2420. Find All Good Indices

Medium Array Dynamic Programming Prefix Sum

Leetcode Link

Problem Description

In this problem, you are given an array `nums` of integers and a positive integer `k`. Your task is to find all the "good" indices in this array. An index `i` is considered "good" if it satisfies two conditions based on the elements around it:

1. The `k` elements immediately before index `i` are in non-increasing order. This means each element is less than or equal to the element before it.
2. The `k` elements immediately after index `i` are in non-decreasing order. This means each element is greater than or equal to the element after it.

The problem constraints are such that the "good" indices have to be in the range `k <= i < n - k`, which means you do not need to consider the first `k` indices and the last `k` indices of the array. The task is to return all "good" indices in increasing order.

For example, given `nums = [2,1,1,1,3,4,1]` and `k = 2`, index `3` is "good" because the two elements before it `[2,1]` are in non-increasing order and the two elements after it `[3,4]` are in non-decreasing order.

Intuition

The key to solving this problem lies in efficiently checking the two conditions for "good" indices without repeatedly iterating over `k` elements for each potential "good" index. To do this, we can preprocess the array to create two additional arrays:

1. An array `decr` to keep track of the length of non-increasing sequences ending at each index. `decr[i]` gives the length of the non-increasing sequence before index `i`.
2. An array `incr` to keep track of the length of non-decreasing sequences starting at each index. `incr[i]` gives the length of the non-decreasing sequence after index `i`.

By precomputing these values, we can quickly check whether an index is "good" by simply verifying if `decr[i] >= k` and `incr[i] >= k`. The preprocessing is efficient because each element only needs to be compared with its immediate predecessor or successor to update the `decr` and `incr` arrays respectively.

The process is as follows:

- Initialize the `decr` and `incr` arrays to be of length `n + 1` with all values set to `1`. This accounts for the fact that each index is by default part of a non-increasing or non-decreasing sequence of length 1 (itself).
- Populate the `decr` array starting from the second element up to the second to last element by comparing each element with the one before it.
- Populate the `incr` array starting from the second to last element back to the second element by comparing each element with the one after it.
- Iterate over the range `k` to `n - k` to collect all "good" indices where both `decr[i] >= k` and `incr[i] >= k` hold true.

The provided solution code correctly implements this approach, thus making the process of finding "good" indices efficient.

Solution Approach

The solution uses a straightforward approach with dynamic programming techniques to keep track of the lengths of non-increasing and non-decreasing subsequences around each index. Here's a step-by-step breakdown of how the algorithm and data structures are used:

1. **Initialization of Arrays**: The `decr` and `incr` arrays are initialized to be of size `n + 1`, with all elements set to `1`. These arrays are used to store the lengths of non-increasing and non-decreasing subsequences, respectively. This initial setup caters to the fact that solitary elements can be considered as subsequences of length one.
2. **Dynamic Programming - Filling `decr` Array**: The `decr` array is populated in a forward pass starting from index `2` up to `n - 1`. For every index `i`, if the current element `nums[i - 1]` is less than or equal to its previous element `nums[i - 2]`, then `decr[i]` is updated to be `decr[i - 1] + 1`, indicating that the non-increasing sequence continues.
   - The check `nums[i - 1] <= nums[i - 2]` confirms that the sequence is non-increasing at the point before `i`.
   - The `decr[i]` array captures the length of non-increasing order, which can be utilized later to check if an index `i` has `k` non-increasing elements before it.
3. **Dynamic Programming - Filling `incr` Array**: Similarly, the `incr` array is filled in a backward pass from index `n - 3` to `0`. For every index `i`, if the current element `nums[i + 1]` is less than or equal to the next element `nums[i + 2]`, then `incr[i]` is updated to be `incr[i + 1] + 1`.
   - The condition `nums[i + 1] <= nums[i + 2]` ensures that the sequence after `i` is non-decreasing.
   - By using `incr[i]`, we can efficiently determine if there are `k` non-decreasing elements after index `i`.
4. **Find Good Indices**: After populating `decr` and `incr`, the solution then iterates through the valid range of indices (`k` to `n - k - 1`) and checks whether the conditions for "good" indices are met for each index. An index `i` is "good" if `decr[i] >= k` and `incr[i] >= k`.
5. **Result Collection**: The indices that satisfy the good condition are added to the resultant list. This is done through a list comprehension that iterates over the valid range and includes the value `i` if it passes the check.

This algorithm effectively reduces the time complexity from what could be a brute-force check using nested loops (which would be O(nk)) down to O(n) because each element in `nums` is processed a constant number of times.

Example Walkthrough

Let's walk through a smaller example to illustrate the solution approach. Assume we have an array `nums = [5, 4, 3, 7, 8, 5, 4, 2, 1, 6]` and `k = 3`. We want to find all "good" indices.

