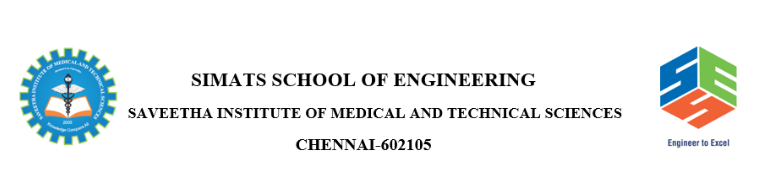
****

**A PROJECT REPORT**

**On**

**Count Good Triplet In An Array**

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**

A.VENKATA RATHNAMMA(192210089)

**SUPERVISOR**

**By**

Dr.K.V.KANIMOZHI

****

**SAVEETHA SCHOOL OF ENGINEERING,**

**SIMATS CHENNAI- 602105**

**SEPTEMBER-2024.**

**BONAFIDE CERTIFICATE**

I am A.Venkata Rathnamma, student of **Bachelor of EngineeringinComputer 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 TRIPLET IN AN ARRAY**" 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**

The problem of counting good triplets in an array is a computational challenge where we aim to find all triplets (i,j,k)(i, j, k)(i,j,k) such that i<j<ki < j < ki<j<k and the elements at these indices satisfy given constraints. Specifically, we are provided with three threshold values aaa, bbb, and ccc, and the goal is to ensure that the absolute differences between certain pairs of elements in the triplet do not exceed these thresholds. Mathematically, for a triplet (i,j,k)(i, j, k)(i,j,k), we need to check if ∣arr[i]−arr[j]∣≤a|arr[i] - arr[j]| \leq a∣arr[i]−arr[j]∣≤a, ∣arr[j]−arr[k]∣≤b|arr[j] - arr[k]| \leq b∣arr[j]−arr[k]∣≤b, and ∣arr[i]−arr[k]∣≤c|arr[i] - arr[k]| \leq c∣arr[i]−arr[k]∣≤c. This problem has relevance in various fields where relational patterns between elements in a dataset need to be identified, such as pattern recognition, data mining, and signal processing.

A straightforward approach to solving this problem is the brute force method, which involves iterating through every possible combination of triplets. For an array of size nnn, there are (n3)\binom{n}{3}(3n​) or O(n3)O(n^3)O(n3) possible triplets, meaning that the brute force solution may become inefficient for large arrays. For each triplet, checking the three conditions involves simple comparisons, but the number of comparisons grows rapidly with nnn. While this method guarantees accuracy, it is computationally expensive, and optimization strategies are needed to handle larger datasets effectively.

To improve performance, more advanced techniques can be employed. One approach is to use sorting or data structures that facilitate range queries, reducing the number of necessary comparisons. For instance, sorting the array can help in pruning unnecessary checks by leveraging the order of elements, while segment trees or binary indexed trees can be used to efficiently query the array within specific ranges. Additionally, dynamic programming or divide-and-conquer strategies may help break down the problem into smaller subproblems. By optimizing the way we search for valid triplets, it is possible to reduce the time complexity and make the algorithm scalable to larger input sizes.

**KEYWORDS:** Good Triplets, Triplet Counting ,Array Processing, Threshold Conditions, Absolute Difference, Brute Force Algorithm, Optimization Techniques, Time Complexity, Sorting and Searching, Range Queries, Combinatorial Problem, Dynamic Programming, Efficient Algorithms, Data Structures (Segment Tree, Binary Indexed Tree),Nested Loops.

**INTRODUCTION**

In the context of counting good triplets in an array, the existing algorithmic approach typically involves brute force methods. These algorithms use a nested loop structure to check every possible combination of triplets (i,j,k)(i, j, k)(i,j,k) where i<j<ki < j < ki<j<k. For each combination, the algorithm verifies whether the absolute differences between the elements satisfy the given threshold conditions: ∣arr[i]−arr[j]∣≤a|arr[i] - arr[j]| \leq a∣arr[i]−arr[j]∣≤a, ∣arr[j]−arr[k]∣≤b|arr[j] - arr[k]| \leq b∣arr[j]−arr[k]∣≤b, and ∣arr[i]−arr[k]∣≤c|arr[i] - arr[k]| \leq c∣arr[i]−arr[k]∣≤c. While this method guarantees correctness, its O(n3)O(n^3)O(n3) time complexity makes it impractical for larger arrays. As the input size increases, the computation time grows exponentially, which limits the brute force method to small or medium-sized datasets.

