2903. Find Indices With Index and Value Difference I

Easy Array

Leetcode Link

Problem Description

In this problem, you're given an integer array called `nums`, with a length of `n`. You're also given two integers: `indexDifference` and `valueDifference`. Your task is to find two indices `i` and `j` such that both `i` and `j` are within the range from `0` to `n - 1` and they meet the following criteria:

1. The absolute difference between `i` and `j` must be at least `indexDifference`,
2. The absolute difference between the values at `nums[i]` and `nums[j]` must be at least `valueDifference`.

In terms of the outcome, you need to return an array called `answer`. This array should consist of the indices `[i, j]` if such a pair of indices exists. If there are multiple valid pairs, you can return any one of them. If no valid pairs are found, then return `[-1, -1]`. An interesting point to note is that according to the problem statement, `i` and `j` can be the same index, which implies that `indexDifference` could potentially be `0`.

Intuition

The primary intuition behind the solution is the usage of a sliding window technique, combined with the maintenance of the minimum and maximum values within the window. The sliding window is defined by two pointers, `i` and `j`, that maintain a distance apart specified by `indexDifference`. The pointers are used as markers to capture a subarray within `nums` to check against our two conditions.

At the outset, `i` starts at the position `indexDifference`, and `j` starts at `0`. By doing this, the gap between `i` and `j` reflects the `indexDifference` requirement of our problem.

We maintain two variables, `mi` and `mx`, to keep track of the indices where the minimum and maximum values are found within our sliding window that ends at the current `j` index. While sliding `i` further along the array, we update `mi` and `mx` to account for the entry of new values into the window and the exit of old values.

When updating `mi` and `mx`, if `nums[j]` is less than `nums[mi]`, we reassign `mi` to `j`, because we have found a new minimum. Conversely, if `nums[j]` is greater than `nums[mx]`, we reassign `mx` to `j` due to identifying a new maximum.

After every movement of `i` and update of `mi` and `mx`, we check our two conditions against `nums[i]` (the current value at `i`). If the difference between `nums[i]` and the value at `nums[mi]` is greater than or equal to `valueDifference`, we have found a valid pair `[mi, i]`. Alternatively, if the difference between the maximum value (`nums[mx]`) and `nums[i]` is also greater than or equal to `valueDifference`, then `[mx, i]` is a valid pair. In this situation, we immediately return the pair as it meets our requirements.

If we reach the end of the array without finding a pair that satisfies both conditions, we conclude that no such pair exists, and we return the default output `[-1, -1]`.

The approach's efficiency comes from the fact that it avoids checking every possible pair of indices, which would otherwise lead to a less efficient solution with a higher time complexity.

Solution Approach

The solution uses a sliding window approach, which involves moving a fixed-size window across the array to examine sub-sections one at a time. This allows for checking the conditions over smaller segments in a single pass through the array, making the solution more efficient than a brute force approach that would involve examining all possible index pairs.

To implement this technique, the algorithm maintains two pointers: `i` and `j`. These pointers define the bounds of the sliding window. The pointer `i` starts at the index equal to `indexDifference` while `j` starts at `0`, thus immediately satisfying the condition `abs(i - j) >= indexDifference` because `i - j` is initialized to `indexDifference`.

As the algorithm iterates over the array, starting from `i = indexDifference`, it keeps track of the indices of the minimum and the maximum values found so far to the left of `j`. These indices are stored in variables `mi` and `mx`, respectively.

```
for i in range(indexDifference, len(nums)):
    j = i - indexDifference
```

Within the loop, we first check whether the current element at index `j` changes the minimum or maximum:

```
if nums[j] < nums[mi]:
    mi = j
if nums[j] > nums[mx]:
    mx = j
```

After updating `mi` and `mx`, we check if `nums[i]` differs enough from the minimum or maximum value to satisfy the `valueDifference` condition:

```
if nums[i] - nums[mi] >= valueDifference:
    return [mi, i]
if nums[mx] - nums[i] >= valueDifference:
    return [mx, i]
```

If either of these checks succeeds, the function immediately returns the corresponding pair of indices, as they meet both prescribed conditions. If the function reaches the end of the array without returning, this means no valid pairs were found, and thus it returns `[-1, -1]`.

