2863. Maximum Length of Semi-Decreasing Subarrays

```
Problem Description
```

<u>Array</u>

Medium

with an element strictly greater than the element it ends with. Additionally, we should return 0 if no such subarray exists in nums. To illustrate, consider the following examples:

In this problem, we are given an array nums consisting of integer elements. Our goal is to determine the length of the longest

contiguous subarray that meets a specific criterion called "semi-decreasing." A subarray is considered semi-decreasing if it starts

• If nums = [5,3,1,2,4], the longest semi-decreasing subarray is [5,3,1], and its length is 3.

Hash Table

Sorting

The problem is essentially asking us to find the maximum distance between the starting index of the highest element and the

adding 1 (to account for array indexing starting from 0).

search, thus providing a much more efficient solution.

ending index of a lower element, ensuring that this sequence forms a subarray. We must handle this task efficiently, as the naive approach of examining every possible subarray could prove to be too slow.

• In the case of nums = [1,2,3,4,5], there are no semi-decreasing subarrays, so the answer is 0.

Intuition To arrive at the solution, we must track the valuable information that can help us identify semi-decreasing subarrays. Recognizing that a semi-decreasing subarray's first element must be strictly greater than its last element, we can sort the unique elements of the array in reverse (from largest to smallest) and keep track of their indices.

A Python dictionary (here d) can be employed to store the indices of each unique element. By iterating through the array and

Once this preparation is done, we can iterate over the unique elements in descending order. For each element x, we use the last occurrence of x (since we want to maximize the subarray's length, and a semi-decreasing subarray's last element must be lower than the first) and find the distance to the smallest index k seen so far of elements greater than x. This smallest index k represents the farthest we can go to the left to start a semi-decreasing subarray that ends with the given element x.

populating this dictionary, we ensure that for each element, we have a list of all indices where it occurs.

The intuition behind maintaining k as the minimum index seen so far is that as we proceed with lower elements, we want to extend our subarray as much to the left as possible. ans represents the length of the longest semi-decreasing subarray found so far, which gets updated each time we find a longer semi-decreasing subarray by subtracting k from the last index of x and

This approach systematically constructs semi-decreasing subarrays over the original array without the need for exhaustive

The implementation of the solution uses a combination of sorting, hash maps (dictionaries in Python), and linear traversal to find the longest semi-decreasing subarray. Here's a step-by-step breakdown of the algorithm based on the provided solution code: The first step is to iterate through the given array nums and build a hash map (d), where the key is the element value, and the value is a list of indices at which the element occurs. This is achieved with the following line of code:

considered. We sort the keys of the dictionary d in reverse order so we can traverse from the highest element to the lowest. This is important because we are looking for subarrays where the first element is strictly greater than the last element.

and k is initialized to inf (infinity), representing the lowest index seen so far for elements greater than the current one being

ending with x. We achieve this by computing the distance between the last occurrence of x and the smallest index seen so far (k). We add 1 to this distance since the array is zero-indexed: ans = $\max(ans, d[x][-1] - k + 1)$

After considering the element x, we update k to hold the smallest index which could potentially be the start of a longer

This means for subsequent smaller elements, we have the information about how far left (to a higher element) we can extend

By iterating through the elements in this fashion, we ensure that we are always considering semi-decreasing subarrays (since we move from larger to smaller elements), and we keep extending our reach to the left as much as possible, thus finding the length of the longest such subarray efficiently.

 $d = {$ 7: [0], 5: [1], 4: [2], 6: [3],

Now, for key 6, its last index is 3, and the smallest index seen so far is 0. The length of the semi-decreasing subarray would

Step 4: Iterate over the sorted keys, calculating the potential length of the semi-decreasing subarray and updating ans:

3: [4]

ans = 0

k = inf

•

•

•

•

Update k to min(k, d[4][0]), so k remains 0. •

For key 5, we calculate ans = max(ans, d[5][-1] - k + 1), which is max(4, 1 - 0 + 1) and ans stays 4.

For key 4, we have ans = max(ans, d[4][-1] - k + 1), which is max(4, 2 - 0 + 1) and ans stays 4.

