2615. Sum of Distances

Medium | Array | Hash Table | Prefix Sum

Leetcode Link

Problem Description

In this problem, we're presented with an integer array `nums` which is 0-indexed. We're tasked to find another array `arr` where each element `arr[i]` represents the sum of absolute differences between index `i` and every index `j` where the value `nums[j]` is equal to `nums[i]`, excluding when `i` and `j` are the same. If there are no such `j` values (meaning there are no other identical values to `nums[i]` in the array), we simply set `arr[i]` to `0`.

To illustrate, if `nums` was `[3, 3, 1]`, then `arr` would be `[1, 1, 0]` because:

- For `i=0`, `nums[0] == 3`, we find another `3` at index `1`, so the sum is `|0 - 1| = 1`. There are no more `3`s, so `arr[0]` is `1`.
- For `i=1`, it's the same situation but in reverse, so `arr[1]` is also `1`.
- For `i=2`, `nums[2] == 1` and there are no other `1`s in the array, so `arr[2]` is `0`.

We need to perform this calculation efficiently for each index and construct the `arr` array accordingly.

Intuition

To find an effective solution, we should try to avoid the naive approach of checking each pair of indices for equality, as that would result in an O(n^2) time complexity, which is impractical for large arrays.

Instead, we can optimize by grouping indices with the same values together. We can use a dictionary to map each unique number in the array to the list of indices where that number appears. This will help us compute the sum of distances for each group in aggregate, rather than individually.

Once we have those index lists, we can iterate over them to calculate the sum of distances for each index in the group. Here's the step-by-step approach:

1. We compute the sum of indices for each group - this gives us a starting sum for the right part of each index.
2. We start from the leftmost index in each group and iterate towards the right.
3. For each index, we maintain two variables: `left` and `right`, which represent the sum of distances to indices on the left and right, respectively.
4. As we move from one index to the next within a group, we update `left` by adding the distance to the next index multiplied by the count of indices that are now on the left.
5. We update `right` by subtracting the same distance multiplied by the count of indices remaining on the right.
6. We use these `left` and `right` accumulators to calculate the total distance for each index, this total distance is what we add to the corresponding position in the `arr` array.

This approach reduces the amount of redundant calculations and brings the complexity down significantly because each distance is essentially calculated in linear time relative to the number of occurrences of each unique value.

Solution Approach

The solution utilizes a hash map (via Python's `defaultdict` from the `collections` module) to group the indices of identical values in `nums`. Here's how the solution is implemented:

1. Initialize a `defaultdict` of lists named `d`. Iterate over `nums` using `enumerate` to build the dictionary where each key is a value in `nums`, and the corresponding value is a list of indices where that key appears.

2. Create an array `ans` filled with zeros of the same length as `nums`. This array will store our resulting distances for each index.

3. For each list of indices `idx` in the hash map:

   - Initialize `left` as `0` because there are no indices to the left of our starting point.
   - Calculate the initial value of `right` by summing up all the indices in `idx` and adjusting for the lowest index (`len(idx) * idx[0]`). This variable represents the sum of distances of all indices to the right of our current index.

4. Iterate over each index `i` in `idx`:

   - Set `ans[idx[i]]` to be the sum of `left` plus `right`, which provides the sum of distances for `nums[idx[i]]`.
   - Before we move to the next index `i+1`, we update `left` and `right`. Update `left` by adding the distance to the next index multiplied by `i + 1` (because we'll have one more element to the left).
   - Update `right` by subtracting the same distance multiplied by `(len(idx) - i - 1)` to account for having one less element to the right.

5. Loop until all indices in `idx` are processed. By the end, `ans` will have the calculated distances for each element in `nums`.

This approach is efficient because it computes the distances for indices of the same value in `nums` as a whole, minimizing repeated work. By using `left` and `right` accumulators, it avoids iterating over all pairs of indices and scales linearly with the number of occurrences of each unique value in `nums`.

The algorithmic patterns used here include hashing (to group indices), enumeration (to iterate with indices), and prefix sums (though not explicitly stored, they are implied in the use of `left` and `right` to keep track of cumulative distances).

Example Walkthrough

For illustration, let's use `nums = [2, 1, 2, 1, 2]` to demonstrate the solution approach.

1. We first create the hash map with the list of indices for each value in `nums`:

   For value `2`, the list of indices is `[0, 2, 4]`. For value `1`, the list of indices is `[1, 3]`.

