1995. Count Special Quadruplets

Enumeration Easy

Problem Description

In this LeetCode problem, we are given an integer array nums. Our task is to find the count of unique quadruplets (a, b, c, d) that fulfill two conditions:

2. The indices a, b, c, and d must follow a strict increasing order, such that a < b < c < d.

The objective is to determine how many such quadruplets exist in the provided array.

1. The sum of three elements at indices a, b, and c is equal to the element at index d: nums[a] + nums[b] + nums[c] == nums[d].

Intuition

observed thus far in the remaining part of the array to the right of b.

To solve this problem, the intuition is to iterate over the array in a structured way so that we can check possible combinations that fulfill the given conditions without redundancy. Instead of checking every possible quadruplet, which could be inefficient, we

can use a counting approach to aid in this process. The solution employs a hashmap, which in Python is provided by the Counter class from the collections module, to keep track of the differences between nums[d] and nums[c]. This is valuable because if we fix b and c, and find a d > c, we can update our counter for the difference nums[d] - nums[c]. Then, for all a < b, we can directly check if the sum nums[a] + nums[b] is present

as a key in our counter. This represents the fact that nums[d] exists such that nums[a] + nums[b] + nums[c] == nums[d]. The approach involves reverse iterating over the potential b indices starting from the third-to-last index down to the first index (since a must be less than b). After fixing b, we iterate over c and d to the end of the array and update the counter for each such pair. Then, in another loop moving from 0 to b, we use the counter to check how many times nums[a] + nums[b] appears as a sum

nums[d] - nums[c], as that indicates valid quadruplets. The count of these occurrences is added to the total answer (ans). **Solution Approach** The implemented solution follows a three-pointer approach that uses the Counter data structure to optimize the searching process for the sum condition specified in the problem statement.

Initialize a variable ans to keep track of the total count of valid quadruplets and obtain the length n of the given nums array.

Here is how the algorithm unfolds:

matching sums for each potential quadruplet.

Perform a reverse iteration over the potential b positions, starting from n-3 and decrementing down to 1. This step ensures

Prepare a Counter object named counter that is going to keep track of the frequency of the differences nums[d] - nums[c]

For each fixed b, iterate forwards through the indices c and d, where c starts immediately after b and d moves past c towards the end of the array. For each c and d pair, update the counter to record the frequency of nums[d] - nums[c]. This setup aids

that there is enough space for c and d to the right, as the condition a < b < c < d must be respected.

in the future checking of the sum condition without extra iteration through the array.

- After populating the counter for a specific b, start another loop from 0 up to b 1 to find valid a indices. With a and b fixed, and the counter holding information for all c and d pairs beyond b, we can find out how many times nums [a] + nums [b]
- appears as a sum nums[d] nums[c]. For each a, increment the total count ans by the value of counter[nums[a] + nums[b]], which represents the number of valid quadruplets with the current a and b. Return the total count ans as the final result.

The Counter is crucial as it serves as a hash map (dictionary in Python) to efficiently record and access frequency counts of

specific sum differences. This allows for constant-time lookups which are much faster than iterating over the array to find

would otherwise have a much higher time complexity. The use of a Counter combines with strategic pointer movement to solve this problem in a more efficient manner. **Example Walkthrough**

Let's consider an example to illustrate the solution approach. Suppose we have the following integer array nums:

By optimizing the lookup process and managing how we iterate through the array, this solution avoids the naive approach that

Let's walk through the algorithm: Initialize ans to 0 as the counter and obtain the length of nums, which is n = 7 in this case. Prepare an empty Counter object named counter to keep track of the frequency of the differences observed.

• For a = 0: The sum nums[a] + nums[b] = 1 + 3 = 4. The counter has 1 for key 1, which does not match, so no valid quadruplet is found

We initialize counter to an empty state again since we shift b leftward. \circ We start by setting c = 5 and d = 6. Since nums[d] - nums[c] = 4 - 3 = 1, update counter to have counter[1] = 1.

For b = 3:

stop the walkthrough here.

Python

class Solution:

Checking each a:

3.

nums = [1, 1, 2, 2, 3, 3, 4]

For b = 4 (nums [b] is 3):

• There's no more room to increment c and d, so we proceed to the next step. Now, for this b (index 4), we loop over all a less than b. We can only consider a at indices 0, 1, 2, and 3.

Begin reverse iteration for b starting from n-3, which is index 4. This leaves room for c and d.

here. \circ For a = 1: The sum nums[a] + nums[b] = 1 + 3 = 4. Again, no valid quadruplet.

For a = 2: The sum nums[a] + nums[b] = 2 + 3 = 5. Again, no valid quadruplet.

