962. Maximum Width Ramp

Medium | Stack | Array | Monotonic Stack | Leetcode Link

Problem Description

In this problem, we're given an integer array called `nums`. We are tasked to find the maximum width of a "ramp" within this array. A "ramp" is defined as a pair of indices `(i, j)` with `i < j`, where the value at the first index is less than or equal to the value at the second one, i.e., `nums[i] <= nums[j]`. The width of this ramp is the difference between the two indices, `j - i`. The goal is to find the maximum width of such a ramp in the given array. If there are no ramps possible, we should return `0`.

Intuition

To solve this problem, the intuition is that we can optimize the search for ramps by maintaining candidates for the start of the ramp in a stack. The key observations are:

- We would like to start ramps as early as possible in the array but with the smallest possible values.
- If we encounter an element greater than or equal to an element at a prospective start of a ramp, we can attempt to form a ramp.
- We process the array backwards to assess the widest ramp possible with the start points we've collected.

The solution leverages a stack to store indices of potential "start points" for ramps. These start points are the indices where the value is less than or equal to all later values we've seen so far as we iterate from front to back. Therefore:

1. We iterate through the array, adding indices to the stack if the current element is smaller than the element at the top of the stack (which corresponds to the last element we added). This way, we keep track of all potential start points.
2. In a second pass, we iterate from the end of the array to the start. For each element, we compare it with the elements at the indices in our stack.
3. Since the stack is sorted by the values at those indices, every time we find an element greater than or equal to the value at the top index on the stack, we can pop the index off and calculate the width of a ramp. If the ramp is wider than any prior ramp we've found, we keep track of this new max width.
4. We continue popping from the stack and calculating ramp widths until the stack is empty or we find an element in `nums` that is smaller than the value at the current stack's top index, meaning no ramp is possible from this index. If the stack empties, we know we've found the maximum width ramp and we can break early.

By using this approach, we ensure that we always measure the width of ramps that start from the earliest point in the array and are as wide as possible. This stack-based approach allows us to quickly disregard ramp starts that will never be optimal, thus optimizing the search for the maximum width ramp in `nums`.

Solution Approach

The solution makes use of a mono-stack or a stack that preserves order by certain criteria, in this case, the increasing order of values from the front to the back. A mono-stack is an ideal data structure to approach this problem due to the following characteristics:

- It's a Last In, First Out (LIFO) structure, which allows us to compare recent entries with new ones efficiently.
- It ensures that we have the smallest element seen so far at any given point on the bottom. This potentially results in the largest width ramp.

Here's a step-by-step breakdown of the solution using two main steps:

1. **Building the stack**:
   - We initialize an empty stack called `stk`.
   - We then iterate through the array `nums` using index `i` and the value `v` at each index.
   - If the stack is empty or the current value `v` is less than the value in `nums` at the index on the top of the stack (`nums[stk[-1]]`), we push the current index `i` onto the stack. This way, we're maintaining a stack of indices where each is a candidate for the start of a ramp (the lowest element so far).
2. **Finding the maximum width ramp**:
   - Next, we initialize the answer `ans` as `0`.
   - We reverse iterate over the array from the last index down to `0`.
   - For each element, we check if the stack is not empty and the current element `nums[i]` is greater than or equal to the element at the index from the top of the stack (`nums[stk[-1]]`).
   - If both conditions hold true, it means the current index `i` can serve as the ending index of a ramp starting at the index from the top of the stack.
   - We pop off the index at the top of the stack and use it to calculate the width of the ramp by subtracting the start index from the current index (`i - stk.pop()`).
   - We store the maximum width encountered in `ans`.
   - If at any point the stack becomes empty, we can stop iterating since it means all potential start points of ramps have been evaluated.

By using this approach, the algorithm minimizes comparisons and avoids unnecessary iterations, hence optimizing the search for the maximum width ramp in the array `nums`. The complexity of the algorithm is `O(N)`, where `N` is the size of the input array, because each element is pushed to the stack at most once and popped at most once.

Example Walkthrough

Let's illustrate the solution approach with a small example using the array `nums = [6, 0, 8, 2, 1, 5]`.

1. **Building the stack**:
   - Initialize an empty stack `stk`.
   - Iterate through each element of `nums`.

Index (i)	Value (v)	Top of Stack	Action
0	6	-	Push index 0 onto `stk` (stack is now [0])
1	0	6	Push index 1 onto `stk` since `nums[0] > nums[1]` (stack is now [0, 1])
2	8	0	Do nothing (8 is not smaller than 6 or 0)
3	2	0	Push index 3 onto `stk` since `nums[1] > nums[3]` (stack is now [0, 1, 3])
4	1	2	Push index 4 onto `stk` since `nums[3] > nums[4]` (stack is now [0, 1, 3, 4])
5	5	1	Do nothing (5 is not smaller than 6, 0, 2, or 1)

   - After this step, `stk = [0, 1, 3, 4]`.

2. **Finding the maximum width ramp**:
   - Initialize `ans` to 0.
   - Reverse iterate over `nums` and compare the current value with the top of `stk`.
   - For this example, we start with the last index, 5.

