1708. Largest Subarray Length K

Easy Greedy Array

Leetcode Link

Problem Description

In this coding problem, we're given an integer array `nums` containing distinct integers, and our task is to find the largest subarray of `nums` with a length of `k`. The term "largest" in the context of subarrays is based on a specific comparison rule. Specifically, an array `A` is considered larger than an array `B` if at the first index (`i`) where the elements of `A` and `B` differ (`A[i] != B[i]`), the element in `A` is greater than the one in `B` (`A[i] > B[i]`). This is similar to comparing strings in lexicographic order, but with numbers.

A subarray, as defined in the problem, is a contiguous section of the original array that maintains the order of elements. We need to find such a subarray of exact length `k` that is the largest among all possible subarrays of the same length, based on the comparison rule described.

Intuition

To determine the largest subarray with a specific length, we first need to understand the comparison rule better. Since the first different element defines which array is larger, we know that having the largest possible element at the start of the subarray will make it as large as possible. Any subarray with a largest element positioned more towards the end cannot be the largest because a comparison will be decided earlier, at the index where the largest elements of two subarrays are different.

With this understanding, our approach is to find the largest element in the array that can be the starting element of a subarray with length `k`. Because the array is zero-indexed, the largest possible starting index of such a subarray is `len(nums) - k`. After we find the index of the largest possible starting element, we can simply return the subarray that starts at that index and spans `k` elements.

The provided code snippet realizes this approach by calculating the maximum element that could start a subarray of length `k` (using `max(nums[: len(nums) - k + 1])`), finds its index, and then creates a subarray starting from that index with the specified length.

Solution Approach

The solution uses a straightforward approach aligning with the intuition behind the problem. Here's how the implemented algorithm works:

1. **Finding the Maximum Starting Element**: Since we are looking for the largest subarray of length `k`, the largest possible starting element must be within the first `len(nums) - k + 1` elements of the array. Any element beyond this range could not be the start of a subarray of length `k` as it would overflow the bounds of the array. Therefore, the method `max(nums[: len(nums) - k + 1])` is used to find the value of the maximum possible starting element.

2. **Determining the Index**: Once we have the value of the largest starting element, we need to find where it is located within the array. We utilize the `index()` method to determine the first occurrence of this maximum element in the array: `i = nums.index(mx)`.

3. **Slicing the Subarray**: With the index `i` located, we can slice the array to get a subarray starting from this index, going up to `i + k`. The slicing operation `nums[i : i + k]` performs this step, efficiently extracting the required subarray.

The algorithm does not use any complex data structures and is based on simple array manipulation techniques. It leverages Python's built-in functions to find maximum values and to slice lists, which makes the implementation concise and very efficient.

There are no complicated patterns or algorithms at play here; it's just a direct application of understanding the problem's constraints and applying basic array operations to achieve the goal. The solution's time complexity is primarily determined by the search for the maximum element, which is O(n), with n being the number of elements within the search range, and the index retrieval, which is also O(n) in the worst case. The slicing operation is O(k), but since k <= n, the overall time complexity remains O(n).

Example Walkthrough

Let's use the following array `nums` and `k` as an example to illustrate the solution approach:

```
nums = [5, 2, 4, 3, 1, 7]
k = 3
```

We want to find the largest subarray of length `k = 3`. Following the steps in the solution approach:

1. **Finding the Maximum Starting Element**:
   Since `k = 3`, we need to find the largest element that will start our subarray within the first `len(nums) - k + 1 = 6 - 3 + 1 = 4` elements. Those elements are `[5, 2, 4, 3]`. The largest element within this range is `5`.

2. **Determining the Index**:
   We find the index of the element `5` in the array, which is at index `0`.

3. **Slicing the Subarray**:
   We extract the subarray starting at index `0` and ending at index `0 + k = 3`. Using array slicing, we get `nums[0:3]`, which gives us the subarray `[5, 2, 4]`.

Based on the solution approach and the provided array `nums`, the largest subarray of length `k` is `[5, 2, 4]`. This subarray starts with the largest possible starting element and adheres to the comparison rule that the earliest difference in elements determines the larger array.

This example confirms that the algorithm is successfully applying the steps to find the largest subarray of a specified length based on the given comparison rule. The solution is straightforward, and with Python's built-in functions for finding maximum values and indexing, it becomes very efficient.

