2121. Intervals Between Identical Elements

Medium Array Hash Table Prefix Sum

Leetcode Link

Problem Description

The problem provides us with an array `arr` where each element may occur more than once. For each element `arr[i]`, we want to calculate the sum of distances (intervals) between `arr[i]` and all other elements in `arr` that have the same value as `arr[i]`. In other words, for every element that matches `arr[i]`, we sum up the absolute differences in their indices. The output should be an array `intervals` of the same length as `arr` where each `intervals[i]` contains the computed sum for `arr[i]`.

To put it simply, if an element appears multiple times in the array, we are looking to find out how far apart all of those occurrences are from each other, summing up these distances for each unique value.

Intuition

The solution approach makes use of a dictionary (or equivalent) to store occurrences of each unique value in the given array. Keys in this dictionary are the unique values, and the values are lists containing the indices where the key occurs.

The overarching idea is to use these lists to calculate the intervals in an efficient manner. A neat observation is that when calculating the sum of intervals for a particular number in `arr`, we notice that if the indices are sorted, each index `v[i]` will contribute `(i + v[i]) - (n - i + v[i])` to the total sum, where `i` is the current position within the occurrences of the number in `arr`, and `n` is the total number of occurrences. Put differently, for any given occurrence, the distances from it to all previous occurrences will each increase by `v[i] - v[i - 1]` when moving to the next occurrence, and the distances to all next occurrences will each decrease by the same amount.

The algorithm's efficiency comes from leveraging this pattern to update the sum iteratively while traversing through the list of occurrences of each unique value, thereby avoiding the need to recompute the sum from scratch for every element.

Here is a step-by-step breakdown of the solution:

1. Create a dictionary (`d`) to group elements of `arr` by their values, mapping each value to a list of indices at which it occurs in `arr`.
2. Initialize an answer array (`ans`) of size `n` with zeros to hold the sum of intervals for each corresponding element in `arr`.
3. Iterate through the grouped values in the dictionary:
    - For the current group, calculate the initial interval sum for the first occurrence.
    - Iterate through the indices in the current group:
        - Calculate the difference between the current index and the previous one.
        - Update the running total (`val`), adding the interval increments for indices before the current one and subtracting for indices after.
        - Store the running total in the corresponding position in the `ans` array.
4. Return the `ans` array, which now contains the sum of intervals for each element in `arr`.

By organizing the elements in this way and cleverly calculating the contributions of each element's occurrences to the sum, the solution achieves efficient computation of the intervals.

Solution Approach

The solution approach can be broken down into the following steps:

1. **Group Elements by Value**: The `defaultdict(list)` is used to create a dictionary `d` where each key is a value in the array `arr`, and the corresponding value is a list of indices at which this key occurs. The `enumerate` function is used to get both the index `i` and the value `v` as we iterate through `arr`. This step essentially groups the occurrences of each element for easy access.

```
d = defaultdict(list)
for i, v in enumerate(arr):
    d[v].append(i)
```

2. **Prepare the Answer Array**: An array `ans` of size `n` (where `n` is the length of the input array `arr`) is initialized with zeros. This will be used to store the computed sum of intervals for each element within `arr`.

```
n = len(arr)
ans = [0] * n
```

3. **Iterate Over the Groups**: Then, we iterate through all lists of indices (all values) present in `d`. For each list `v`, we calculate the sum of intervals for the first element in the list as `val`. Here, `v` refers to the total number of occurrences in the list `v`.

```
for v in d.values():
    m = len(v)
    val = sum(v - v[0] + m
```

4. **Compute the Intervals Using a Running Total**: As we iterate over the indices in `v`:

    - We incrementally update `val` by adding the cumulative distance going forward and subtracting the cumulative distance going backwards in the list of occurrences.
    - The `delta` represents the difference between the current index and the previous one, which affects the cumulative distances in this manner.
    - The final sum for each occurrence `i` of the value is then saved in `ans[i]`.

```
for i, p in enumerate(v):
    delta = v[i] - v[i - 1] if i >= 1 else 0
    val += i * delta - (m - i) * delta
    ans[p] = val
```

5. **Return the Result**: The `ans` array is returned, which holds the sum of intervals for each element of the original array `arr`.

The solution approach combines dictionary mapping with efficient handling of sequential updates, thereby avoiding redundant calculations. This reduces the complexity considerably compared to a naive approach that might involve nested loops. The use of a running total (incremental adjustment of the `val`) to calculate the distances as we iterate through each list of occurrences is key to the efficiency of the solution.

Example Walkthrough

Let's consider a small example to illustrate the solution approach. Suppose we have the following array `arr`:

```
arr = [1, 3, 1, 3, 2, 1]
```

First, we want to group the elements by value. We will get a dictionary `d` that looks like this:

```
1 d = {
2   1: [0, 2, 5],
3   3: [1, 3],
4   2: [4]
5 }
```

Here, the keys of the dictionary are the unique values from `arr`, and the values are lists containing the indices at which the corresponding key occurs in `arr`.

