<https://leetcode.com/problems/3sum-with-multiplicity/description/>

Given an integer array arr, and an integer target, return the number of tuples i, j, k such that i < j < k and arr[i] + arr[j] + arr[k] == target.

As the answer can be very large, return it modulo 10^9 + 7.

**Example 1:**

**Input:** arr = [1,1,2,2,3,3,4,4,5,5], target = 8

**Output:** 20

**Explanation:** Enumerating by the values (arr[i], arr[j], arr[k]):(1, 2, 5) occurs 8 times;(1, 3, 4) occurs 8 times;(2, 2, 4) occurs 2 times;(2, 3, 3) occurs 2 times.

**Example 2:**

**Input:** arr = [1,1,2,2,2,2], target = 5

**Output:** 12

**Explanation:** arr[i] = 1, arr[j] = arr[k] = 2 occurs 12 times:We choose one 1 from [1,1] in 2 ways,and two 2s from [2,2,2,2] in 6 ways.

**Example 3:**

**Input:** arr = [2,1,3], target = 6

**Output:** 1

**Explanation:** (1, 2, 3) occured one time in the array so we return 1.

**Constraints:**

3 <= arr.length <= 3000

0 <= arr[i] <= 100

0 <= target <= 300

**Attempt 1: 2024-06-13**

**Solution 1: Three Pointers (60 min)**

class Solution {

    public int threeSumMulti(int[] A, int target) {

        Arrays.sort(A);

        int mod = 1000000007;

        int result = 0;

        for(int i = 0; i < A.length - 2; i++) {

            int lo = i + 1;

            int hi = A.length - 1;

            long count = 0;

            while(lo < hi) {

                if(A[lo] + A[hi] == target - A[i]) {

                    // Count repeat elements from two directions, and

                    // their production means how many tuples result same

                    if(A[lo] != A[hi]) {

                        long countL = 1;

                        long countR = 1;

                        while(lo + 1 < hi && A[lo + 1] == A[lo]) {

                            countL++;

                            lo++;

                        }

                        while(hi - 1 > lo && A[hi - 1] == A[hi]) {

                            countR++;

                            hi--;

                        }

                        count += (countL \* countR) % mod;

                        lo++;

                        hi--;

                    } else {

                        // Since we already sort the array if A[lo] == A[hi] means

                        // all elements between index lo and hi in A are same value

                        long n = hi - lo + 1;

                        count += (n \* (n - 1) / 2) % mod;

                        // No need to continue current loop since all elements are same

                        break;

                    }

                } else if(A[lo] + A[hi] > target - A[i]) {

                    hi--;

                } else {

                    lo++;

                }

            }

            result = (int)(result + count) % mod;

        }

        return result;

    }

}

Time complexity: O(N^2)

Space complexity: O(1)

**Refer to**

<https://leetcode.com/problems/3sum-with-multiplicity/discuss/181080/Java-sorting-solution-O(N2)>

The solution is similar to 3Sum problem (<https://leetcode.com/submissions/detail/128547721/).>

**The key is how to skip duplicate numbers:**

Case 1: A[l] == A[r], the total possible answer is num \* (num-1) / 2, where num = r-l+1.

Case 2: A[l] != A[r], find how many duplicates in the left-hand side (which is cntL) and how many duplicates in the right-hand side (which is cntR). Then, search for the next possible answers.

class Solution {

public int threeSumMulti(int[] A, int target) {

int mod = 1000000007;

int ans = 0;

Arrays.sort(A);

for(int i = 0; i < A.length-2; i++) {

long cnt = 0;

int l = i+1, r = A.length-1;

while(l < r) {

if(A[i] + A[l] + A[r] > target) r--;

else if(A[i] + A[l] + A[r] < target) l++;

else {

if(A[l] != A[r]) {

long cntL = 1, cntR = 1;

while(l+1 < r && A[l] == A[l+1]) { cntL++; l++; }

while(l < r-1 && A[r] == A[r-1]) { cntR++; r--; }

cnt += (cntL \* cntR) % mod;

l++;

r--;

} else {

long n = r-l+1;

cnt += (n \* (n-1) / 2) % mod;

break;

}

}

}

ans = (int) (ans + cnt) % mod;

}

return ans;

}

}

Time complexity: O(N^2)

