Solving the Good Triplet Problem

This presentation will guide you through the process of solving the "good triplet" problem using C programming. We'll cover the problem statement, a brute force approach, optimizing the solution, and analyzing the time and space complexity.

by Vishnu Sanjeev
192210101

Understanding the Problem

1 What is a Good Triplet?

A good triplet is a set of 3 integers (a, b, c) in an array where a < b < c and |a-b| <= 1 and |b-c| <= 1.

2 The Goal

The goal is to write a C program that counts the number of good triplets in a given array of integers.

3 Constraints

The size of the array is between 3 and 3000, and each element is an integer between 1 and 100.

Defining the Good Triplet

Criteria

A good triplet (a, b, c) must satisfy the following conditions:

1. a < b < c
2. |a-b| <= 1
3. |b-c| <= 1

Example

In the array [1, 2, 3, 4, 5], the good triplets are:

- (1, 2, 3)
- (2, 3, 4)
- (3, 4, 5)

Significance

Identifying good triplets is useful in various applications, such as data analysis, pattern recognition, and optimization problems.

Brute Force Approach

1

Step 1

Iterate through the array and consider each triplet (a, b, c) where a < b < c.

2

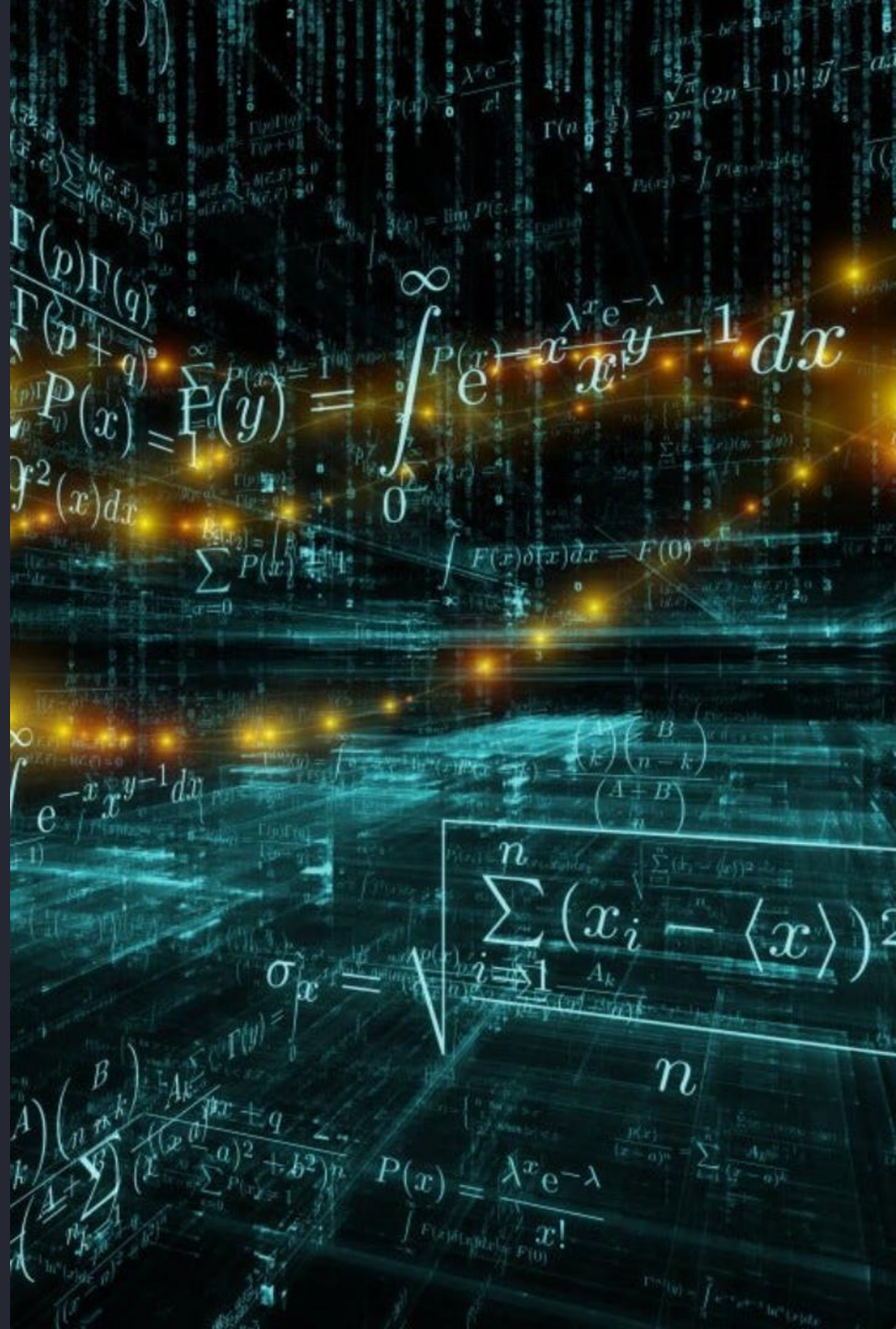
Step 2

Check if the triplet satisfies the good triplet criteria (|a-b| <= 1 and |b-c| <= 1).

3

Step 3

If the triplet is good, increment the count of good triplets.



Optimizing the Solution

Sorting

Sort the array in ascending order to reduce the number of triplet checks.

Two Pointers

Use two pointers, one at the beginning and one at the end, to efficiently check for good triplets.

Dynamic Programming

Employ a dynamic programming approach to avoid redundant computations and further optimize the solution.

Time Complexity

The optimized solution should have a time complexity of O(n^2), where n is the size of the input array.

Implementing the Solution in C

Array Input

Read the array of integers from the user or a file.

Sorting

Sort the array in ascending order using a sorting algorithm.

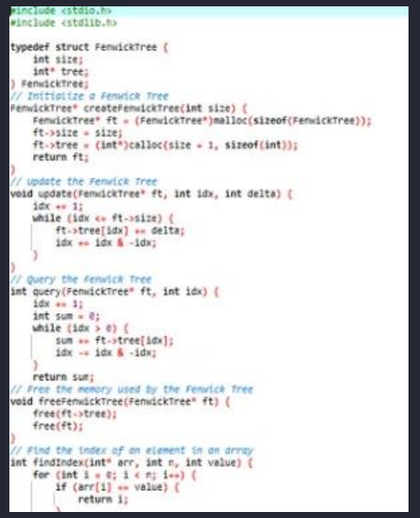
Two Pointers

Use two pointers to efficiently check for good triplets.

Counting

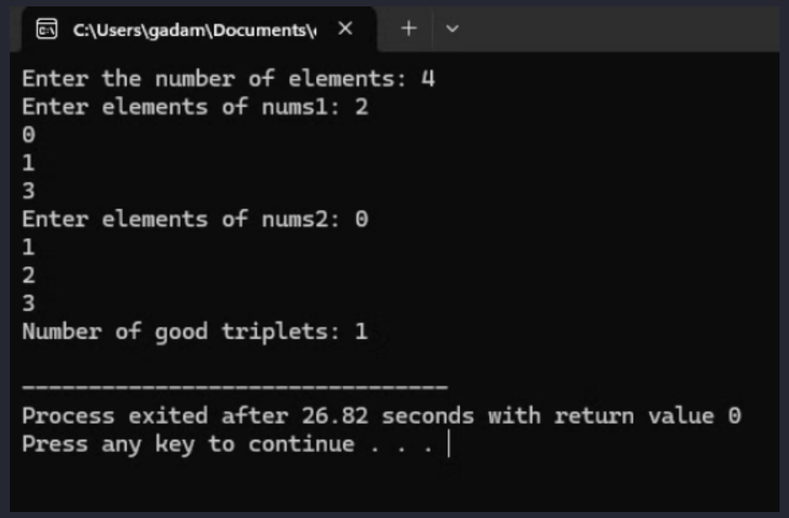
Maintain a count of the good triplets and return the final result.