The answer array `ans` will initially be all zeros:

```
ans = [0, 0, 0, 0, 0, 0]  # since there are 6 elements in arr
```

We iterate over the entries in `d`:

- For the group with key `1`, v is `[0, 2, 5]`. The length of `v` (`m`) is 3. The initial interval sum `val` is calculated as `val = sum([0, 2, 5]) - (0 * 3) = 7`.
- Within the group:
    - The difference (`delta`) between the first and second occurrence of `1` is `2 - 0 = 2`. Update `val` to `val + (1 * 2) - (2 * 2) = 7 + 2 - 4 = 5`. `ans[2]` is set to `5`.
    - The difference between the second and third occurrence is `5 - 2 = 3`. Update `val` to `val + (2 * 3) - (1 * 3) = 5 + 6 - 3 = 8`. `ans[5]` is set to `8`.
- Moving on to the group with key `3`, v is `[1, 3]`. Here, `m` is 2 and initial `val` is `val = sum([1, 3]) - (1 * 2) = 4 - 2 = 2`.
- Within the group:
    - The difference (`delta`) between the first and second occurrence of `3` is `3 - 1 = 2`. Update `val` to `val + (1 * 2) - (1 * 2) = 2 + 0 - 2 = 2`. `ans[1]` and `ans[3]` are both set to `2` since this is the total distance for both occurrences.
- Finally, for the group with key `2`, v is `[4]`. Because there is only one occurrence, the distance is zero. `ans[4]` remains `0`.

The final `ans` array will be:

```
ans = [8, 2, 5, 2, 0, 8]
```

In this final array, each `ans[i]` is the sum of intervals between the occurrences of `arr[i]` in the original array. This efficiently solves the problem by using a dictionary to map values to indices and by calculating the sum incrementally.

The solution leverages the spatial ordering of index occurrences, making use of a running total instead of performing redundant calculations, which would happen if doing this for each element individually without considering the previously calculated distances.

Python Solution

```
from collections import defaultdict

class Solution:
    def getDistances(self, arr: List[int]) -> List[int]:
        # Dictionary to store the indices of each unique value in the array
        index_map = defaultdict(list)
        # Length of the array
        n = len(arr)

        # Populate the dictionary with the indices for each value
        for index, value in enumerate(arr):
            index_map[value].append(index)

        # Initialize the answer list with zeros
        answer = [0] * n

        # Loop through the dictionary to calculate the distances for each value
        for indices in index_map.values():
            # Calculate the initial total distance for the first occurrence
            total_distance = sum(indices) - indices[0] * n

            # Calculate the distances for all other occurrences of the same value
            for i, current_index in enumerate(indices):
                # Calculate the change in distance when moving from one occurrence to the next
                delta = indices[i] - indices[i - 1] if i >= 1 else 0
                # Update the total distance considering the number of elements to the left and right
                total_distance += i * delta - (len(indices) - i) * delta
                # Store the total distance at the current index in the answer list
                answer[current_index] = total_distance

        return answer
```

Java Solution

```
class Solution {
    public long[] getDistances(int[] arr) {
        // A map to store the indices of each unique value in the array
        Map<Integer, List<Integer>> indexMap = new HashMap<>();
        int length = arr.length;

        // Populate the map with indices for each value
        for (int i = 0; i < length; ++i) {
            indexMap.computeIfAbsent(arr[i], k -> new ArrayList<>()).add(i);
        }

        // Array to store the result
        long[] result = new long[length];

        // Iterate over each list of indices for all unique values
        for (List<Integer> indices : indexMap.values()) {
            int size = indices.size();
            long sumIndices = 0;

            // Calculate the sum of all indices for the current value
            for (int index : indices) {
                sumIndices += index;
            }

            // Initialize the sum for the first location of this value
            sumIndices -= (size * indices.get(0));

            // Calculate the total distance for each index occurrence
            for (int i = 0; i < size; ++i) {
                // Difference between the current and previous index
                int diff = i > 0 ? indices.get(i) - indices.get(i - 1) : 0;
                // Update sumIndices to include distances of current index from others
                sumIndices += i * diff - (size - i) * diff;
                // Set the total distance for the current index in the result array
                result[indices.get(i)] = sumIndices;
            }
        }

        return result;
    }
}
```