In terms of data structures, no additional structures are needed beyond the use of a few variables to keep track of the indices and values encountered. This algorithm is space-efficient because it operates directly on the input array without requiring extra space proportional to the input size.

The choice of a sliding window and keeping track of minimum and maximum values eliminates the need to compare every element with every other element, thereby reducing the time complexity from O(n^2) to O(n), where n is the length of the input array.

Example Walkthrough

Let's walk through an example to illustrate the solution approach described above. Consider the integer array `nums = [1, 2, 3, 4, 5]`, with `indexDifference = 3` and `valueDifference = 3`.

Our task is to find indices `i` and `j` such that `abs(i - j) >= indexDifference` and `abs(nums[i] - nums[j]) >= valueDifference`.

According to the given solution approach, we initialize the sliding window by setting `i` to `indexDifference` and `j` to `0`. This immediately satisfies the first condition as the difference between the indices `i = 3` and `j = 0` is `3`, which is equal to `indexDifference`.

We then maintain `mi` and `mx` to keep track of the minimum and maximum values within the window.

Here's how we proceed step by step:

1. On the first iteration where `i = 3`:
   - We have `j = 0`
   - We initiate `mi` to `j` since there are no previous values to compare with, and similarly, `mx` is also initiated to `j`
   - The elements under consideration are `[nums[0], nums[3]]` i.e., `[1, 4]`
   - We see that `nums[3] - nums[0] = 4 - 1 = 3`, which is equal to `valueDifference`
   - Since `nums[3] - nums[0]` fulfills the `valueDifference` condition, we return `[0, 3]` as the indices that satisfy both conditions.

In this example, we directly found a pair that met both conditions, and thus we would return `[0, 3]`. However, if we needed to continue the iteration, we would:

2. Increment `i` to the next position and decrement `j` to keep the window size constant while satisfying the `indexDifference`. If `nums[j]` changes the minimum or maximum within the new window, update `mi` or `mx` accordingly.
3. Check if `nums[i]` differs enough from `nums[mi]` or `nums[mx]` as explained previously.
4. If we find a pair, we return it. If not, we continue iterating until `i` reaches the end of the array.

If no valid pairs are found by the end of the array, we return `[-1, -1]` as specified.

Python Solution

```python
from typing import List

class Solution:
    def findIndices(self, nums: List[int], idx_diff: int, val_diff: int) -> List[int]:
        # Initialize min and max index pointers
        min_idx = max_idx = 0

        # Traverse the array, starting from the index that enables the required index difference
        for current_idx in range(idx_diff, len(nums)):

            # Compute the comparison index that matches the index difference
            compare_idx = current_idx - idx_diff

            # Check and update the min and max indices based on the values at compare_idx
            if nums[compare_idx] < nums[min_idx]:
                min_idx = compare_idx
            if nums[compare_idx] > nums[max_idx]:
                max_idx = compare_idx

            # If the value difference requirement is met with the minimum, return the indices
            if nums[current_idx] - nums[min_idx] >= val_diff:
                return [min_idx, current_idx]

            # If the value difference requirement is met with the maximum, return the indices
            if nums[max_idx] - nums[current_idx] >= val_diff:
                return [max_idx, current_idx]

        # If the required value difference isn't found, return [-1, -1] as per problem statement
        return [-1, -1]

```

Java Solution

```java
class Solution {

    // Method to find indices in an array such that the difference between their values is at least a given value and their positions
    public int[] findIndices(int[] nums, int indexDifference, int valueDifference) {
        int minIndex = 0;   // Initializing the minimum value index
        int maxIndex = 0;   // Initializing the maximum value index

        // Loop through the array starting from the index equal to the indexDifference to the end of the array
        for (int i = indexDifference; i < nums.length; ++i) {
            int currentIndex = i - indexDifference;  // Calculate the index to compare with

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

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

            // Check if the difference between the current value and the minimum value found so far is at least valueDifference
            if (nums[i] - nums[minIndex] >= valueDifference) {
                return new int[] {minIndex, i}; // Return the indices if condition is met
            }

            // Check if the difference between the maximum value found so far and the current value is at least valueDifference
            if (nums[maxIndex] - nums[i] >= valueDifference) {
                return new int[] {maxIndex, i}; // Return the indices if condition is met
            }
        }

        // Return [-1, -1] if no such pair of indices is found
        return new int[] {-1, -1};
    }
}

