2364. Count Number of Bad Pairs

Medium Array Hash Table

Leetcode Link

Problem Description

In this problem, you are provided with an integer array called `nums`, where the array is indexed starting from `0`. You must identify all pairs of indices `(i, j)` that qualify as a "bad pair". A pair is designated as a bad pair if it satisfies two conditions:

1. `i < j` (the index `i` comes before index `j`)
2. `j - i` is not equal to `nums[j] - nums[i]`

This means that the relative difference between the indices should equal the relative difference between their respective elements for it to be a good pair; otherwise, it is considered bad.

Your task is to calculate and return the total count of bad pairs in the `nums` array.

Intuition

The brute-force approach would be to check all possible pairs `(i, j)` where `i < j` and count the number of pairs where `j - i` does not equal `nums[j] - nums[i]`. However, this approach has a time complexity of O(n^2) which would not be efficient for large arrays.

To optimize this, we can make a key observation: instead of focusing on whether a pair is bad, concentrate on finding the good pairs because they adhere to a specific pattern, namely `j - i = nums[j] - nums[i]`. This can be rearranged to `j - nums[j] = i - nums[i]`. If this property holds true for any two indices, they form a good pair. If it doesn't, they form a bad pair.

The intuition for the solution is to transform the problem such that we keep track of how many times we have seen the same difference (`i - nums[i]`) before. When we process the array, at each index `i`, we calculate this difference and check it against a count tracker. For a new difference value at index `i`, all previous occurrences of this value (the count tracked so far) represent the number of good pairs with the new index `i`. Hence, bad pairs can be determined by subtracting the number of good pairs at each step from the total number of pairs seen so far.

The Counter dictionary in the solution keeps track of how many times we have seen each difference. For each new element `x` at index `i`, we increment the counter for `i - x` which indicates we've seen another instance where this difference occurs. We then update the answer by adding `i` (the total number of pairs with the current index so far) minus the count of this difference (the number of good pairs). The final answer reflects the total number of bad pairs.

This solution provides an efficient way to calculate the number of bad pairs with a time complexity of O(n), which is a significant improvement over the brute-force method.

Solution Approach

The implementation uses a hash map to track the counts of the differences. In Python, this is conveniently handled using the `Counter` class from the collections module. The `Counter` works like a standard dictionary but is specialized for counting objects. It has methods that make it easy to increment counts for each object.

The key step in the algorithm is to iterate over the array `nums` while using the `enumerate` function that provides both the index `i` and the value `x`. The difference `i - x` is the transformed value we are interested in. Here's how the algorithm proceeds:

1. Initialize a `Counter` object `cnt` that will track the counts of differences.
2. Initialize `ans` to 0, which will keep track of the number of bad pairs.
3. Loop through the `nums` array using an `enumerate` loop to get both the index `i` and the value `x` at that index.
4. For each element, the difference `i - x` is calculated, which will be used to determine if a pair is good.
   - `ans` is incremented by `i - cnt[i - x]`, which represents the number of bad pairs up to the current index `i`. This is calculated by subtracting the number of good pairs (tracked by `cnt[i - x]`) from the total number of pairs that could have been formed with all the previous elements (given by `i`).
5. The count for the current difference `i - x` is incremented in `cnt` to record that we have seen another instance where this difference occurs.
6. After the loop ends, `ans` contains the total count of bad pairs, which is returned.

The reason why this approach works is due to the transformation of the problem into a more manageable counting problem. By tracking the occurrence of the differences, we avoid directly comparing every possible pair, which results in a much more efficient algorithm.

The key components of the implementation are:

- **Enumeration**: Providing both the index and the value to easily calculate our necessary difference.
- **Hash Map (`Counter`)**: For efficient storage and retrieval of counts associated with each difference. This allows us to efficiently keep track of the number of good pairs per difference.
- **Transformation**: Condensing the condition from `j - i != nums[j] - nums[i]` to `i - nums[i]`, making it easier to track and count through a hash map.

By utilizing these components, we achieve a linear time solution, which is optimal for this problem.

Example Walkthrough

Let's illustrate the solution approach using a small example. Consider the `nums` array as `[1, 2, 3, 5]`. We want to find all bad pairs where `i < j` and `j - i` is not equal to `nums[j] - nums[i]`.

First, we initialize a `Counter` object `cnt` and a variable `ans` to count the bad pairs, starting with `ans` set to 0.

Now, we start iterating through the array with indices and values:

- At `i = 0`, `x = 1`, the difference `i - x` is `-1`. This difference hasn't been seen before, so `cnt[-1]` is 0. We update `ans` with `0 - 0 = 0`, and increment `cnt[-1]`.
- At `i = 1`, `x = 2`, the difference `i - x` is `-1` again. Now, `cnt[-1]` is 1 (from the previous count). We update `ans` with `1 - 1 = 0`, but we already have 0 bad pairs. Thus, `ans` stays as 0, and we increment `cnt[-1]` to 2.
- At `i = 2`, `x = 3`, the difference `i - x` is `-1` yet again. `cnt[-1]` is now 2. We update `ans` with `2 - 2 = 0`, and the number of bad pairs `ans` stays at `0`. We increment `cnt[-1]` to 3.
- At `i = 3`, `x = 5`, the difference `i - x` is `-2`. This is a new difference, so `cnt[-2]` is 0. We update `ans` with `3 - 0 = 3`, indicating that up to this index, we have 3 bad pairs. We increment `cnt[-2]`.