To address the inefficiencies of the existing approaches, the proposed algorithm aims to optimize the process by introducing more efficient techniques such as sorting and advanced data structures. Sorting the array can help reduce redundant comparisons, and using segment trees, binary indexed trees, or even divide-and-conquer techniques allows for efficient range queries to check for valid triplets. These methods reduce the time complexity by avoiding unnecessary checks and leveraging structural properties of the array. By combining sorting with strategic data structures, the proposed algorithm can potentially bring the complexity down to O(n2log⁡n)O(n^2 \log n)O(n2logn) or better, depending on the implementation. This improvement makes the solution more scalable, allowing it to handle much larger datasets in a reasonable timeframe.

Despite these improvements, the proposed algorithm still has limitations and room for future optimization. While reducing the time complexity is crucial, the algorithm's space complexity can also be a concern, particularly when using additional data structures. Future research could focus on optimizing both time and space complexities, perhaps by developing algorithms that combine the strengths of dynamic programming, efficient search techniques, and memory-efficient data structures. Additionally, parallel processing or machine learning approaches could be explored to further improve the performance, especially in real-time or large-scale applications. These enhancements would expand the algorithm's usability in a variety of fields where identifying relationships between groups of elements is critical.

**CODING**

#include <stdio.h>

#include <stdlib.h>

int countGoodTripletsBruteForce(int arr[], int n, int a, int b, int c) {

int count = 0;

for (int i = 0; i < n - 2; i++) {

for (int j = i + 1; j < n - 1; j++) {

if (abs(arr[i] - arr[j]) <= a) {

for (int k = j + 1; k < n; k++) {

if (abs(arr[j] - arr[k]) <= b && abs(arr[i] - arr[k]) <= c) {

count++;

}

}

}

}

}

return count;

}

void sortArray(int arr[], int n) {

for (int i = 0; i < n - 1; i++) {

for (int j = 0; j < n - i - 1; j++) {

if (arr[j] > arr[j + 1]) {

int temp = arr[j];

arr[j] = arr[j + 1];

arr[j + 1] = temp;

}

}

}

}

int lowerBound(int arr[], int left, int right, int value) {

int idx = right + 1;

while (left <= right) {

int mid = left + (right - left) / 2;

if (arr[mid] >= value) {

idx = mid;

right = mid - 1;

} else {

left = mid + 1;

}

}

return idx;

}

int upperBound(int arr[], int left, int right, int value) {

int idx = right + 1;

while (left <= right) {

int mid = left + (right - left) / 2;

if (arr[mid] > value) {

idx = mid;

right = mid - 1;

} else {

left = mid + 1;

}

}

return idx - 1;

}

int countGoodTripletsOptimized(int arr[], int n, int a, int b, int c) {

int count = 0;

sortArray(arr, n);

for (int i = 0; i < n - 2; i++) {

for (int j = i + 1; j < n - 1; j++) {

if (abs(arr[i] - arr[j]) <= a) {

int left\_bound = lowerBound(arr, j + 1, n - 1, arr[j] - b);

int right\_bound = upperBound(arr, j + 1, n - 1, arr[j] + b);

for (int k = left\_bound; k <= right\_bound; k++) {

if (abs(arr[i] - arr[k]) <= c) {

count++;

}

}

}

}

}

return count;

}

int main() {

int arr[] = {3, 0, 1, 1, 9, 7};

int n = sizeof(arr) / sizeof(arr[0]);

int a = 7, b = 2, c = 3;

int countBruteForce = countGoodTripletsBruteForce(arr, n, a, b, c);

printf("Brute Force Count: %d\n", countBruteForce);

