Problem Recap

Given an integer array nums, return all the triplets [nums[i], nums[j], nums[k]] such that i != j, i != k, and j != k, and nums[i] + nums[j] + nums[k] == 0. The solution set must not contain duplicate triplets.

Approach 1: Brute Force

Idea:

- Iterate through all possible triplets in the array and check if their sum equals zero.
- Use three nested loops to generate all possible combinations of i, j, and k.

```
vector<vector<int>> threeSum(vector<int>& nums) {
    vector<vector<int>> result;
    int n = nums.size();

    for (int i = 0; i < n - 2; i++) {
        for (int j = i + 1; j < n - 1; j++) {
            for (int k = j + 1; k < n; k++) {
                if (nums[i] + nums[j] + nums[k] == 0) {
                    vector<int> triplet = {nums[i], nums[j], nums[k]};
                    sort(triplet.begin(), triplet.end());
                    if (find(result.begin(), result.end(), triplet) ==
result.end()) {
                        result.push_back(triplet);
                    }
                }
            }
        }
    }

    return result;
}
```

Complexity:

- **Time Complexity:** O(n^3), where n is the number of elements in nums.
- **Space Complexity:** O(n) for storing the result.

Approach 2: Improved Approach with Sorting and Two-Pointers

Idea:

- Sort the array first.
- Use a fixed pointer for nums[i] and use two pointers (left and right) for nums[j] and nums[k] to find the sum 0.
- Skip duplicates to avoid repeated triplets.

```cpp
class Solution {
public:
    vector<vector<int>> threeSum(vector<int>& nums) {
        vector<vector<int>> result;
        int n = nums.size();
        sort(nums.begin(), nums.end());

        for (int i = 0; i < n - 2; i++) {
            if (i > 0 && nums[i] == nums[i - 1]) continue; // Skip duplicate
`nums[i]`

            int left = i + 1;
            int right = n - 1;

            while (left < right) {
                int sum = nums[i] + nums[left] + nums[right];

                if (sum == 0) {
                    result.push_back({nums[i], nums[left], nums[right]});
                    while (left < right && nums[left] == nums[left + 1])
left++; // Skip duplicate `nums[left]`
                    while (left < right && nums[right] == nums[right - 1])
right--; // Skip duplicate `nums[right]`
                    left++;
                    right--;
                } else if (sum < 0) {
                    left++;
                } else {
                    right--;
                }
            }
        }

        return result;
    }
};
```

Complexity:

- **Time Complexity:** O(n^2) due to the double loop with two pointers.
- **Space Complexity:** O(n) for storing the result.

Approach 3: Optimal Approach with Hashing (Optional)

Idea:

- This approach would use hashing to check for the existence of required elements, but the two-pointer method is usually preferred due to better control over duplicate management.

Explanation of the Two-Pointer Approach:

- **Step 1:** Sort the array.
- **Step 2:** Fix one element (nums[i]) and use two pointers (left and right) to find the other two elements.
- **Step 3:** Move the pointers accordingly:
    - If the sum is less than 0, increment the left pointer.
    - If the sum is greater than 0, decrement the right pointer.
    - If the sum equals 0, add the triplet to the result and skip duplicates.

This approach is efficient and handles duplicates naturally, making it a widely used solution for this problem.