Finally, for key 3, we have ans = max(ans, d[3][-1] - k + 1), which is max(4, 4 - 0 + 1), so ans stays 4.

Step 3: Sort the keys of dictionary d in reverse order. We get the sorted keys as [7, 6, 5, 4, 3].

Update k to min(k, d[7][0]), which will be min(inf, 0) so k becomes 0.

Update k to min(k, d[6][0]), which will be min(0, 3) so k stays 0.

k does not need to be updated as we have finished the iterations.

Therefore, the length of the longest semi-decreasing subarray in nums is 4.

Initialize a dictionary to hold lists of indices for each unique number

Initialize variables for the maximum subarray length and the smallest index seen so far

Solution Implementation **Python**

Step 5: The largest ans we found was 4, corresponding to the subarray [7, 5, 4, 6], which starts with 7 and ends with 6.

// TreeMap to store values in descending order and their index occurrences in list TreeMap<Integer, List<Integer>> valueIndicesMap = new TreeMap<>(Comparator.reverseOrder()); // Loop through the nums array and store each number with its index

// minus the minimum index encountered so far plus 1

minIndexSoFar = Math.min(minIndexSoFar, indices.get(0));

C++ #include <vector>

int maxSubarravLength(vector<int>& nums) {

map<int, vector<int>, greater<int>> indexMap;

for (int i = 0; i < nums.size(); ++i) {</pre>

indexMap[nums[i]].push_back(i);

// Fill the indexMap with indices for each number in nums

int maxLength = 0; // Tracks the maximum subarray length found

int minIndexWithinMaxValue = 1 << 30; // Initialize to a very large number</pre>

// This function finds the maximum length of a subarray for an array of numbers // where the maximum number in the subarrav is at least twice as large as all other numbers in the subarray. function maxSubarrayLength(nums: number[]): number { // Create a map to store indices for each number in the array. const indexMap: Map<number, number[]> = new Map(); // Populate the map with each number's indices.

TypeScript

};

- for (const key of sortedKeys) { // Get the array of indices corresponding to the current key. const indices = indexMap.get(key)!; // Calculate the subarray length where the current key is possibly the maximum // and compare it with the length seen so far to keep the longest. maxSubarrayLength = Math.max(maxSubarrayLength, indices.at(-1) - smallestIndex + 1);
- index_dict[num].append(i) # Initialize variables for the maximum subarray length and the smallest index seen so far max_length, smallest_index_seen = 0, inf # Iterate over the numbers in descending order

0(n log n).

• Enumeration and creation of dictionary: The first for loop goes through all the entries in nums, thus it is O(n) where n is the length of nums. • Sorting of dictionary keys: The keys of the dictionary are sorted in reverse order. Since the number of unique elements (keys in the dictionary) could be at most n, the sort operation has a complexity of O(u log u) where u is the number of unique keys, with a worst-case scenario being

The time complexity of the code is determined by several key operations:

- only accesses the first and last elements of the list of indices associated with each key. Given these considerations, the overall time complexity is 0(n log n) due to the sorting step which is likely the bottleneck.
- Dictionary storage: The dictionary d stores lists of indices per unique numbers in nums. In the worst case, where all elements are unique, the

Hence, the total space complexity is O(n), taking into account the space occupied by the dictionary which is the dominant term.

• Iterating over sorted keys: This iteration potentially goes through all unique keys (at most n), and for each key, the complexity is 0(1) since it

The enumerate function is used to get both the index (i) and the value (x) during iteration. We introduce two variables, ans and k. ans is initialized to 0, representing the length of the longest subarray found so far,

for i, x in enumerate(nums):

for x in sorted(d, reverse=True):

 $k = \min(k, d[x][0])$

our subarray.

Example Walkthrough

After the first step, d will look like this:

Step 2: Initialize ans to 0 and k to inf.

be 3 - 0 + 1 = 4. So ans becomes 4.