Space complexity: O(1)

**Solution 2: Two Pointers + Hash Table (60 min)**

class Solution {

    public int threeSumMulti(int[] A, int target) {

        Map<Integer, Integer> map = new HashMap<Integer, Integer>();

        int result = 0;

        int mod = 1000000007;

        for(int i = 0; i < A.length; i++) {

            result = (result + map.getOrDefault(target - A[i], 0)) % mod;

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

                map.put(A[j] + A[i], map.getOrDefault(A[j] + A[i], 0) + 1);

            }

        }

        return result;

    }

}

Time complexity: O(N^2)

Space complexity: O(1)

**Refer to**

<https://leetcode.com/problems/3sum-with-multiplicity/solutions/181128/10-lines-Super-Super-Easy-Java-Solution/>

**Think Outside of The Box!**

Intuitively, you will try to solve it based on the solution of 3Sum.

But... Build a map for counting different sums of two numbers. The rest of things are straightfoward.

class Solution {

public int threeSumMulti(int[] A, int target) {

Map<Integer, Integer> map = new HashMap<>();

int res = 0;

int mod = 1000000007;

for (int i = 0; i < A.length; i++) {

res = (res + map.getOrDefault(target - A[i], 0)) % mod;

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

int temp = A[i] + A[j];

map.put(temp, map.getOrDefault(temp, 0) + 1);

}

}

return res;

}

}

**Solution 3: Two Pointers + Hash Table (60 min, this solution take 302 ms)**

class Solution {

    private static final int MOD = (int)1e9 + 7;

    public int threeSumMulti(int[] arr, int target) {

        int[] freq = new int[101];

        for(int num : arr) {

            freq[num]++;

        }

        long result = 0;

        for(int j = 0; j < arr.length; j++) {

            int second = arr[j];

            // Decrement count since this number is being used in the current triplet

            freq[second]--;

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

                int first = arr[i];

                int third = target - first - second;

                if(third >= 0 && third <= 100) {

                    result = (result + freq[third]) % MOD;

                }

            }

        }

        return (int) result;

    }

}

Time Complexity: O(N^2)

Space Complexity: O(1)

**Refer to**

<https://algo.monster/liteproblems/923>

**Problem Description**

The problem requires us to find the number of distinct triplets (i, j, k) in an integer array arr where the condition i < j < k is satisfied, and the sum of the elements at these indices is equal to a given target integer. Specifically, arr[i] + arr[j] + arr[k] == target should hold true for the counted triplets. Since the number of such triplets can be quite large, we are asked to return the answer modulo 10^9 + 7 to keep the output within integer value limits.

**Intuition**

The key to solving this problem is to consider how we can traverse the array to find all the valid triplets efficiently. We want to avoid checking all possible triplets due to the high time complexity that approach would imply.

The algorithm usually starts by counting the occurrences of each number in the array using a Counter, which is a special kind of dictionary in Python. This allows us to know how many times each number appears in the array without traversing it multiple times.

Once we have these counts, we iterate through the array for the second number of the triplet. **For each possible second number b at index j, we decrease its count by 1 to ensure that we do not use the same instance of b when looking for the third number.**

**Next, we traverse the array again up to the second number's index j to find every possible first number a. With both a and b known, we calculate the required third number c by subtracting the sum of a and b from the target.**

If c is present in our Counter (which means it's somewhere in the original array), we can form a triplet (a, b, c) that sums up to target. We then add the count of c from our Counter to our answer, since there are as many possibilities to form a triplet with a and b as the count of c. It's important to use modulo with 10^9 + 7 during each addition to keep the number within bounds.

The iteration is cleverly structured to ensure that each element is used according to its occurrence and that i < j < k always holds true by managing the index and counts carefully

**Solution Approach**

The solution uses a combination of a hashmap (in Python, a Counter) and a two-pointer approach to find the valid triplets.

The algorithm goes as follows:

A Counter (which is a specialized hashmap/dictionary in Python) is initialized to count the occurrences of the elements in the given arr. This data structure allows for O(1) access to the count of each element, which is essential for efficient computation of the number of triplets.

We define a variable ans to keep the running total of the number of valid triplets.

The mod variable is set to 10\*\*9 + 7 to ensure that we perform all our arithmetic operations modulo this number.

