Problem Description

contiguous subarray that meets a specific criterion called "semi-decreasing." A subarray is considered semi-decreasing if it starts with an element strictly greater than the element it ends with. Additionally, we should return 0 if no such subarray exists in nums.

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

of examining every possible subarray could prove to be too slow.

A Python dictionary (here d) can be employed to store the indices of each unique element. By iterating through the array and populating this dictionary, we ensure that for each element, we have a list of all indices where it occurs.

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.

gets updated each time we find a longer semi-decreasing subarray by subtracting k from the last index of x and adding 1 (to account for array indexing starting from 0). This approach systematically constructs semi-decreasing subarrays over the original array without the need for exhaustive search,

Solution Approach

1. 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:

1 for i, x in enumerate(nums): d[x].append(i)

2. We introduce two variables, ans and k. ans is initialized to 0, representing the length of the longest subarray found so far, and k is initialized to inf (infinity), representing the lowest index seen so far for elements greater than the current one being considered.

3. 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

- 4. For each element x (sorted from largest to smallest), we calculate the prospective length of the semi-decreasing subarray ending with x. We achieve this by computing the distance between the last occurrence of x and the smallest index seen so far
- 1 ans = $\max(ans, d[x][-1] k + 1)$

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

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

the longest such subarray efficiently.

Applying the algorithm step by step:

1 k = min(k, d[x][0])

decreasing subarray for the next iterations:

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

(k). We add 1 to this distance since the array is zero-indexed:

Example Walkthrough Let's walk through a small example to illustrate the solution approach. Consider the following simple array:

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

Step 1: Build a hash map (d) with element values as keys and a list of their indices as values. After the first step, d will look like this:

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

update the smallest index k.

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

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

6: [3],

1 ans = 0

2 k = inf

1 nums = [7, 5, 4, 6, 3]

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

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[5][0]), so k remains 0. • For key 4, we have ans = max(ans, d[4][-1] - k + 1), which is max(4, 2 - 0 + 1) and ans stays 4.

• 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

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 be 3

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.

Populate the dictionary with indices of each number

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

index_dict = defaultdict(list)

for i, num in enumerate(nums):

index_dict[num].append(i)

public int maxSubarrayLength(int[] nums) {

int minIndexSoFar = Integer.MAX_VALUE;

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

for (auto& [num, indices] : indexMap) {

function maxSubarrayLength(nums: number[]): number {

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

// Initialize the maximum subarray length variable.

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

let maxSubarrayLength = 0;

let smallestIndex = Infinity;

for (const key of sortedKeys) {

// Iterate through the sorted keys.

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

// Iterate through the map starting from the largest key

// to the last occurrence of the current number

return maxLength; // Return the maximum length found

1 // This function finds the maximum length of a subarray for an array of numbers

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

const sortedKeys = Array.from(indexMap.keys()).sort((a, b) => b - a);

// Initialize a variable to keep track of the smallest index seen so far.

// Create a map to store indices for each number in the array.

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

// Update minIndexWithinMaxValue to the smallest index seen so far

minIndexWithinMaxValue = min(minIndexWithinMaxValue, indices.front());

2 // where the maximum number in the subarray is at least twice as large as all other numbers in the subarray.

maxLength = max(maxLength, indices.back() - minIndexWithinMaxValue + 1);

max_length, smallest_index_seen = 0, inf

Iterate over the numbers in descending order

for num in sorted(index_dict, reverse=True):

and the smallest index seen so far

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

1 from collections import defaultdict from math import inf class Solution:

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

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

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])

 $max_length = max(max_length, index_dict[num][-1] - smallest_index_seen + 1)$

// TreeMap to store values in descending order and their index occurrences in list

TreeMap<Integer, List<Integer>> valueIndicesMap = new TreeMap<>(Comparator.reverseOrder());

• For key 5, we calculate ans = max(ans, d[5][-1] - k + 1), which is max(4, 1 - 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 5: The largest ans we found was 4, corresponding to the subarray [7, 5, 4, 6], which starts with 7 and ends with 6.