   Our hash map (`d`) now looks like this:

   ```
   {
       2: [0, 2, 4],
       1: [1, 3]
   }
   ```

2. We then initialize the `ans` array to `[0, 0, 0, 0, 0]`, corresponding to the five elements in `nums`.

3. We begin processing the list of indices for the value `2`:

   - **Initial setup for value 2:** `left = 0`, `right` is calculated by `(2 + 4) - (2 * 0) = 6`.

   - **First index (i=0):**

     - We set `ans[0] = left + right` → `ans[0] = 0 + 6 = 6`.
     - We update `left` to be `0 + (1 * 2)` since we're moving one step to the right, adding two spaces (distance) from the first index.
     - We update `right` by subtracting `2 * (3 - 1 - 0)` to account for the two spaces we're moving away from the remaining two indices (2 and 4).
   - **Second index (i=1):**

     - Now `left = 2` and `right = 2` (from previous calculations).
     - Thus, `ans[2] = left + right` → `ans[2] = 2 + 2 = 4`.
     - Update `left` to be `2 + (2 * 2)` as we're now considering the total distance from the first and second index to the third.
     - Update `right` to be `2 - (2 * (2 - 1 - 1))`, no change because only one index remains to the right, and we're moving one step away from it.
   - **Third index (i=2):**

     - Now `left = 6` and `right = 0` (there are no more indices to the right).
     - So, `ans[4] = left + right` → `ans[4] = 6 + 0 = 6`.

   With value `2` processed, `ans` becomes `[6, 0, 4, 0, 6]`.

4. Next, proceed with the indices for value `1`:

   - **Initial setup for value 1:** `left = 0`, `right` is calculated by `(3) - (1 * 1) = 2`.

   - **First index (i=0):**

     - `ans[1] = left + right` → `ans[1] = 0 + 2 = 2`.
     - Update `left` to be `0 + (1 * 1)`,
     - Update `right` to be `2 - (1 * (2 - 1 - 0))`, which makes `right = 1` now.
   - **Second index (i=1):**

     - `left = 1` and `right = 0`.
     - `ans[3] = left + right` → `ans[3] = 1 + 0 = 1`.

   Now, our answer array `ans` is completely updated to `[6, 2, 4, 1, 6]`.

After processing both lists of indices, we get `arr = [6, 2, 4, 1, 6]` corresponding to the sum of absolute differences for each index in `nums`, as intended by the solution approach.

Python Solution

```python
from collections import defaultdict
from typing import List

class Solution:
    def distance(self, nums: List[int]) -> List[int]:
        # Create a dictionary to store indices of each number in nums
        indices_dict = defaultdict(list)
        for index, num in enumerate(nums):
            indices_dict[num].append(index)

        # Initialize the answer list with zeros
        distances = [0] * len(nums)

        # Iterate over the groups of indices for each unique number
        for indices in indices_dict.values():
            # Initialize the distance from the left and right side
            left_distance = 0
            right_distance = sum(indices) - len(indices) * indices[0]

            # Iterate through the indices to compute distances
            for idx_position in range(len(indices)):
                # The total distance is the sum of left and right distances
                distances[indices[idx_position]] = left_distance + right_distance

                # Update the left and right distances for the next index (if not at the end)
                if idx_position + 1 < len(indices):
                    gap = indices[idx_position + 1] - indices[idx_position]
                    left_distance += gap * (idx_position + 1)
                    right_distance -= gap * (len(indices) - idx_position - 1)

        # Return the list of distances
        return distances

```

Java Solution

```java
class Solution {
    public long[] distance(int[] nums) {
        int n = nums.length; // Get the length of the input array.
        long[] answer = new long[n]; // Initialize the answer array to store the final distances.
        Map<Integer, List<Integer>> indexMap = new HashMap<>(); // Map to store the indices of each number in nums.

        // Loop through nums to fill the indexMap.
        for (int i = 0; i < n; ++i) {
            // If the key doesn't exist, create a new list, then add the index.
            indexMap.computeIfAbsent(nums[i], k -> new ArrayList<>()).add(i);
        }

        // Iterate over the entry set of the map to calculate distances.
        for (List<Integer> indices : indexMap.values()) {
            int m = indices.size(); // The frequency of the current number.
            long leftSum = 0; // Sum to store the left side distances.
            long rightSum = -1L * m * indices.get(0); // Sum to store the right side distances, initially inverted.

            // Initial rightSum calculation by summing all indices.
            for (int index : indices) {
                rightSum += index;
            }

            // Iterate over each index for the current number.
            for (int i = 0; i < m; ++i) {
                // Calculate the final distance for the current index.
                int currentIndex = indices.get(i);
                answer[currentIndex] = leftSum + rightSum;

                // Recalculate leftSum and rightSum for the next index if it exists.
                if (i + 1 < m) {
                    int nextIndex = indices.get(i + 1);
                    long diff = nextIndex - currentIndex; // Difference between the current and the next index.
                    leftSum += diff * (i + 1L);
                    rightSum -= diff * (m - i - 1L);
                }
            }
        }

        return answer; // Return the completed answer array.
    }
}