C++ Solution

```
class Solution {
public:
    vector<long long> getDistances(vector<int>& arr) {
        unordered_map<int, vector<size_t>> indexMap; // Maps each unique value to the indices it appears at

        size_t arr_size(arr.size()); // Size of the input array
        // Index each element's value in the map
        for (int i = 0; i < arr_size; ++i) {
            indexMap[arr[i]].push_back(i);
        }

        vector<long long> answer(arr_size); // Initialize the answer vector with the size of the input array
        // Iterate over each element in the map to compute the sum of distances
        for (auto& item : indexMap) {
            auto& indices = item.second; // References the vector of indices for this element value
            size_t count = indices.size(); // Number of occurrences of this element value
            long long sum = 0;

            // Calculate the initial sum of distances from the first occurrence
            for (size_t idx = 1; idx < count; ++idx) {
                sum += idx;
            }
            sum -= count * indices[0]; // Subtract distances as though all elements were at the first occurrence

            // Loop over indices to distribute the sum to the appropriate position in the answer vector
            for (int i = 0; i < indices.size(); ++i) {
                // Calculate the difference between the positions of the current and previous occurrences
                int diff = i > 0 ? indices[i] - indices[i - 1] : 0;
                // Update the sum: add the difference for the previous occurrences,
                // and subtract it for the remaining occurrences
                sum += i * diff - (count - i) * diff;
                // Assign the sum to the index of the current occurrence in the answer vector
                answer[indices[i]] = sum;
            }
        }

        return answer; // Return the computed sum of distances vector
    }
};
```

Typescript Solution

```
// Define the type for mapping each unique value to the indices it appears at
let indexMap: Record<number, number[]> = {};

// Define the method signature with the appropriate TypeScript types
function getDistances(arr: number[]): bigint[] {
    indexMap = {}; // Reset the indexMap for each call
    const size: number = arr.length; // Size of the input array

    // Index each element in the map
    for (let i = 0; i < size; ++i) {
        if (!indexMap[arr[i]]) {
            indexMap[arr[i]] = [];
        }
        indexMap[arr[i]].push_back(i);
    }

    // Initialize the answer array with the size of the input array,
    // filled with BigInts representing zero for each element
    let answer: bigint[] = new Array<bigint>(size).fill(BigInt(0));

    // Iterate over each element in the map to compute the sum of distances
    for (let itemKey in indexMap) {
        let indices: number[] = indexMap[itemKey]; // References the array of indices for this element value
        let count: number = indices.length; // Number of occurrences of this element value
        let sum: bigint = BigInt(0);

        // Calculate the initial sum of distances from the first occurrence
        for (let idx = 1; idx < count; ++idx) {
            sum += BigInt(idx);
        }
        sum -= BigInt(count * indices[0]); // Subtract distances as though all elements were at the first occurrence

        // Loop over indices to distribute the sum to the appropriate position in the answer array
        for (let i = 0; i < indices.size(); ++i) {
            // Calculate the difference between the positions of the current and previous occurrences
            let diff: number = i > 0 ? indices[i] - indices[i - 1] : 0;
            // Update the sum: add the difference for the previous occurrences,
            // and subtract it for the remaining occurrences
            sum += BigInt(i * diff - (count - i) * diff);
            // Assign the sum to the index of the current occurrence in the answer vector
            answer[indices[i]] = sum;
        }
    }

    return answer; // Return the computed sum of distances array
};
```

Time and Space Complexity

The given Python code involves creating a dictionary to group indices of identical elements and then computing the sum of distances for each unique element in the array.

Time Complexity:

1. Constructing the dictionary `d` where each value is a list of indices has a time complexity of $O(n)$, where `n` is the length of the array `arr`. This is because we iterate over the array once.
2. The outer loop `for v in d.values()` has at most `n` iterations in the worst-case scenario where all array elements are unique.
3. Inside this loop, we calculate the initial `val` for each group of identical elements. The summation `sum(v) - v[0] * m` where `m` is the number of times a distinct element appears in the array. The term `v[0] * m` is $O(n)$ since it involves `n` multiplications. As such, this part has a time complexity of $O(n)$ for each unique element.
4. The inner loop `for i, p in enumerate(v)` iterates `v` times (where `v` is the size of the list `v`). The calculation within this loop is $O(1)$, as it consists of simple arithmetic operations. Therefore, the time complexity for the inner loop is $O(n)$.

Combining these observations, the overall time complexity of the algorithm depends on the distribution of elements in the array. In the worst-case scenario, where all elements are unique, the time complexity tends to $O(n * n)$ which simplifies to $O(n)$. In the average and expected case, where elements are repeated, the time complexity would be $O(n * k * n)$ where `k` is the number of unique elements. However, since the sum of all `n` for each unique element is `n`, the average-case time complexity remains linear, or $O(n)$.

Space Complexity:

1. The space complexity of the dictionary `d` can be up to $O(n)$ in the case where all elements are distinct since we store an index for each element.
2. The `ans` list also takes up $O(n)$ space since it stores the result for each element in the `arr`.
3. Auxiliary space complexity is $O(1)$ as there are no other data structures that grow with input size; only a fixed number of variables are used.

Therefore, the space complexity of the code is $O(n)$ accounting for the space required for the dictionary `d` and the answer array `ans`.