2905. Find Indices With Index and Value Difference II

Medium Array Leetcode Link

Problem Description

In this problem, you are given an array `nums` of integers and two additional integers, `indexDifference` and `valueDifference`. Your task is to find any two indices `i` and `j` such that the following two conditions are satisfied:

1. The absolute difference between the indices `i` and `j` is greater than or equal to `indexDifference`, that is `abs(i - j) >= indexDifference`.
2. The absolute difference between the values at indices `i` and `j` in the array `nums` is greater than or equal to `valueDifference`, that is `abs(nums[i] - nums[j]) >= valueDifference`.

You need to return an array where the first element is the index `i` and the second element is the index `j`. If no such pair of indices exists, you should return `[-1, -1]`.

Note that the indices `i` and `j` may be the same, which implies that the constraints do not preclude the possibility of comparing an element with itself, as long as the index difference is non-existent (zero).

Intuition

The intuition behind the solution involves iterating through the array while keeping track of the minimum and maximum values seen so far, separated by at least `indexDifference`. As we traverse the array from the starting index set by `indexDifference`, we keep updating the minimum index `mi` and the maximum index `mx` when a smaller or larger element appears in the range that satisfies the index difference condition.

During this traversal:

- We want to check if the current maximum value we have seen so far (`nums[mx]`) minus the current value (`nums[i]`) is greater than or equal to `valueDifference`. If this condition is met, we have found a valid pair of indices `[mx, i]` and return them.
- Similarly, we check if the current value (`nums[i]`) minus the current minimum value we have seen so far (`nums[mi]`) is also greater than or equal to `valueDifference`. If this condition is met, we have found another valid pair of indices `[mi, i]` and return them.

The reason we can perform these checks is that by maintaining the smallest and largest values with the required index difference, we maximize the chances of finding a valid `valueDifference`. If no such pair is found throughout the traversal, we return `[-1, -1]` indicating the absence of such a pair of indices satisfying the given conditions.

Solution Approach

The solution uses a simple linear scan approach, enhanced with clever tracking of minimum and maximum values found within a valid index difference range. Here's the breakdown of the approach step-by-step, using the Solution provided:

1. The function `findIndices` takes the array `nums`, `indexDifference`, and `valueDifference` as its parameters.
2. Two pointers, `mi` and `mx`, are initialized to `0`. They will keep track of the indices of the minimum and maximum values found within the range specified by the `indexDifference`.
3. We use a `for` loop to iterate over the array starting from the `indexDifference` up to the length of the array. This ensures that we always have a range of elements where the earliest element (`nums[j]`) is exactly `indexDifference` apart from the current element (`nums[i]`).
4. At each iteration, we calculate `j` to point to the element which is `indexDifference` behind the current element `i`.
5. Next, we update the `mi` and `mx` pointers if the value at `nums[j]` is less than the current minimum or greater than the current maximum, respectively.
6. Immediately after updating `mi` and `mx`, we perform the check for the `valueDifference` condition:
   - If the difference between the current element's value (`nums[i]`) and the minimum value (`nums[mi]`) is greater than or equal to `valueDifference`, we return the indices `[mi, i]` as a result.
   - Similarly, if the difference between the maximum value (`nums[mx]`) and the current element's value is greater than or equal to `valueDifference`, we return the indices `[mx, i]` as a result.
7. If we do not find any indices that satisfy both conditions by the end of the loop, we return `[-1, -1]`.

Let's detail the functionality with the help of an example: Suppose `nums = [5, 3, 4, 2, 1, 3]`, `indexDifference = 3`, and `valueDifference = 2`.

- We initialize `mi = mx = 0`.
- When `i = 3` (`indexDifference = 3`), `j = 0`. Now, `nums[i] = 2` and `nums[j] = 5`. `mi` is unchanged as `nums[j] < nums[mi]` does not hold true, but `nums[j] > nums[mx]` holds true so `mx` is updated to `0`.
- Next checks for value difference, `nums[i] - nums[mi] < valueDifference` and `nums[mx] - nums[i]` is `5 - 2 = 3` which is greater than `valueDifference` so the function returns `[mx, i]` which is `[0,3]`.