• For a = 3: The sum nums[a] + nums[b] = 2 + 3 = 5. Again, no valid quadruplet.

 \circ Move d to 6, nums[d] - nums[c] = 4 - 2 = 2, the counter is updated to counter[2] = 1.

Move b to the next index, 3 (nums[b] is 2), and repeat steps 3 and 4.

from collections import Counter # Importing Counter from collections module

count = 0 # Initialize the count of quadruplets to 0

length = len(nums) # Store the length of the input list

Start c_index from the element right after b_index

Update the frequency counter for each pair (c, d)

frequency_counter[nums[d_index] - nums[c_index]] += 1

count += frequency_counter[nums[a_index] + nums[b_index]]

// Counter array to hold the frequency of differences between nums[d] and nums[c]

// We are iterating from the third last element down to the second element

for d_index in range(c_index + 1, length):

Count quadruplets for each pair (a, b_index)

return count # Return the final count of quadruplets

int count = 0; // Holds the number of valid quadruplets found

// because we need at least two more elements for 'c' and 'd'

for (int d = c + 1; d < size; ++d) {

for (int a = 0; a < b; ++a) {

// Define a global array to store input numbers.

let frequency: number[] = new Array(310).fill(0);

def countQuadruplets(self, nums: List[int]) -> int:

for b_index in range(length - 3, 0, -1):

for a index in range(b index):

frequency_counter = Counter()

c_index = b_index + 1

count = 0 # Initialize the count of quadruplets to 0

Counter to track the frequency of (nums[d] - nums[c])

length = len(nums) # Store the length of the input list

Iterate from the third last element down to the second element

frequency_counter[nums[d_index] - nums[c_index]] += 1

count += frequency_counter[nums[a_index] + nums[b_index]]

Start c_index from the element right after b_index

Update the frequency counter for each pair (c, d)

for d_index in range(c_index + 1, length):

Count quadruplets for each pair (a, b_index)

return count # Return the final count of quadruplets

};

TypeScript

let nums: number[] = [];

if $(nums[d] - nums[c] >= 0) {$

frequency[nums[d] - nums[c]]++;

// Now looking for 'a' which is to the left of 'b'.

// Reset the frequency array for the next iteration.

return count; // Return the final count of special quadruplets.

// Global array to store the frequencies of values for the differences of c and d.

count += frequency[nums[a] + nums[b]];

fill(frequency.begin(), frequency.end(), 0);

// Increment the frequency of this particular difference.

// If a sum of nums[a] and nums[b] happened to be the difference

// This ensures that we only count the differences relevant to the current 'b'.

// previously recorded, it contributes to the total count.

// Initialized to 310 based on the constraints that nums[i] <= 100

int length = nums.length; // The length of the input array

def countQuadruplets(self, nums: List[int]) -> int:

c_index = b_index + 1

for a_index in range(b_index):

int[] differenceCounter = new int[310];

- Reset counter. \circ Start c at 4 and d > c. Suppose c = 4 and d = 5, nums[d] - nums[c] = 3 - 2 = 1, we update the counter to counter[1] = 1.
 - There's no more room for d, move c to 5, and d to 6, nums[d] nums[c] = 4 3 = 1, the counter is now counter[1] = 2 since we've observed another 1. Loop over a indices less than 3:
 - ∘ For a = 2: The sum nums[a] + nums[b] = 2 + 2 = 4. The counter has 2 for key 2, which matches. We found one valid quadruplet, (nums[a], nums[b], nums[c], nums[d]) = (2, 2, 2, 4). So, ans = ans + counter[4 - nums[b]] = 0 + 1 = 1. We would continue to reverse iterate b down to the index 1 and repeat the above steps. However, for the sake of brevity, let's

∘ For a = 0: The sum nums[a] + nums[b] = 1 + 2 = 3. This does not match any key in the counter.

∘ For a = 1: The sum nums[a] + nums[b] = 1 + 2 = 3. Again, no match and no valid quadruplet.

problem by strategically checking for the sums that align with our conditions. Solution Implementation

The total ans at the end of the full iteration (not shown for the entire array to keep this brief) would give us the count of all unique

quadruplets where the sum of elements at a, b, and c equals the element at d, while maintaining the order a < b < c < d.

