3.Hard\03.3_sum.cpp

```cpp
/*
QUESTION:
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.

Example:
Input: nums = [-1,0,1,2,-1,-4]
Output: [[-1,-1,2],[-1,0,1]]
Explanation:
nums[0] + nums[1] + nums[2] = (-1) + 0 + 1 = 0.
nums[1] + nums[2] + nums[4] = 0 + 1 + (-1) = 0.
nums[0] + nums[3] + nums[4] = (-1) + 2 + (-1) = 0.
The distinct triplets are [-1,0,1] and [-1,-1,2].
Notice that the order of the output and the order of the triplets does not matter.
*/

/*
APPROACH:
To find all triplets that sum up to zero, we can follow these steps:
1. Sort the input array in non-decreasing order.
2. Iterate through the array and fix the first element as nums[k] (where k = 0 to n-1).
3. Use two pointers (i and j) to find the other two elements such that nums[i] + nums[j] = -nums[k].
4. Move the pointers accordingly to find all possible triplets.
5. Skip duplicate elements to avoid duplicate triplets.
6. Return the resulting triplets.
*/

vector<vector<int>> threeSum(vector<int> &nums)
{
    vector<vector<int>> ans;
    sort(nums.begin(), nums.end());

    for (int k = 0; k < nums.size(); k++)
    {
        int i = k + 1;
        int j = nums.size() - 1;
        int target = -nums[k];

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

            if (sum == target)
            {
                ans.push_back({nums[k], nums[i], nums[j]});
                i++;
                j--;

                // Skip duplicate elements
                while (i < j && nums[i] == nums[i - 1])
                    i++;
```

```cpp
                while (i < j && nums[j] == nums[j + 1])
                    j--;
            }
            else if (sum < target)
            {
                i++;
            }
            else
            {
                j--;
            }
        }

        // Skip duplicate elements
        while (k + 1 < nums.size() && nums[k + 1] == nums[k])
            k++;
    }

    return ans;
}

/*
TIME COMPLEXITY: O(n^2), where n is the size of the input array.
The sorting step takes O(n log n), and the two-pointer traversal takes O(n^2) in the worst case.
Hence, the overall time complexity is O(n^2).

SPACE COMPLEXITY: O(1), as we are using a constant amount of extra space for storing the output and variables.
*/

```