The solution is efficient, traversing the list only once (O(n)), with constant space complexity (O(1)), as it doesn't require additional storage proportional to the input size.

Example Walkthrough

Let's walk through an example to illustrate the solution approach using the following inputs: `nums = [1, 5, 9, 3, 5]`, `indexDifference = 2`, and `valueDifference = 4`.

1. We initialize pointers `mi = mx = 0` to keep track of the indices of the minimum and maximum values found.
2. We iterate over the array starting from `i = indexDifference`, which is `2` in this case. At `i = 2`, `nums[i] = nums[2] = 9`.
3. For each `i`, we calculate `j = i - indexDifference`. When `i = 2`, `j = 2 - 2 = 0`.
4. Now we compare the current element `nums[i]` with `nums[mi]` and `nums[mx]` to check if we need to update `mi` or `mx`. Since `nums[mi] = nums[0] = 1` and `nums[mx] = nums[0] = 1`, and the current element `nums[i] = 9` is greater than both, we update `mx` to `2`.
5. We check the `valueDifference` condition with the updated indices. Here, `nums[mx] - nums[i]` is `0` since `mx = i = 2`. However, this check is redundant now as it's comparing the element with itself.
6. We move to the next iteration, with `i = 3`, and `nums[i] = nums[3] = 3`, and we calculate `j = i - indexDifference = 1`. Now, `nums[j] = nums[1] = 5`. At this point, `mi` remains `0` because `nums[1] > nums[0]`, but we don't update `mx` because it currently points to `2` where the value is higher (`9`).
7. We check `valueDifference` again for these values. We find that `abs(nums[mx] - nums[i]) = abs(9 - 3) = 6`, which is greater than `valueDifference`, so we return the indices `[mx, i]` which is `[2, 3]`.
8. Since we have found a valid pair that satisfies the conditions, the function terminates and returns `[2, 3]`.

Through this example, we demonstrated how the algorithm scans the list while maintaining the indices of the minimum and maximum values to efficiently find a valid pair `[i, j]` that satisfies both the index difference and value difference conditions. If no such pair is found by the time the algorithm has iterated through the entire array, it would return `[-1, -1]`.

Python Solution

```python
from typing import List

class Solution:
    def findIndices(self, nums: List[int], index_diff: int, value_diff: int) -> List[int]:
        # Initialize variables to keep track of the minimum and maximum values within the window
        min_index = max_index = 0

        # Iterate through the list starting from the index that allows the full index_diff window
        for i in range(index_diff, len(nums)):
            # Calculate the starting index of the window
            window_start = i - index_diff

            # Update min_index if a new minimum is found within the window
            if nums[window_start] < nums[min_index]:
                min_index = window_start

            # Update max_index if a new maximum is found within the window
            if nums[window_start] > nums[max_index]:
                max_index = window_start

            # Check if the current number minus the minimum number within the window meets the value_diff
            if nums[i] - nums[min_index] >= value_diff:
                # If the condition is met, return the indices that represent the difference
                return [min_index, i]

            # Check if the maximum number within the window minus the current number meets the value_diff
            if nums[max_index] - nums[i] >= value_diff:
                # If the condition is met, return the indices that represent the difference
                return [max_index, i]

        # If no such indices are found, return [-1, -1]
        return [-1, -1]
```

Java Solution

```java
class Solution {

    // Method to find the indices of two elements in the array satisfying given conditions
    public int[] findIndices(int[] nums, int indexDifference, int valueDifference) {
        // Initialize minimum and maximum element indices
        int minIndex = 0;
        int maxIndex = 0;

        // Iterate through the array starting from the indexDifference position
        for (int i = indexDifference; i < nums.length; ++i) {
            int j = i - indexDifference; // Calculate the index to compare with

            // Update the minimum index if a smaller value is found
            if (nums[j] < nums[minIndex]) {
                minIndex = j;
            }

            // Update the maximum index if a larger value is found
            if (nums[j] > nums[maxIndex]) {
                maxIndex = j;
            }

            // Check if the current value and the value at minIndex have the required valueDifference
            if (nums[i] - nums[minIndex] >= valueDifference) {
                return new int[] {minIndex, i}; // Return the indices if condition is satisfied
            }

            // Check if the value at maxIndex and the current value have the required valueDifference
            if (nums[maxIndex] - nums[i] >= valueDifference) {
                return new int[] {maxIndex, i}; // Return the indices if condition is satisfied
            }
        }

        // If no such pair is found, return [-1, -1]
        return new int[] {-1, -1};
    }
}
```