By the end of the loop, we have counted `ans = 3` bad pairs. To confirm, we can check the conditions for each pair:

- `(0, 1): 1 - 0 = 2 - 1` (Good pair)
- `(0, 2): 2 - 0 = 3 - 1` (Good pair)
- `(0, 3): 3 - 0 ≠ 5 - 1` (Bad pair)
- `(1, 2): 2 - 1 = 3 - 2` (Good pair)
- `(1, 3): 3 - 1 ≠ 5 - 2` (Bad pair)
- `(2, 3): 3 - 2 ≠ 5 - 3` (Bad pair)

There are indeed 3 bad pairs: `(0, 3)`, `(1, 3)`, `(2, 3)`.

The counter at the end of processing will be `{ -1: 3, -2: 1 }`, reflecting the idea that negative counts corresponded to instances where a good pair did not occur, leading to bad pairs.

This example shows how the algorithm can effectively identify bad pairs by focusing on counting the occurrences of the `i - nums[i]` and using that to quickly determine the number of bad pairs without comparing each pair directly.

Python Solution

```python
from collections import Counter

class Solution:
    def countBadPairs(self, nums: List[int]) -> int:
        # Initialize a counter to keep track of occurrences of differences
        difference_counter = Counter()

        # Initialize the answer which will store the count of bad pairs
        bad_pairs_count = 0

        # Enumerate over nums to get index (i) and number (x)
        for index, number in enumerate(nums):
            # A pair (j, i) is bad if j < i and j - i != nums[i] - nums[j]
            # which can be rewritten as i - nums[j] != j - nums[j]
            # Thus, we count how many indexes do not have the same (index - number) as the current one
            # The difference (index - number) serves as a key in the counter.
            bad_pairs_count += index - difference_counter[index - number]

            # Increment the count for this difference in the counter
            difference_counter[index - number] += 1

        # Return the total count of bad pairs
        return bad_pairs_count
```

Please note that you will need to import `List` from `typing` module at the beginning of the code in order to type hint the `nums` parameter as `List[int]`. You should add the following line at the beginning with the other imports:

```python
from typing import List
```

Java Solution

```java
class Solution {

    // The function 'countBadPairs' returns the number of 'bad pairs' in the array.
    // A 'bad pair' (j, k) is defined as one where j < k and nums[j] - nums[k] != j - k.
    public long countBadPairs(int[] nums) {
        // This map is used to track the occurrence frequency of calculated values.
        Map<Integer, Integer> countMap = new HashMap<>();
        // Initialize 'badPairsCount' which will store the final count of bad pairs.
        long badPairsCount = 0;

        // Iterate through the array 'nums'.
        for (int i = 0; i < nums.length; ++i) {
            // Calculate 'difference' as the difference between the current index and the current element.
            int difference = i - nums[i];

            // Increment 'badPairsCount' by the number of indices processed minus the number of occurrences of 'difference.'
            // If 'difference' has not been seen, 'getOrDefault' will return 0, meaning no good pairs have been skipped.
            badPairsCount += i - countMap.getOrDefault(difference, 0);

            // Update the count of 'difference' in the map for future reference.
            // The 'merge' method is used to add 1 to the current count of 'difference' or to set it to 1 if not present.
            countMap.merge(difference, 1, Integer::sum);
        }

        return badPairsCount; // Return the total count of bad pairs.
    }
}
```

C++ Solution

```cpp
#include <unordered_map>
#include <vector>

class Solution {
public:
    // Function to count the number of 'bad' pairs in the array
    // A pair (i, j) is considered 'bad' if i < j and j - i != nums[j] - nums[i]
    long long countBadPairs(std::vector<int>& nums) {
        std::unordered_map<int, int> count_map; // Map to store the frequency of difference values
        long long bad_pairs_count = 0; // Initialize the count of bad pairs to 0

        // Iterate over the elements in nums
        for (int i = 0; i < nums.size(); ++i) {
            int diff = i - nums[i]; // Calculate the difference between index and value

            // The total number of pairs with elements before index i is i
            // Subtract the count of the number of 'good' pairs (where the diff is the same)
            bad_pairs_count += i - count_map[diff];

            // Increment the count of the current difference in the map
            ++count_map[diff];
        }

        // Return the final count of bad pairs
        return bad_pairs_count;
    }
};
```

Typescript Solution

```typescript
// Function to count the number of bad pairs in an array.
// Pairs (i, j) are considered bad if i < j and j - i != nums[j] - nums[i].
function countBadPairs(nums: number[]): number {
    // Map to keep track of the number of occurrences of the difference between index and value.
    const countMap = new Map<number, number>();
    // Initialize the count of bad pairs.
    let badPairsCount = 0;

    // Iterate through the array elements.
    for (let i = 0; i < nums.length; ++i) {
        // Calculate the difference between the current index and the value at that index.
        const diff = i - nums[i];
        // The number of good pairs so far with the same difference is subtracted from the total pairs so far.
        // This gives the number of bad pairs involving the current element.
        badPairsCount += i - (countMap.get(diff) ?? 0);
        // Update the count of good pairs with the current difference in the map.
        countMap.set(diff, (countMap.get(diff) ?? 0) + 1);
    }

    // Return the total count of bad pairs.
    return badPairsCount;
}
```

Time and Space Complexity

The time complexity of the given code is `O(n)`, where `n` is the length of the `nums` array. The reason for this is that the code uses a single loop that iterates through all elements of `nums`, performing a constant amount of work within each iteration. The `cnt` Counter dictionary lookup and update operations are `O(1)` on average, so they don't change the overall linear time complexity.

The space complexity of the code is also `O(n)` because the `cnt` Counter dictionary can potentially grow to have as many distinct keys as there are elements in `nums`. In the worst case, each iteration introduces a new key-value pair into the `cnt` dictionary.