We loop through each element b of the array using its index j. This element is considered the second element of our potential triplet.

Before starting the inner loop, we decrease the count of b in the Counter by one. This ensures that we don't count the same element b twice when looking for the third element of the triplet.

An inner loop runs through the array up to the current index j, selecting each element a as a candidate for the first element of the triplet. The index i is implicit in this loop.

We then calculate the required third element c of the triplet by subtracting the sum of a and b from the target, i.e., c = target - a - b.

The total count of valid triplets is then incremented by the count of c from the Counter if c is present. We use modulo mod to keep the answer within the range of valid integers.

Finally, we return ans as the result.

The algorithm ensures that no element is used more often than it appears in the array, and the i < j < k condition is naturally upheld by the two nested loops and the management of the Counter.

The use of the Counter and looping through the array only once per element significantly reduces the computational complexity compared to checking all possible triplets directly. This pattern is a common approach in problems involving counting specific arrangements or subsets in an array, especially when the array elements are bound to certain conditions.

**Example Walkthrough**

Let's consider an example with arr = [1, 1, 2, 2, 3, 3] and target = 6. We want to find all unique triplets i, j, k (with i < j < k) such that arr[i] + arr[j] + arr[k] == target.

We begin by initializing a Counter to count the occurrences of each element in arr. The Counter will look like this: {1:2, 2:2, 3:2} which reflects that each number 1, 2, and 3 occurs twice in the array.

Set ans = 0 to keep track of the total number of valid triplets and mod = 10\*\*9 + 7 for modulo operations.

We now look for the second number b for all triplets by iterating through arr. Consider j=2 where b = arr[j] = 2.

We decrement the count of b in Counter by 1. Now the Counter will look like this: {1:2, 2:1, 3:2}.

We start the inner loop to select a as the first element of the triplet. We iterate from start to j-1, in this case, from 0 to 1.

For a = arr[0] = 1 at i=0, we calculate the needed c = target - a - b = 6 - 1 - 2 = 3.

The count of c in the Counter is 2, which means we can form two triplets (1, 2, 3) with i=0, j=2. We increment ans by the count of c in Counter. So, ans = (ans + 2) % mod.

For a = arr[1] = 1 at i=1, we calculate c = 6 - 1 - 2 = 3 again. Since i < j, it's valid and we have not used the same a as before. ans is updated to ans = (ans + 2) % mod.

Continue this process for every j from 0 to len(arr), and you'll count all valid triplets.

After finishing, ans is the total number of valid triplets.

For this example, the possible triplets are two instances of (1, 2, 3) using the first 1 and first 2, two using the first 1 and second 2, two using the second 1 and first 2, and two using the second 1 and second 2. These are counted as four unique triplets because the pairs (1, 2) are distinct by their positions. There's no triplet using any two '3's because that would require i not to be less than j. Therefore, ans = 4.

Remember that in a real problem with a large arr, not all triples (a, b, c) will be unique because of duplicate values, and there will be a variable number of contributions to ans from each triplet depending on how many times each value occurs.

**Solution Implementation**

class Solution {

// Define the modulo constant for taking modulus

private static final int MOD = 1\_000\_000\_007; // 1e9 + 7 is represented as 1000000007

public int threeSumMulti(int[] arr, int target) {

int[] count = new int[101]; // Array to store count of each number, considering constraint 0 <= arr[i] <= 100

// Populate the count array with the frequency of each value in arr

for (int num : arr) {

++count[num];

}

long ans = 0; // To store the result, using long to avoid integer overflow before taking the modulus

// Iterate through all elements in arr to find triplets

for (int j = 0; j < arr.length; ++j) {

int second = arr[j]; // The second element in the triplet

--count[second]; // Decrement count since this number is being used in the current triplet

// Iterate from the start of the array to the current index 'j'

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

int first = arr[i]; // The first element in the triplet

int third = target - first - second; // Calculate the third element

// Check if third element is within range and add the count to the answer

if (third >= 0 && third <= 100) {

ans = (ans + count[third]) % MOD; // Use the modulo to avoid overflow and get the correct result

}

}

}

// Cast and return the final answer as an integer

return (int) ans;

}

}

**Time and Space Complexity**

The time complexity of the provided Python code snippet is O(n^2) where n is the number of elements in the input list arr. This complexity arises because there is a nested for-loop where the outer loop runs through the elements of arr (after decrementing the count of the current element), and the inner loop iterates up to the current index j of the outer loop. For each pair (a, b), the code looks up the count of the third element c that is needed to sum up to the target. Even though the lookup in the counter is O(1), the nested loops result in quadratic complexity.