```

C++ Solution

```cpp
class Solution {
public:
    // Function to calculate distances
    vector<long long> distance(vector<int>& nums) {
        // n holds the length of the input array nums
        int n = nums.size();
        // Create a vector to store the final answer
        vector<long long> ans(n);
        // Use a map to store indices of elements in nums
        unordered_map<int, vector<int>> indexMap;

        // Populate the map with the indices of each element
        for (int i = 0; i < n; ++i) {
            indexMap[nums[i]].push_back(i);
        }

        // Iterate over each element of the map
        for (auto& [value, indices] : indexMap) {
            // m holds the number of occurrences of a particular value in nums
            int m = indices.size();
            // left is used to keep track of the cumulative distance from the left
            long long left = 0;
            // right is used to keep track of the cumulative distance from the right
            long long right = -1LL * m * indices[0];
            // Calculate the initial right value by summing up all indices
            for (int index : indices) {
                right += index;
            }
            // Iterate over all occurrences of the current value
            for (int i = 0; i < m; ++i) {
                // Store the total distance at the current index
                ans[indices[i]] = left + right;

                // Update left and right for the next values if not at the end
                if (i + 1 < m) {
                    left += (indices[i + 1] - indices[i]) * (i + 1);
                    right -= (indices[i + 1] - indices[i]) * (m - i - 1);
                }
            }
        }
        // Return the final vector of distances
        return ans;
    }
};

```

Typescript Solution

```typescript
// Function to calculate distances
function distance(nums: number[]): number[] {
    // n holds the length of the input array nums
    let n: number = nums.length;
    // Create an array to store the final answer
    let ans: number[] = new Array(n);
    // Use a map to store indices of elements in nums
    let indexMap: Map<number, number[]> = new Map<number, number[]>();

    // Populate the map with the indices of each element
    for (let i = 0; i < n; ++i) {
        if (!indexMap.has(nums[i])) {
            indexMap.set(nums[i], []);
        }
        indexMap.get(nums[i])!.push(i);
    }

    // Iterate over each element of the map
    indexMap.forEach((indices: number[], value: number) => {
        // m holds the number of occurrences of a particular value in nums
        let m: number = indices.length;
        // left is used to keep track of the cumulative distance from the left
        let left: number = 0;
        // right is used to keep track of the cumulative distance from the right
        let right: number = -1 * m * indices[0];
        // Calculate the initial right value by summing up all indices
        for (let index of indices) {
            right += index;
        }
        // Iterate over all occurrences of the current value
        for (let i = 0; i < m; ++i) {
            // Store the total distance at the current index
            ans[indices[i]] = left + right;

            // Update left and right for the next values if not at the end
            if (i + 1 < m) {
                left += (indices[i + 1] - indices[i]) * (i + 1);
                right -= (indices[i + 1] - indices[i]) * (m - i - 1);
            }
        }
    });

    // Return the final array of distances
    return ans;
}

```

Time and Space Complexity

The above code maintains a dictionary to store indices of each unique element in the list `nums` and then computes the distance sum for each unique element.

Time Complexity:

- The first for loop iterates over all `n` elements of the list `nums` to create the dictionary `d`, which takes `O(n)` time.
- The second loop processes each group of indices stored in the dictionary `d`. Let `k` be the maximum size of these index groups. For each group, it iterates over all elements in the list of indices which, in the worst case, could take `O(k)`. Since there are as many groups as there are unique elements in `nums`, we can say that it takes `O(u * k)` time, where `u` is the number of unique elements.
- Inside this loop, it performs constant-time operations like addition and subtraction for updating left and right variables.

Since `k` times the number of unique elements `u` cannot exceed `n` (as each element is part of one group and is visited once), the second loop's time complexity is bounded by `O(n)`.

Therefore, the total time complexity is `O(n)` + `O(n)` = `O(n)`.

Space Complexity:

- The space taken by the dictionary `d` depends on the number of unique elements in `nums`. If there are `u` unique elements, it could take up to `O(u * k)` space, where `k` is the average size of the index lists for each unique element. However, since there are `n` total elements, `u * k` is also bounded by `n`. Therefore, it uses `O(n)` space.
- We also have the `ans` array, which is equal in size to `nums`, adding another `O(n)` space.

Adding these together, the space complexity of the code is `O(n)` + `O(n)` = `O(n)`.