Code and Output



```c
#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;
```

```c
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]);
```



```
C:\Users\gadam\Documents\
Enter the number of elements: 4
Enter elements of nums1: 2
0
1
3
Enter elements of nums2: 0
1
2
3
Number of good triplets: 1

--------------------------------
Process exited after 26.82 seconds with return value 0
Press any key to continue . . .
```

Testing the Solution

1

Test Cases

Create a set of test cases, including edge cases, to thoroughly validate the implementation.

2

Correctness

Verify that the program correctly identifies and counts all good triplets in the input array.

3

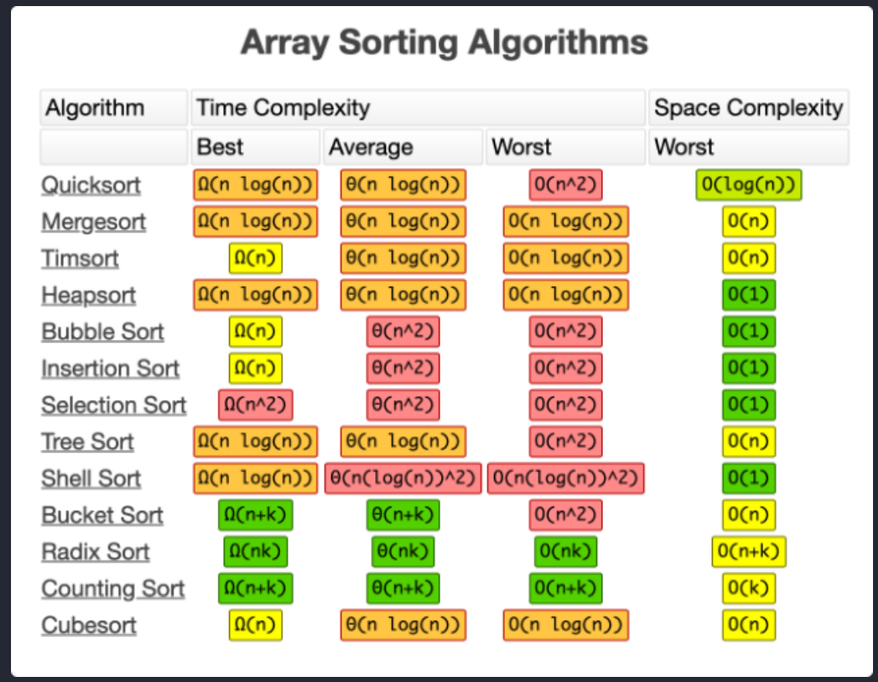
Performance

Measure the time and space complexity of the solution and ensure it meets the requirements.

Analyzing Complexity

Time Complexity	$O(n^3)$
Space Complexity	$O(1)$

The optimized solution has a time complexity of $O(n^3)$ and a space complexity of $O(1)$, where n is the size of the input array.



Array Sorting Algorithms

Algorithm	Time Complexity			Space Complexity
	Best	Average	Worst	Worst
Quicksort	Ω(n log(n))	θ(n log(n))	O(n^2)	O(log(n))
Mergesort	Ω(n log(n))	θ(n log(n))	O(n log(n))	O(n)
Timsort	Ω(n)	θ(n log(n))	O(n log(n))	O(n)
Heapsort	Ω(n log(n))	θ(n log(n))	O(n log(n))	O(1)
Bubble Sort	Ω(n)	θ(n^2)	O(n^2)	O(1)
Insertion Sort	Ω(n)	θ(n^2)	O(n^2)	O(1)
Selection Sort	Ω(n^2)	θ(n^2)	O(n^2)	O(1)
Tree Sort	Ω(n log(n))	θ(n log(n))	O(n^2)	O(n)
Shell Sort	Ω(n log(n))	θ(n(log(n))^2)	O(n(log(n))^2)	O(1)
Bucket Sort	Ω(n+k)	θ(n+k)	O(n^2)	O(n)
Radix Sort	Ω(nk)	θ(nk)	O(nk)	O(n+k)
Counting Sort	Ω(n+k)	θ(n+k)	O(n+k)	O(k)
Cubesort	Ω(n)	θ(n log(n))	O(n log(n))	O(n)