The space complexity of the code is O(m) where m is the number of unique elements in the input list arr. This complexity is due to the use of a Counter to store frequencies of all unique elements in arr. The space taken by the counter will directly depend on the number of unique values.

**Solution 4: Two Pointers + Hash Table + Math (Binomial Coefficient) (60 min, this solution only take 2 ms)**

class Solution {

    public int threeSumMulti(int[] A, int target) {

        long[] c = new long[101];

        for (int a : A) c[a]++;

        long res = 0;

        for (int i = 0; i <= 100; i++)

            for (int j = i; j <= 100; j++) {

                int k = target - i - j;

                if (k > 100 || k < 0) continue;

                if (i == j && j == k)

                    res += c[i] \* (c[i] - 1) \* (c[i] - 2) / 6;

                else if (i == j && j != k)

                    res += c[i] \* (c[i] - 1) / 2 \* c[k];

                else if (j < k)

                    res += c[i] \* c[j] \* c[k];

            }

        return (int)(res % (1e9 + 7));

    }

}

Time Complexity:

3 <= A.length <= 3000, so N = 3000

But 0 <= A[i] <= 100

So my solution is O(N + 101 \* 101)

**Refer to**

<https://leetcode.com/problems/3sum-with-multiplicity/solutions/181131/c-java-python-o-n-101-101/>

Count the occurrence of each number.

using hashmap or array up to you.

Loop i on all numbers,

loop j on all numbers,

check if k = target - i - j is valid.

Add the number of this combination to result.

3 cases covers all possible combination:

i == j == k

i == j != k

i < k && j < k

Time Complexity:

3 <= A.length <= 3000, so N = 3000

But 0 <= A[i] <= 100

So my solution is O(N + 101 \* 101)

<https://leetcode.com/problems/3sum-with-multiplicity/solutions/181131/c-java-python-o-n-101-101/comments/481912>

**Tried to explain the three cases. The task is to pick 3 numbers, now you do not want to name it i, j, k, just three numbers in mind.**

**Case 1: three numbers are the same. Then you will have i == j == k.**

**Case 2: two of the three numbers are equal. For example, 5, 4, 5. To avoid duplicates, you can choose to make i=5, j=5, k = 4. Once this is chosen, you do not want to have others (e.g. i = 5, k =5, j = 4, or j = 5, k =5, i =4). Otherwise, You will make duplicates. In brief, because two numbers are the same, you only want to have either i, j, or j, k, or k, i to be the same numbers. In lee215's solution, he picked up i, j.**

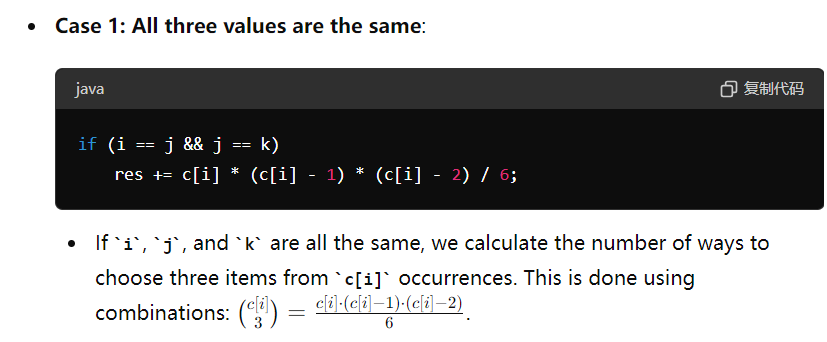
**Also refer to explain from ChatGPT for Case 1 and Case 2:**