Step 1: Initialization of Arrays

We initialize `decr` and `incr` arrays of length `n + 1` (where `n` is the length of `nums`, which is 10) and set all values to `1`.

Initial `decr`: `[1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1]`
Initial `incr`: `[1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1]`

Step 2: Dynamic Programming - Filling `decr` Array

We populate `decr` array from the second element to the second to last:

- `decr[2]` remains `1` (since `3 <= 5` is false).
- `decr[3]` = `decr[2] + 1` = `2` (since `7 <= 3` is true).
- `decr[4]` = `decr[3] + 1` = `3` (since `8 <= 7` is true).
- `decr[5]` resets to `1` (since `5 <= 8` is false).
- ... Continue this for the rest of the array.

Updated `decr`: `[1, 1, 1, 2, 3, 1, 1, 2, 3, 1, 1]`

Step 3: Dynamic Programming - Filling `incr` Array

We populate `incr` array in a backward pass:

- `incr[8]` = `incr[9] + 1` = `2` (since `2 <= 1` is false).
- `incr[7]` = `incr[8] + 1` = `3` (since `4 <= 2` is false).
- `incr[6]` resets to `1` (since `5 <= 4` is false).
- ... Continue this pass for the rest of the array.

Updated `incr`: `[1, 1, 1, 1, 1, 1, 1, 3, 2, 1, 1]`

Step 4: Find Good Indices

We then loop through the range `k` to `n - k` and check if both `decr[i] >= k` and `incr[i] >= k`:

- `i = 3` does not satisfy `decr[3] >= 3`.
- `i = 4` satisfies both `decr[4] >= 3` and `incr[4] >= 3`.
- ... Continue this for the range.

Step 5: Result Collection

We collect all "good" indices. For our example, the only "good" index we find is `4`. Thus, the output is `[4]`.

Applying this solution to larger arrays is efficient because it avoids repeated calculations for each index, instead utilizing the precomputed `decr` and `incr` arrays for quick lookups.

Python Solution

```python
from typing import List

class Solution:
    def good_indices(self, nums: List[int], k: int) -> List[int]:
        # Initialize the length of the 'nums' list
        n = len(nums)

        # Initialize two lists to track the non-increasing sequence lengths
        # to the left and non-decreasing sequence lengths to the right of every index
        non_increasing_lengths = [1] * (n + 1)
        non_decreasing_lengths = [1] * (n + 1)

        # Build the non-increasing sequence length array
        for i in range(2, n + 1):
            # If current and previous elements form a non-increasing sequence,
            # increment the length at the current index
            if nums[i - 1] <= nums[i - 2]:
                non_increasing_lengths[i] = non_increasing_lengths[i - 1] + 1

        # Build the non-decreasing sequence length array in reverse order
        for i in range(n - 3, -1, -1):
            # If the next element and the one after it form a non-decreasing sequence,
            # increment the length at the current index
            if nums[i + 1] <= nums[i + 2]:
                non_decreasing_lengths[i] = non_decreasing_lengths[i + 1] + 1

        # Find all 'good' indices, where both the non-increasing sequence on the left
        # and the non-decreasing sequence on the right are at least 'k' elements long
        good_indices = [i for i in range(k, n - k) if non_increasing_lengths[i] >= k and non_decreasing_lengths[i] >= k]

        return good_indices
```

Java Solution

```java
class Solution {
    public List<Integer> goodIndices(int[] nums, int k) {
        // Get the length of the array
        int n = nums.length;
        // Create array to store the lengths of decreasing sequences
        int[] decreasingLengths = new int[n];
        // Create array to store the lengths of increasing sequences
        int[] increasingLengths = new int[n];

        // Initially set lengths of sequences to 1 for all elements
        Arrays.fill(decreasingLengths, 1);
        Arrays.fill(increasingLengths, 1);

        // Calculate lengths of non-increasing sequences to the left of every index
        for (int i = 1; i < n; ++i) {
            if (nums[i] <= nums[i - 1]) {
                decreasingLengths[i] = 1 + decreasingLengths[i - 1];
            }
        }

        // Calculate lengths of non-decreasing sequences to the right of every index
        for (int i = n - 2; i > 0; --i) {
            if (nums[i] <= nums[i + 1]) {
                increasingLengths[i] = 1 + increasingLengths[i + 1];
            }
        }

        // Initialize list to store all the good indices
        List<Integer> goodIndices = new ArrayList<>();

        // Traverse the array and add indices to the list that are good indices
        for (int i = k; i < n - k; ++i) {
            // Check if there are at least k non-increasing elements before it
            // and at least k non-decreasing elements after it
            if (decreasingLengths[i - 1] >= k && increasingLengths[i + 1] >= k) {
                goodIndices.add(i);
            }
        }

        // Return the list containing all the good indices found
        return goodIndices;
    }
}
```

