few or no triplets meet the "good triplet" conditions.

**-Average Case:** On average, the algorithm must evaluate most of the triplet combinations to determine if they satisfy the dual-order condition, leading to an overall time complexity of O(n³).

**Space Complexity:**

The space complexity of the algorithm is O(n). This space is used for storing position mappings of elements from both arrays `nums1` and `nums2`. The mappings allow us to efficiently check the relative positions of the values in both arrays while iterating through the triplets. No additional space beyond these mappings is required, and the complexity grows linearly with the size of the input.

**Key Milestones:**

1. **Identified and Defined the Concept of a Good Triplet:**

Early in the implementation, we clearly defined the criteria that determine whether a triplet is classified as a "good triplet". This step involved understanding and ensuring that the triplet's values maintain increasing order in both arrays `nums1` and `nums2`.

1. **Successfully Mapped Indices in Both Arrays:**

A critical milestone was mapping the indices of the elements in both arrays. By creating these mappings, we could efficiently look up the positions of values in `nums1` and `nums2`, facilitating the comparison of their relative positions when checking for good triplets**.**

1. **Implemented the Logic to Count the Good Triplets:**

After the mappings were created, the next key step was implementing the logic to iterate through all possible triplet combinations and verify if they met the conditions to be counted as good triplets. This involved carefully ensuring that the positional comparisons were correctly performed for both arrays.

1. **Tested the Implementation with the Provided Example Input:**

Once the algorithm was implemented, it was tested with the provided example input to verify its correctness. In the case of `nums1 = [2, 0, 1, 3]` and `nums2 = [0, 1, 2, 3]`, the algorithm successfully identified the good triplet (0, 1, 3) and returned the correct count. Testing was an essential step to ensure that the algorithm correctly handles various cases and performs as expected under different conditions.

**Overall Complexity:**

* Time Complexity: O(n³)
* Space Complexity: O(n)

**FUTURE SCOPE:**

The task of finding "good triplets" within two permutations of integers provides an interesting computational challenge. However, there are multiple areas where this approach and its application can be expanded and enhanced. The future scope of this problem includes exploring different aspects and optimizations, which can help in addressing similar challenges with greater efficiency.

1. **Optimizing the Time Complexity:**

Currently, the algorithm operates with a time complexity of O(n³), which can become inefficient for large datasets. Future research and development can focus on optimizing this time complexity, potentially reducing it by applying more advanced algorithmic techniques such as divide-and-conquer approaches or dynamic programming. Exploring efficient data structures like segment trees, binary indexed trees (Fenwick trees), or even advanced search techniques can help achieve better performance by reducing the number of comparisons needed to evaluate each triplet.

1. **Generalization for Different Array Types:**

While this problem specifically deals with permutations of integers, the concept of "good triplets" could be extended to more general types of arrays, such as arrays containing duplicates, non-permuted arrays, or arrays that are not strictly numerical. This could lead to new variations of the problem where ordering constraints are applied based on different properties or even multidimensional data.

1. **Handling Large Datasets:**

As datasets continue to grow, dealing with larger inputs efficiently will be a critical area of focus. Optimizing memory usage and finding ways to parallelize the algorithm can help manage large-scale data in real-world scenarios. Implementing distributed computing techniques or leveraging GPU-based processing could be key strategies to handle larger permutations while maintaining reasonable execution times.

1. **Application to Real-World Problems:**

The concept of good triplets has potential applications in various domains such as genetics, where sequences of DNA need to be analyzed for patterns in multiple datasets, or in social network analysis, where relationships between users might need to be compared across different platforms. Extending the use case beyond simple integer permutations could open doors to more practical and interdisciplinary applications.

1. **Exploring Statistical or Approximate Solutions:**

For scenarios where exact solutions are less critical, future work could explore statistical or approximate methods for counting good triplets. Using sampling techniques or probabilistic algorithms could provide faster results in exchange for minor approximations. This could be especially useful in fields like big data analysis or machine learning, where perfect accuracy is not always necessary.

1. **Incorporating Additional Constraints:**

Future research could explore adding more complex constraints to the problem, such as finding triplets that also satisfy additional conditions related to specific value ranges, or incorporating external factors (such as weights or priorities). These extended problems can lead to more complex and specialized solutions, offering insights into a broader class of combinatorial optimization problems.

1. **Visualization and Analytics:**

As the problem grows in complexity, providing visual representations of the solution process might help to better understand the dynamics of good triplet identification. Developing interactive tools or dashboards that allow users to visualize triplet selection across different arrays could offer educational value and practical insights, especially when dealing with multi-dimensional data.

**GANTT CHART**

|  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| DURATION TASK | DAY  1 | DAY  2 | DAY  3 | DAY  4 | DAY  5 | DAY  6 | DA  Y 7 | DA  Y 8 | DAY  9 |
| PROBLEM STATEMENT |  |  |  |  |  |  |  |  |  |
| ABSTRACT |  |  |  |  |  |  |  |  |  |
| INTRODUCTI ON |  |  |  |  |  |  |  |  |  |
| UNDERSTAND ING PROBLEM |  |  |  |  |  |  |  |  |  |
| OBJECTIVE AND  APPROACH |  |  |  |  |  |  |  |  |  |
| CODING |  |  |  |  |  |  |  |  |  |
| TIME  COMPLEXITY  ANALYSIS |  |  |  |  |  |  |  |  |  |
| FUTURE SCOPE |  |  |  |  |  |  |  |  |  |
| PRESENTATIO  N |  |  |  |  |  |  |  |  |  |

**CONCLUSION:**

The project successfully accomplished the task of counting good triplets in two given permutation arrays, demonstrating a clear and methodical approach to solving the problem. The algorithm was implemented and tested within the provided constraints, and the results validated its correctness and functionality. While the current methodology achieves the desired outcome, there is room for further optimization, particularly in terms of improving time complexity to handle larger datasets more efficiently. Future work can focus on refining the algorithm to reduce computational overhead, exploring alternative methods, and extending the problem's applicability to more complex or generalized scenarios. Through these enhancements, the solution could become more versatile and scalable for real-world applications.