```

C++ Solution

```cpp
#include <vector>

class Solution {
public:
    // Method to find the two indices in the array nums such that the difference
    // between their values is at least valueDifference and their index difference is exactly indexDifference
    // Args:
    //   nums: The input vector of integers
    //   indexDiff: The required difference between the indices of the two elements
    //   valueDiff: The minimum required value difference between the two elements
    // Returns:
    //   A vector with two elements: the indices of the elements in nums that satisfy the above criteria
    //   If no such pair exists, returns {-1, -1}
    std::vector<int> findIndices(std::vector<int>& nums, int indexDiff, int valueDiff) {
        int minIndex = 0, maxIndex = 0; // Initialized to store the index of minimum and maximum values seen so far
        for (int i = indexDiff; i < nums.size(); ++i) {
            int j = i - indexDiff; // Calculate the corresponding index
            if (nums[j] < nums[minIndex]) {
                minIndex = j; // Update minIndex if a new minimum is found
            }
            if (nums[j] > nums[maxIndex]) {
                maxIndex = j; // Update maxIndex if a new maximum is found
            }
            // Check if the difference between the current value and the minimum value seen so far is at least valueDiff
            if (nums[i] - nums[minIndex] >= valueDiff) {
                return {minIndex, i}; // Pair found, return indices
            }
            // Check if the difference between the maximum value seen so far and the current value is at least valueDiff
            if (nums[maxIndex] - nums[i] >= valueDiff) {
                return {maxIndex, i}; // Pair found, return indices
            }
        }
        return {-1, -1}; // If no pair found, return {-1, -1}
    }
};

```

Typescript Solution

```typescript
// This function finds two indices such that the difference of the elements at these indices
// is at least the given valueDifference and the indices are separated by at most the given
// indexDifference.
// nums: The array of numbers to search within
// indexDifference: The maximum allowed difference between the indices
// valueDifference: The minimum required difference between the values at the indices
// Returns an array with two numbers representing the indices, or [-1, -1] if no such pair exists
function findIndices(nums: number[], indexDifference: number, valueDifference: number): number[] {
    // Initialize the indices for the minimum value (minIndex) and maximum value (maxIndex) found.
    let minIndex = 0;
    let maxIndex = 0;

    // Iterate over the array, starting from the element at the indexDifference.
    for (let currentIndex = indexDifference; currentIndex < nums.length; currentIndex++) {
        // Calculate the index of the element we are comparing against,
        // which is indexDifference behind the current index.
        const compareIndex = currentIndex - indexDifference;

        // Update minIndex if the current compareIndex points to a new minimum value
        if (nums[compareIndex] < nums[minIndex]) {
            minIndex = compareIndex;
        }

        // Update maxIndex if the current compareIndex points to a new maximum value
        if (nums[compareIndex] > nums[maxIndex]) {
            maxIndex = compareIndex;
        }

        // Check if the difference between the current element and the minimum value is large enough.
        if (nums[currentIndex] - nums[minIndex] >= valueDifference) {
            return [minIndex, currentIndex]; // Return the indices if the condition is met.
        }

        // Check if the difference between the maximum value and the current element is large enough.
        if (nums[maxIndex] - nums[currentIndex] >= valueDifference) {
            return [maxIndex, currentIndex]; // Return the indices if the condition is met.
        }
    }

    // If no suitable pair of indices is found, return [-1, -1].
    return [-1, -1];
}

```

Time and Space Complexity

Time Complexity

The time complexity of the provided code is `O(n)`. This is achieved by iterating over the array once from `indexDifference` to the length of the array `len(nums)`. Only constant time checks and updates are performed within the loop, leading to a linear time complexity relative to the array's size.

Space Complexity

The space complexity of the code is `O(1)`. No additional space proportional to the input size is used. Only a fixed number of variables `mi`, `mx`, and `j` are used, which occupy constant space regardless of the input array size.