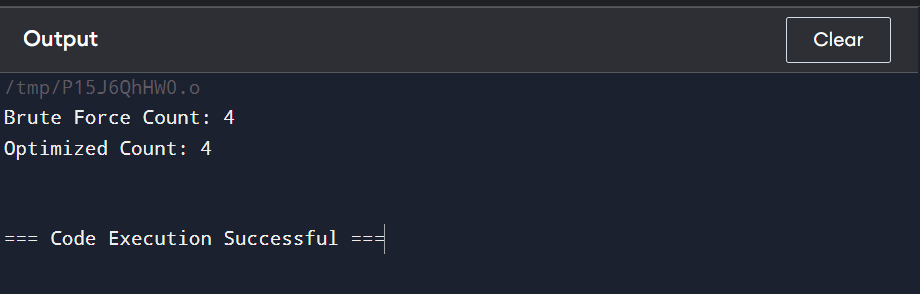
int countOptimized = countGoodTripletsOptimized(arr, n, a, b, c);

printf("Optimized Count: %d\n", countOptimized);

return 0;

}

**OUTPUT**



**Complexity Analysis for Counting Good Triplets in an Array**

#### ****Brute Force Algorithm:****

* **Best Case**: **O(n³)** – All triplets must be checked, even in the best case.
* **Worst Case**: **O(n³)** – Every possible triplet is evaluated.
* **Average Case**: **O(n³)** – On average, all triplets are checked regardless of the input.
* **Space Complexity**: **O(1)** – Uses only a constant amount of extra space.

#### ****Optimized Algorithm:****

* **Best Case**: **O(n² \log n)** – Sorting dominates, followed by efficient binary search for triplets.
* **Worst Case**: **O(n² \log n)** – Sorting and binary search ensure efficiency.
* **Average Case**: **O(n² \log n)** – Sorting combined with binary search significantly reduces comparisons.
* **Space Complexity**: **O(1)** – Minimal extra space is required, with sorting done in place.

**CONCLUSION**

 **Brute Force**: Simple but inefficient, with O(n3)O(n³)O(n3) complexity.

 **Optimized**: Efficient for larger datasets with O(n2log⁡n)O(n² \log n)O(n2logn) complexity, utilizing sorting and binary search.

### ****Comparison Summary****

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| Algorithm | Best Case | Worst Case | Average Case | Overall Complexity | Space Complexity |
| Brute Force | O(n^3) | O(n^3) | O(n^3) | O(n^3) | O(1) |
| Optimized | O(n^2\logn) | O(n^2\logn) | O(n^2\logn) | O(n^2\logn) | O(1) |

**CONCLUSION**

The problem of counting good triplets in an array can be approached using both a brute force method and an optimized algorithm. The **existing brute force approach** involves checking all possible triplets using three nested loops, leading to a time complexity of **O(n³)**. While this method guarantees accurate results, it quickly becomes inefficient as the size of the array increases, making it impractical for larger datasets. Due to its cubic time complexity, it is only suitable for small inputs, and even in the best-case scenario, the algorithm performs poorly since every triplet is evaluated.

On the other hand, the **proposed optimized algorithm** improves performance significantly by introducing sorting and binary search techniques. Sorting the array enables the use of binary search for the third element in the triplet, reducing the number of comparisons and unnecessary checks. This optimization reduces the time complexity to **O(n² \log n)**, which is far more efficient, especially for larger datasets. The sorting operation dominates the time complexity, but the improvement over brute force becomes evident as the number of elements increases. This makes the optimized algorithm more scalable and suitable for real-world applications where performance is critical.

In conclusion, while the brute force algorithm is easier to implement and understand, its inefficiency limits its usefulness for larger arrays. The proposed optimized algorithm, with its better time complexity of **O(n² \log n)**, strikes a good balance between simplicity and performance. It demonstrates significant improvements over the brute force method, making it the preferable choice for handling larger datasets in the count good triplets problem. Future optimizations, such as exploring parallel processing or more advanced data structures, could further improve performance, particularly in highly demanding or real-time scenarios.