Update k to min(k, d[5][0]), so k remains 0.

def maxSubarrayLength(self, nums: List[int]) -> int:

Populate the dictionary with indices of each number

semi-decreasing subarray for the next iterations:

d[x].append(i)

Solution Approach

For each element x (sorted from largest to smallest), we calculate the prospective length of the semi-decreasing subarray

Let's walk through a small example to illustrate the solution approach. Consider the following simple array: nums = [7, 5, 4, 6, 3]Applying the algorithm step by step:

Step 1: Build a hash map (d) with element values as keys and a list of their indices as values.

For key 7, since it's the largest element, there cannot be a semi-decreasing subarray starting with 7. But we use its index 0 to update the smallest index k.

from collections import defaultdict

return max_length

index_dict = defaultdict(list)

for i, num in enumerate(nums):

index_dict[num].append(i)

max_length, smallest_index_seen = 0, inf

Iterate over the numbers in descending order

for num in sorted(index dict, reverse=True):

Return the maximum computed subarray length

public int maxSubarravLength(int[] nums) {

from math import inf

class Solution:

Java

class Solution {

#include <map>

class Solution {

public:

#include <algorithm>

Update max length using the highest index of the current number # and the smallest index seen so far max length = max(max length, index dict[num][-1] - smallest index seen + 1)# Update the smallest index seen so far to be the lowest index of the current number smallest_index_seen = min(smallest_index_seen, index_dict[num][0])

for (int i = 0; i < nums.length; ++i) { valueIndicesMap.computeIfAbsent(nums[i], k -> new ArrayList<>()).add(i); // Initialize the result for maximum subarray length (answer) int answer = 0; // Initialize a variable to hold the minimum index encountered so far int minIndexSoFar = Integer.MAX_VALUE; // Iterate through the sorted values in descending order for (List<Integer> indices : valueIndicesMap.values()) { // Update the answer with the maximum length found till now

// Return the maximum subarray length return answer;

// The length is calculated using the last occurrence index of a value

answer = Math.max(answer, indices.get(indices.size() - 1) - minIndexSoFar + 1);

// Map to store numbers and their indices, sorted in descending order of the key (number)

// Update the minimum index if the first occurrence index of the current value is smaller

// Iterate through the map starting from the largest key for (auto& [num, indices] : indexMap) { // Update maxLength using the distance from the current smallest index (minIndexWithinMaxValue) // to the last occurrence of the current number maxLength = max(maxLength, indices.back() - minIndexWithinMaxValue + 1); // Update minIndexWithinMaxValue to the smallest index seen so far minIndexWithinMaxValue = min(minIndexWithinMaxValue, indices.front());

return maxLength; // Return the maximum length found

const sortedKeys = Array.from(indexMap.keys()).sort((a, b) => b - a); // Initialize the maximum subarray length variable. let maxSubarrayLength = 0; // Initialize a variable to keep track of the smallest index seen so far. let smallestIndex = Infinity; // Iterate through the sorted keys.

// Get all unique numbers from the map keys and sort them in descending order.

// Update the smallestIndex with the smallest index of the current key's indices.

Initialize a dictionary to hold lists of indices for each unique number

Update max length using the highest index of the current number

smallest_index_seen = min(smallest_index_seen, index_dict[num][0])

max length = max(max length, index dict[num][-1] - smallest index seen + 1)

Update the smallest index seen so far to be the lowest index of the current number

for (let i = 0; $i < nums.length; ++i) {$

indexMap.get(nums[i])!.push(i);

indexMap.set(nums[i], []);

if (!indexMap.has(nums[i])) {

// Return the maximum subarray length found. return maxSubarrayLength; from collections import defaultdict from math import inf class Solution:

def maxSubarrayLength(self, nums: List[int]) -> int:

for num in sorted(index dict, reverse=True):

and the smallest index seen so far

Return the maximum computed subarray length

Populate the dictionary with indices of each number

index_dict = defaultdict(list)

for i, num in enumerate(nums):

smallestIndex = Math.min(smallestIndex, indices[0]);

return max_length Time and Space Complexity **Time Complexity**

Space Complexity The space complexity of the code is influenced by:

space complexity due to the dictionary would be O(n).

• Miscellaneous variables: Variables ans, k, and the space for iteration do not depend on size of nums and thus contribute a constant factor.