Python Solution

```python
from typing import List

class Solution:
    def largest_subarray(self, nums: List[int], k: int) -> List[int]:
        # Find the maximum element in the first 'len(nums) - k + 1' elements,
        # because any subarray longer than this won't have enough elements
        # to contain a subarray of length 'k' starting from the max element.
        max_val = max(nums[:len(nums) - k + 1])

        # Find the index of the first occurrence of the maximum element
        index_of_max = nums.index(max_val)

        # Return the subarray of length 'k' starting from the index of the maximum element
        return nums[index_of_max : index_of_max + k]

```

Java Solution

```java
class Solution {
    public int[] largestSubarray(int[] nums, int k) {
        // Initialize variables to store the starting index of the maximum subarray
        int maxStartIndex = 0;
        int maxElement = nums[0]; // Assume the first element is the maximum to begin with

        // Iterate through the array to find the starting index of the subarray
        // with the largest possible first number, as the subarray must be the largest lexicographically
        for (int currentIndex = 0; currentIndex <= nums.length - k; currentIndex++) {
            // If the current element is greater than the previously found maximum,
            // update maxElement and the starting index of the largest subarray
            if (maxElement < nums[currentIndex]) {
                maxElement = nums[currentIndex];
                maxStartIndex = currentIndex;
            }
        }

        // Create an array to store the answer
        int[] largestSubarray = new int[k];

        // Copy the elements of the largest subarray from the original array
        for (int j = 0; j < k; j++) {
            largestSubarray[j] = nums[maxStartIndex + j];
        }

        // Return the largest subarray found
        return largestSubarray;
    }
}

```

C++ Solution

```cpp
#include <vector> // Include the vector header for using the vector container
#include <algorithm> // Include the algorithm header for the max_element function

class Solution {
public:
    // Function to find the largest subarray of length k.
    std::vector<int> largestSubarray(std::vector<int>& nums, int k) {

        // Find the position of the maximum element that could be the start of a subarray of length k.
        // The search space is reduced by size - k + 1 because any element beyond that cannot be the start of a subarray of length k.
        auto startPosition = std::max_element(nums.begin(), nums.begin() + nums.size() - k + 1);

        // Create a vector from the startPosition to startPosition + k, which is the required subarray.
        // This uses the iterator range constructor of vector to create a subarray.
        return std::vector<int>(startPosition, startPosition + k);
    }
};

```

Typescript Solution

```typescript
// Import necessary functions from 'lodash' to replicate C++ std::max_element functionality
// Note: You would need to install 'lodash' with `npm install lodash` to make this work
import { maxBy, range } from 'lodash';

// Function to find the largest subarray of length k
function largestSubarray(nums: number[], k: number): number[] {
    // Find the position of the maximum element that could be the start of a subarray of length k
    // The search space is reduced by nums.length - k + 1 because any element beyond that
    // cannot be the start of a subarray of length k
    let maxStartIndex = maxBy(range(nums.length - k + 1), i => nums[i]);

    // If maxStartIndex is undefined due to some error, default to 0
    maxStartIndex = maxStartIndex === undefined ? 0 : maxStartIndex;

    // Create a subarray from the maxStartIndex to maxStartIndex + k, which is the required subarray
    return nums.slice(maxStartIndex, maxStartIndex + k);
}

// Example usage:
// let result = largestSubarray([1,4,3,2,5], 3);
// console.log(result); // Outputs: [3,2,5]

```

Time and Space Complexity

Time Complexity

The time complexity of the code is determined by two main operations: finding the maximum value in the first `len(nums) - k + 1` elements, and finding the index of the maximum value found.

- `max(nums[: len(nums) - k + 1])`: Finding the maximum value in a list of `n-k+1` elements takes `O(n-k+1)` operations, where `n` is the length of the `nums` list.
- `nums.index(mx)`: In the worst case, finding the index of a value in a list takes `O(n)` time.

Therefore, the time complexity of the code is `O(n-k+1 + n)`, which simplifies to `O(n)` since `k` is a constant and `n-k+1` is linear with respect to `n`.

Space Complexity

The space complexity of the function is mainly due to the storage of the subarray that is returned. No additional space is used that grows with the size of the input, except for the output list.

- `return nums[i : i + k]`: Creating a subarray of size `k` takes `O(k)` space.

The space complexity for this code is `O(k)`, where `k` is the size of the window (subarray) to return. This space is required to store the output subarray.