counting pairs hackerrank solution

Counting pairs hackerrank solution is a popular coding challenge that tests a programmer's ability to efficiently compute the number of pairs of integers in an array that sum to a specific target value. This problem is commonly encountered in coding interviews and competitive programming contests. In this article, we will explore the problem statement, possible approaches to solve it, and present an optimal solution to help you understand the underlying concepts better.

Problem Statement

The basic premise of the "Counting Pairs" problem is as follows:

You are given an array of integers and a target sum. Your task is to count all unique pairs of integers in the array that add up to the target sum. A pair is considered unique if the two numbers are different, even if they appear multiple times in the array.

For example, if you have an array `[1, 2, 3, 4, 3]` and a target sum of `6`, the unique pairs that sum to `6` would be `(2, 4)` and `(3, 3)`. The output should be `2` since these are the two unique pairs.

Understanding the Constraints

Before diving into the solution, it's important to understand the constraints and requirements of the problem:

- The array can contain both positive and negative integers.
- Elements in the array may appear more than once.
- The target sum can also be negative.
- Time complexity and efficiency are crucial, especially with larger arrays.

Given these constraints, a naive solution that checks all possible pairs would be inefficient for larger inputs. Hence, we need to consider more efficient algorithms.

Approaches to Solve the Problem

There are multiple ways to solve the "Counting Pairs" problem. Here are three common approaches:

1. Brute Force Approach

The simplest way to solve the problem is to use a brute force method. This involves iterating through the array with two nested loops to check each pair of elements.

1. Initialize a counter to zero.
2. For each element in the array, check every subsequent element to see if their sum equals the target.
3. If a valid pair is found, increment the counter.

This approach has a time complexity of O(n^2), which is not optimal for larger arrays.

2. Using a Hash Table

A more efficient approach involves using a hash table (or dictionary) to keep track of the numbers we have seen so far. This approach has an average time complexity of O(n).

The steps are as follows:

1. Initialize an empty hash table to store the frequency of each number.
2. Iterate through the array and for each number, calculate its complement (the difference between the target sum and the current number).
3. Check if the complement exists in the hash table. If it does, increment the counter by the frequency of the complement.
4. Update the frequency of the current number in the hash table.

This method is more efficient and can handle larger datasets effectively.

3. Two-Pointer Technique

If the array is sorted, we can use the two-pointer technique to find pairs that sum to the target. This approach also has an O(n) time complexity.

The steps involved are:

1. Sort the array.
2. Initialize two pointers: one at the start of the array and the other at the end.

3. While the left pointer is less than the right pointer:

    - If the sum of the two pointers equals the target, increment the counter and move both pointers inward.
    - If the sum is less than the target, move the left pointer to the right.
    - If the sum is greater than the target, move the right pointer to the left.

This method is also efficient but requires sorting the array first, which can add additional time complexity.

Optimal Solution Implementation

Let's implement the hash table approach since it is the most straightforward and efficient for unsorted arrays. Below is a sample Python implementation of the counting pairs problem:

```python
def count_pairs(arr, target):
Create a dictionary to store the frequency of each number
freq = {}
count = 0

Iterate through the array
for number in arr:
Calculate the complement
complement = target - number

If the complement exists in the frequency table, add its count to the result
if complement in freq:
count += freq[complement]

Update the frequency of the current number
if number in freq:
freq[number] += 1
else:
freq[number] = 1

return count

Example usage
arr = [1, 2, 3, 4, 3]
target = 6
print(count_pairs(arr, target)) Output: 2
```

Conclusion

In conclusion, the "Counting Pairs" problem is a classic example of how to effectively count pairs in an array that sum to a specific target. By utilizing efficient data structures such as hash tables, we can significantly reduce the time complexity of our solution. The brute force method, while simple, is not feasible for large datasets due to its O(n^2) performance.

Understanding the various approaches and their complexities not only prepares you for solving similar problems in coding interviews but also enhances your problem-solving skills overall. As you practice more, you will become adept at recognizing which method to apply based on the constraints of the problem at hand. Whether you are tackling the "Counting Pairs" problem or any other coding challenge, remember to think critically about efficiency and clarity in your solutions.

Frequently Asked Questions

What is the 'Counting Pairs' problem on HackerRank?

The 'Counting Pairs' problem on HackerRank involves finding the number of pairs (i, j) in a given array such that a specific condition (like a difference or sum) is met, often requiring efficient algorithms due to potentially large input sizes.

What is a common approach to solve the 'Counting Pairs' problem efficiently?

A common approach to efficiently solve the 'Counting Pairs' problem is to use a hash map (or dictionary) to store the frequency of elements and then iterate through the array to count valid pairs based on the required condition.

Can you explain the two-pointer technique in the context of 'Counting Pairs'?

The two-pointer technique involves sorting the array and using two indices to traverse it, adjusting the pointers based on the difference between the elements to efficiently count pairs that meet specific criteria.

What are the time complexity considerations for the 'Counting Pairs' problem?

The time complexity for a naive solution could be O(n^2), but using hashing or the two-pointer technique can reduce it to O(n log n) due to sorting or O(n) for linear scans, making the solution feasible for larger datasets.

How can edge cases affect the solution for 'Counting Pairs'?

Edge cases, such as duplicate elements, empty arrays, or arrays with all

identical values, can significantly impact the correctness and efficiency of the solution, making it essential to account for these scenarios in the implementation.

Counting Pairs Hackerrank Solution