C++ Solution

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

class Solution {
public:
    // Function to find all good indices based on the given conditions
    vector<int> goodIndices(vector<int>& nums, int k) {
        int n = nums.size(); // Get the number of elements in nums
        // Arrays to keep the lengths of non-increasing and non-decreasing subsequences
        vector<int> nonIncreasing(n, 1);
        vector<int> nonDecreasing(n, 1);

        // Calculate lengths of non-increasing subsequences from the start
        for (int i = 1; i < n; ++i) {
            if (nums[i] <= nums[i - 1]) {
                nonIncreasing[i] = nonIncreasing[i - 1] + 1;
            }
        }

        // Calculate lengths of non-decreasing subsequences from the end
        for (int i = n - 2; i >= 0; --i) {
            if (nums[i] <= nums[i + 1]) {
                nonDecreasing[i] = nonDecreasing[i + 1] + 1;
            }
        }

        // Vector to store the good indices
        vector<int> goodIndices;

        // Iterate through the array to find all the good indices
        for (int i = k; i < n - k; ++i) {
            // Check if the current index i is a good index
            if (nonIncreasing[i - 1] >= k && nonDecreasing[i + 1] >= k) {
                goodIndices.push_back(i);
            }
        }

        // Return the list of all good indices
        return goodIndices;
    }
};
```

Typescript Solution

```typescript
// TypeScript doesn't have a direct equivalent to the C++ <vector> library, so we use arrays instead.

// Function to find all good indices based on the given conditions
function goodIndices(nums: number[], k: number): number[] {
    const n = nums.length; // Total number of elements in nums
    // Arrays to keep the lengths of non-increasing and non-decreasing subsequences
    let nonIncreasing: number[] = new Array(n).fill(1);
    let nonDecreasing: number[] = new Array(n).fill(1);

    // Calculate lengths of non-increasing subsequences from the start
    for (let i = 1; i < n; ++i) {
        if (nums[i] <= nums[i - 1]) {
            nonIncreasing[i] = nonIncreasing[i - 1] + 1;
        }
    }

    // Calculate lengths of non-decreasing subsequences from the end
    for (let i = n - 2; i >= 0; --i) {
        if (nums[i] <= nums[i + 1]) {
            nonDecreasing[i] = nonDecreasing[i + 1] + 1;
        }
    }

    // Array to store the good indices
    let goodIndicesList: number[] = [];

    // Iterate through the array to find all the good indices
    for (let i = k; i < n - k; ++i) {
        // Check if the current index i is a good index
        if (nonIncreasing[i - 1] >= k && nonDecreasing[i + 1] >= k) {
            goodIndicesList.push(i);
        }
    }

    // Return the list of all good indices
    return goodIndicesList;
}
```

Time and Space Complexity

The given Python code consists of two main parts – first, creating non-increasing (`decr`) and non-decreasing (`incr`) prefix arrays, and second, iterating through the range `[k, n - k)` to check and collect good indices based on the condition given in the problem.

Time Complexity

1. **Building the non-increasing prefix array `decr`**:
   - We iterate once from the index `2` to `n - 1` (one-based indexing). In each iteration, we check a condition and possibly increment a value at the current index based on the previous index.
   - Each operation is $O(1)$, and since we do this `n - 2` times, this part has a time complexity of $O(n)$.
2. **Building the non-decreasing prefix array `incr`**:
   - Similarly, we iterate backward from `n - 3` to `0`, doing an $O(1)$ operation each time.
   - This back traversal is done `n - 1` times, also yielding a time complexity of $O(n)$.
3. **Finding good indices:**
   - We iterate through the range `[k, n - k)` and check two conditions for each index `i`, which again takes $O(1)$ per index.
   - There are `n - 2k` such indices, leading this part to have a time complexity of $O(n - 2k)$. However, since `k` is at most `n`, this simplifies to $O(n)$.

Considering all three parts, the overall time complexity combines to:

- $O(n) + O(n) + O(n) = O(3n)$ which simplifies to $O(n)$.

Space Complexity

1. **Space used by `decr` and `incr` arrays:**
   - Both arrays have a size of `n + 1`, so the space taken by each is $O(n)$.
   - Combined, they utilize `2 * (n + 1)` memory space, which simplifies to $O(2n)$ or just $O(n)$.
2. **Space used for the output list:**
   - In the worst-case scenario, every index from `k` to `n - k` may be a good index, so this list can take up to `n - 2k` spaces.
   - The worst case for this list is also when `k` is very small compared to `n`, which would make its space complexity approach $O(n)$.

Therefore, combining the space complexities from the arrays and the final output list, we get:

- $O(n) + O(n) = O(2n)$ which simplifies to $O(n)$.

In conclusion, the time complexity of the code is $O(n)$ and the space complexity is $O(n)$.