This walkthrough demonstrates how the combination of reverse iteration and the use of a Counter can efficiently solve the

Counter to track the frequency of (nums[d] - nums[c]) frequency_counter = Counter() # Iterate from the third last element down to the second element for b_index in range(length - 3, 0, -1):

Java class Solution { public int countQuadruplets(int[] nums) {

```
for (int b = length - 3; b > 0; b--) {
            int c = b + 1;
           // We are calculating the frequency of the difference between nums[d] and nums[c]
            // and storing it in our counter. This will help us to check for quadruplets quickly later.
            for (int d = c + 1; d < length; d++) {</pre>
                int difference = nums[d] - nums[c];
                if (difference >= 0) { // Ensure we don't have a negative index for the counter array
                    ++differenceCounter[difference];
           // Now we check for all 'a' values that come before 'b'
           // For each 'a', if the sum of nums[a] and nums[b] exists as an index in the counter array,
            // it means there exists a 'c' and 'd' such that nums[a] + nums[b] = nums[c] + nums[d].
            // Hence, we add the frequency (number of occurrences) of that sum (difference) to the count of quadruplets.
            for (int a = 0; a < b; ++a) {
                int sum = nums[a] + nums[b];
                count += differenceCounter[sum];
       return count; // Return the total count of quadruplets found
C++
class Solution {
public:
   // Function to count the number of special quadruplets [a, b, c, d] in the given array.
    // A quadruplet is considered special if a + b + c = d.
    int countQuadruplets(vector<int>& nums) {
        int count = 0; // This will hold the final count of special quadruplets.
        int size = nums.size(); // Get the size of the input array.
       vector<int> frequency(310, 0); // Array to store the frequencies of values for the differences of c and d.
       // Start from the second-to-last element and go backwards, as this is the 'b' in the quadruplet.
        for (int b = size - 3; b > 0; --b) {
           // 'c' is always to the right of 'b', so start from 'b + 1'.
            for (int c = b + 1; c < size - 1; ++c) {
                // 'd' is always to the right of 'c', so start from 'c + 1'.
```

```
// Function to count the number of special quadruplets [a, b, c, d] in 'nums' array.
  // A quadruplet is considered special if a + b + c = d.
  function countQuadruplets(): number {
      let count = 0; // This will hold the final count of special quadruplets.
      let size = nums.length; // Get the size of the input array.
      // Start from the second-to-last element and go backwards, as this is the 'b' in the quadruplet.
      for (let b = size - 3; b > 0; --b) {
          // 'c' is always to the right of 'b', so start from 'b + 1'.
          for (let c = b + 1; c < size - 1; ++c) {
              // 'd' is always to the right of 'c', so start from 'c + 1'.
              for (let d = c + 1; d < size; ++d) {</pre>
                  if (nums[d] - nums[c] >= 0) {
                      // Increment the frequency of this particular difference.
                      frequency[nums[d] - nums[c]]++;
          // Now looking for 'a' which is to the left of 'b'.
          for (let a = 0; a < b; ++a) {
              // If a sum of nums[a] and nums[b] happened to be the difference
              // previously recorded, it contributes to the total count.
              count += frequency[nums[a] + nums[b]];
          // Reset the frequency array for the next iteration.
          // This ensures that we only count the differences relevant to the current 'b'.
          frequency.fill(0);
      return count; // Return the final count of special quadruplets.
  // Example usage:
  // nums = [1, 2, 3, 4];
  // let result = countQuadruplets();
  // console.log(result); // Should log the count of special quadruplets
from collections import Counter # Importing Counter from collections module
```

```
Time and Space Complexity
 The given Python code defines a method countQuadruplets which counts the number of quadruples (a, b, c, d) in an array nums
 where a < b < c < d and nums[a] + nums[b] + nums[c] == nums[d].
```

Time Complexity: The outermost loop runs from the third-to-last element to the beginning (n - 3 to 1), which gives us at most n iterations. Inside

this loop, we have two nested loops: 1. The first nested loop iterates over the range from c + 1 to n - 1, where c starts from b + 1. In the worst case, this loop runs for n - b - 2

iterations.

class Solution:

2. The second nested loop runs from 0 to b - 1. In the worst-case scenario, where b is close to n / 2, this loop also contributes a factor of n/2 iterations. The nested loops are not entirely independent: as b decreases, the number of iterations in the first nested loop increases, but the

iterations in the second nested loop decrease proportionally. The time complexity of these nested loops is hence proportional to the sum of the two sequences, which forms an arithmetic progression from 1 to approximately n/2. The sum of the first n/2 positive integers is given by (n/2) * ((n/2) + 1) / 2,

simplifying to 0(n^2). Therefore, the overall time complexity of the function is $0(n^2)$.

Space Complexity: The space complexity is governed by the Counter object, which stores a mapping of the differences between nums [d] and

nums [c]. In the worst case, the Counter could store up to n different values if all the differences are unique. This gives us a space complexity of O(n) for the counter variable.

Thus, the space complexity of the function is O(n).