23 # Return the maximum computed subarray length return max_length 26

// Initialize the result for maximum subarray length (answer)

// Initialize a variable to hold the minimum index encountered so far

// Loop through the nums array and store each number with its index for (int i = 0; i < nums.length; ++i) { valueIndicesMap.computeIfAbsent(nums[i], k -> new ArrayList<>()).add(i); 9

int answer = 0;

Java Solution

class Solution {

Python Solution

10

11

12

13

14

15

16

17

18

19

20

21

22

10

11

12

13

14

15

16

13

14

15

16

17

18

19

20

21

22

23

24

25

26

27

28

29

30

32

10

11

12

13

14

15

16

17

18

19

20

21

22

23

24

25

31 };

17 // Iterate through the sorted values in descending order for (List<Integer> indices : valueIndicesMap.values()) { 18 // Update the answer with the maximum length found till now 19 // The length is calculated using the last occurrence index of a value 20 // minus the minimum index encountered so far plus 1 answer = Math.max(answer, indices.get(indices.size() -1) - minIndexSoFar + 1); 23 24 // Update the minimum index if the first occurrence index of the current value is smaller 25 minIndexSoFar = Math.min(minIndexSoFar, indices.get(0)); 26 27 28 // Return the maximum subarray length 29 return answer; 30 32 C++ Solution 1 #include <vector> 2 #include <map> #include <algorithm> class Solution { public: int maxSubarrayLength(vector<int>& nums) { // Map to store numbers and their indices, sorted in descending order of the key (number) map<int, vector<int>, greater<int>> indexMap; 9 10 // Fill the indexMap with indices for each number in nums for (int i = 0; i < nums.size(); ++i) {</pre> 12

// Update maxLength using the distance from the current smallest index (minIndexWithinMaxValue)

const indexMap: Map<number, number[]> = new Map(); // Populate the map with each number's indices. for (let i = 0; i < nums.length; ++i) {</pre> if (!indexMap.has(nums[i])) { 9

Typescript Solution

```
26
           // Get the array of indices corresponding to the current key.
27
           const indices = indexMap.get(key)!;
28
29
           // Calculate the subarray length where the current key is possibly the maximum
30
           // and compare it with the length seen so far to keep the longest.
           maxSubarrayLength = Math.max(maxSubarrayLength, indices.at(-1) - smallestIndex + 1);
31
32
33
           // Update the smallestIndex with the smallest index of the current key's indices.
34
           smallestIndex = Math.min(smallestIndex, indices[0]);
35
37
       // Return the maximum subarray length found.
       return maxSubarrayLength;
38
40
Time and Space Complexity
Time Complexity
The time complexity of the code is determined by several key operations:
```

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

Space Complexity

- 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 0(n log n). • 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 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.

• 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

- The space complexity of the code is influenced by: • Dictionary storage: The dictionary d stores lists of indices per unique numbers in nums. In the worst case, where all elements are
- constant factor.

• In the case of nums = [1,2,3,4,5], there are no semi-decreasing subarrays, so the answer is 0. The problem is essentially asking us to find the maximum distance between the starting index of the highest element and the ending index of a lower element, ensuring that this sequence forms a subarray. We must handle this task efficiently, as the naive approach

To arrive at the solution, we must track the valuable information that can help us identify semi-decreasing subarrays. Recognizing

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

our subarray as much to the left as possible. ans represents the length of the longest semi-decreasing subarray found so far, which thus providing a much more efficient solution.

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 intuition behind maintaining k as the minimum index seen so far is that as we proceed with lower elements, we want to extend

The enumerate function is used to get both the index (i) and the value (x) during iteration.

because we are looking for subarrays where the first element is strictly greater than the last element.

Intuition 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.

To illustrate, consider the following examples: • If nums = [5,3,1,2,4], the longest semi-decreasing subarray is [5,3,1], and its length is 3.

unique, the 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

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