C++ Solution

```cpp
#include <vector>
using namespace std;

class Solution {
public:
    vector<int> findIndices(vector<int>& nums, int indexDifference, int valueDifference) {
        int minIndex = 0; // Initialize variable to keep track of the minimum value's index
        int maxIndex = 0; // Initialize variable to keep track of the maximum value's index

        // Iterate through the vector, starting from the index specified by indexDifference
        for (int i = indexDifference; i < nums.size(); ++i) {
            int comparisonIndex = i - indexDifference; // Calculate the comparison index

            // Update minIndex if the current comparison value is smaller than the smallest found so far
            if (nums[comparisonIndex] < nums[minIndex]) {
                minIndex = comparisonIndex;
            }

            // Update maxIndex if the current comparison value is greater than the largest found so far
            if (nums[comparisonIndex] > nums[maxIndex]) {
                maxIndex = comparisonIndex;
            }

            // If the value difference between the current index and the minIndex is greater than or equal to valueDifference, return
            if (nums[i] - nums[minIndex] >= valueDifference) {
                return {minIndex, i};
            }

            // If the value difference between the maxIndex and the current index is greater than or equal to valueDifference, return
            if (nums[maxIndex] - nums[i] >= valueDifference) {
                return {maxIndex, i};
            }
        }

        // If no such indices are found that satisfy the conditions, return {-1, -1}
        return {-1, -1};
    }
};
```

Typescript Solution

```typescript
function findIndices(nums: number[], indexDifference: number, valueDifference: number): number[] {
    // Initialize indices for the minimum and maximum values seen so far.
    let [minIndex, maxIndex] = [0, 0];

    // Iterate through the array, starting from the index specified by the `indexDifference`.
    for (let currentIndex = indexDifference; currentIndex < nums.length; ++currentIndex) {
        // Calculate the corresponding index that is `indexDifference` steps before the `currentIndex`.
        let comparisonIndex = currentIndex - indexDifference;

        // Update the `minIndex` if the current comparison value is lower than the stored minimum.
        if (nums[comparisonIndex] < nums[minIndex]) {
            minIndex = comparisonIndex;
        }

        // Update the `maxIndex` if the current comparison value is higher than the stored maximum.
        if (nums[comparisonIndex] > nums[maxIndex]) {
            maxIndex = comparisonIndex;
        }

        // If the difference between the current value and the minimum value seen so far meets
        // or exceeds the `valueDifference`, return the indices as an array.
        if (nums[currentIndex] - nums[minIndex] >= valueDifference) {
            return [minIndex, currentIndex];
        }

        // If the difference between the maximum value seen so far and the current value
        // meets or exceeds the `valueDifference`, return the indices as an array.
        if (nums[maxIndex] - nums[currentIndex] >= valueDifference) {
            return [maxIndex, currentIndex];
        }
    }

    // If no pairs meet the conditions, return [-1, -1] to indicate failure.
    return [-1, -1];
}
```

Time and Space Complexity

Time Complexity

The given code iterates over the array `nums` using a single loop starting from `indexDifference` up to `len(nums)`. For each iteration, the code performs constant-time operations such as comparisons and assignments. These operations do not depend on the size of the input array, except for the iteration that is linear in terms of the number of elements in `nums`. Therefore, the time complexity of the code is `O(n)`, where `n` is the length of the `nums` array.

Space Complexity

The space complexity of the code is determined by the extra space used aside from the input. The code uses a constant number of extra variables (`mi`, `mx`, `i`, `j`) that do not scale with the size of the input array. Therefore, the space complexity is `O(1)`, as there is no additional data structure or space-dependent on the input size used within the function.