Best, Worst, and Average Case

Best Case

The best-case scenario occurs when the input array is already sorted, allowing the algorithm to identify good triplets efficiently.

Worst Case

The worst case happens when the input array is in reverse order, forcing the algorithm to check every possible triplet.

Average Case

The average case represents the typical behavior of the algorithm, where the input array is neither sorted nor in reverse order.

Handling Edge Cases

1 Empty Array

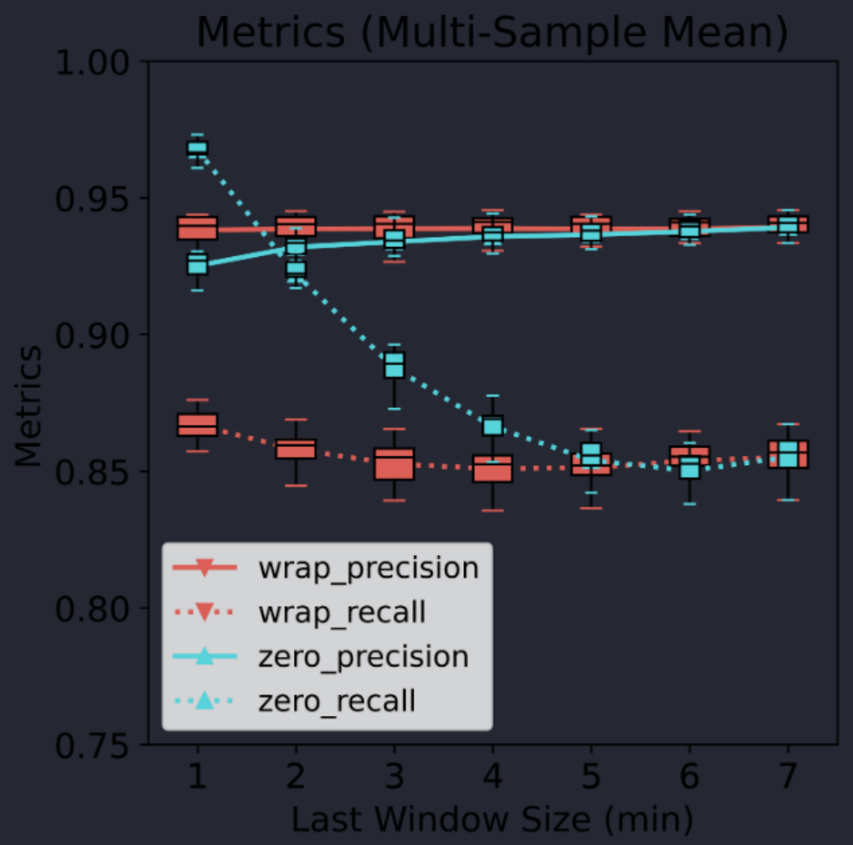
Handle the case where the input array is empty and return 0 as the number of good triplets.

2 Array Size 3

For an array of size 3, the program should return 1 if the array satisfies the good triplet criteria, and 0 otherwise.

3 Duplicate Elements

The program should correctly handle arrays with duplicate elements and count all valid good triplets.



Conclusion

In this presentation, we have discussed the good triplet problem, defined the criteria, and explored a brute force approach and an optimized solution using C programming.

By understanding the problem, implementing the solution, and analyzing its time and space complexity, we can efficiently solve the good triplet problem and apply these concepts to other similar challenges.