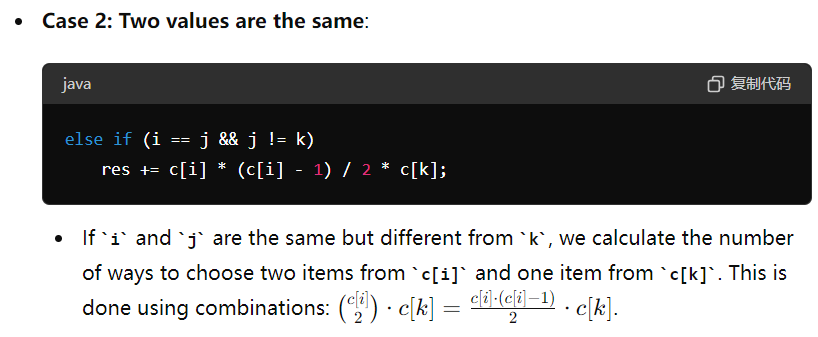
if (i == j && j == k)

res += c[i] \* (c[i] - 1) \* (c[i] - 2) / 6;

else if (i == j && j != k)

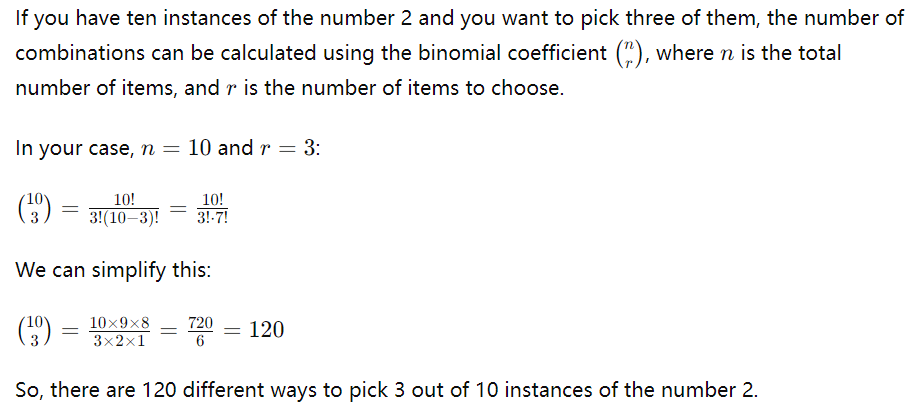
res += c[i] \* (c[i] - 1) / 2 \* c[k];

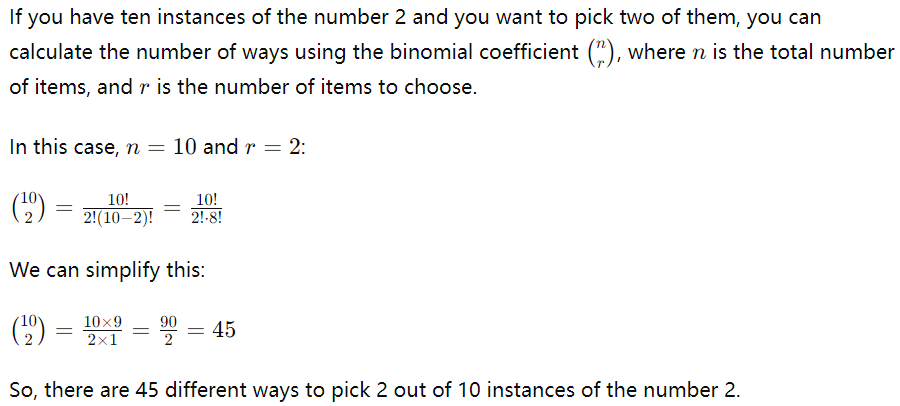




**The mathematics foundation for above same number combinations is below:**

**Binomial Coefficient C(r / n), where n is the total number of items and r is the number of items to choose**





public int threeSumMulti(int[] A, int target) {

long[] c = new long[101];

for (int a : A) c[a]++;

long res = 0;

for (int i = 0; i <= 100; i++)

for (int j = i; j <= 100; j++) {

int k = target - i - j;

if (k > 100 || k < 0) continue;

if (i == j && j == k)

res += c[i] \* (c[i] - 1) \* (c[i] - 2) / 6;

else if (i == j && j != k)

res += c[i] \* (c[i] - 1) / 2 \* c[k];

else if (j < k)

res += c[i] \* c[j] \* c[k];

}

return (int)(res % (1e9 + 7));

}

**Refer to**

[L15.P3.4.3Sum (Ref.L18)](note://EC0082E510FD4360AD113BFC40B23D1F)