Index (i)	Num Value	Stack (`stk`)	Top of Stack Value	Action
5	5	[0, 1, 3, 4]	1	Pop index 4. Calculate ramp width `5 - 4 = 1`.
5	5	[0, 1, 3]	2	Pop index 3. Calculate ramp width `5 - 3 = 2`. Since `2 > 1`,
5	5	[0, 1]	0	Do nothing (5 is not smaller than 6 or 0)
4	1	[0, 1]	0	Do nothing (1 is not greater than 0)
3	2	[0, 1]	0	Do nothing (2 is not greater than 0)
2	8	[0, 1]	0	Pop index 1. Calculate ramp width `2 - 1 = 1`. Since `1 < 2`,
2	8	[0]	6	Pop index 0. Calculate ramp width `2 - 0 = 2`. Since `2 == 2`,
1	0	[]	-	(No action since the stack is empty)

   - After comparing elements, `ans` remains at its highest value, which is `2`.

At the end of this process, we have determined that the maximum width of a ramp in the array `nums = [6, 0, 8, 2, 1, 5]` is `2`, which occurs between the elements at indices 1 and 3 (value 0 and value 2, respectively), and also between the elements at indices 0 and 2 (value 6 and value 8, respectively). The algorithm optimized the search for the maximum width ramp efficiently, using a mono-stack to keep track of potential starting points and reverse iterating to find the longest ramp possible.

Python Solution

```python
from typing import List

class Solution:
    def maxWidthRamp(self, nums: List[int]) -> int:
        # Initialize a stack to keep track of indices of the potential start of the ramp.
        stack = []

        # Iterate through the given numbers along with their indices.
        for index, value in enumerate(nums):
            # If the stack is empty or the current value is less than the value at the last index of the stack,
            # this could be a potential start of a ramp, so store the index.
            if not stack or nums[stack[-1]] > value:
                stack.append(index)

        # Initialize the answer variable with 0 to keep track of the maximum width.
        max_width = 0

        # Iterate backwards from the end of nums to find the maximum ramp.
        # We go backwards since we're looking for the largest index j (end of the ramp)
        # that forms a ramp with the start indices in the stack.
        for i in range(len(nums) - 1, -1, -1):
            # While the stack is not empty and the current number is greater than or equal to
            # the number at the index of the top of the stack, we have a potential ramp.
            while stack and nums[stack[-1]] <= nums[i]:
                # Calculate the width of the ramp and update the max_width if this ramp is wider.
                max_width = max(max_width, i - stack.pop())
            # If the stack becomes empty, no more start positions are left to check,
            # and the maximum ramp is found.
            if not stack:
                break

        # Return the maximum width of the ramp found.
        return max_width

```

Java Solution

```java
class Solution {
    public int maxWidthRamp(int[] nums) {
        int n = nums.length; // Store the length of the array
        Deque<Integer> stack = new ArrayDeque<>(); // Use a deque as a stack for indices

        // Populate the stack with indices of elements where each is the smallest
        // seen so far from left to right
        for (int i = 0; i < n; ++i) {
            if (stack.isEmpty() || nums[stack.peek()] > nums[i]) {
                stack.push(i);
            }
        }

        int maxWidth = 0; // Initialize the maximum width of the ramp

        // Iterate from the end of the array towards the beginning
        for (int i = n - 1; i >= 0; --i) {
            // Try to extend the ramp from the end as far as possible
            while (!stack.isEmpty() && nums[stack.peek()] <= nums[i]) {
                int startIndex = stack.pop(); // Retrieve the start index of a ramp
                maxWidth = Math.max(maxWidth, i - startIndex); // Update the maxWidth
            }

            // If the stack is empty, all possible start indices have been explored
            if (stack.isEmpty()) {
                break;
            }
        }

        return maxWidth; // Return the maximum width of a ramp found
    }
}

```

C++ Solution

```cpp
#include <vector>
#include <stack>
#include <algorithm> // For using max()

class Solution {
public:
    int maxWidthRamp(vector<int>& nums) {
        int n = nums.size(); // Get the size of the input vector.
        stack<int> indexStack; // Initialize a stack to keep track of indices.

        // Fill the stack with the indices of elements which are either the first element or smaller than the
        // the smallest encountered element so far, as they can potentially form the starting point of a ramp.
        for (int i = 0; i < n; ++i) {
            if (indexStack.empty() || nums[indexStack.top()] > nums[i]) {
                indexStack.push(i);
            }
        }

        int maxRampWidth = 0; // Initialize maximum ramp width (answer variable).

        // Moving from right to left, attempt to find the maximum ramp width by comparing elements
        // On the stack with elements to their right. As this is a ramp, we're looking for a smaller or equal
        // number to the left (marked by the index on the stack) of our current number.
        for (int i = n - 1; i >= 0; --i) {
            while (!indexStack.empty() && nums[indexStack.top()] <= nums[i]) {
                // Update maximum ramp width if the current ramp is wider.
                maxRampWidth = max(maxRampWidth, i - indexStack.top());
                indexStack.pop(); // Pop from the stack as we found a valid ending point for this starting index.
            }
            // There's no need to continue if the stack is empty.
            if (indexStack.empty()) break;
        }

        return maxRampWidth; // Return the maximum width of a ramp found.
    }
};

```

Typescript Solution

```typescript
function maxWidthRamp(nums: number[]): number {
    const n: number = nums.length; // Get the length of the input array.
    const indexStack: number[] = []; // Initialize an array to work as a stack for indices.

    // Fill the stack with the indices of elements which are either the first element or smaller than
    // the smallest encountered element so far, as they could be the starting point of a ramp.
    for (let i = 0; i < n; i++) {
        if (indexStack.length === 0 || nums[indexStack[indexStack.length - 1]] > nums[i]) {
            indexStack.push(i);
        }
    }

    let maxRampWidth: number = 0; // Initialize the maximum ramp width (answer variable).

    // Move from right to left, attempting to find the maximum ramp width by comparing elements
    // on the stack with elements to their right. For a ramp, a smaller or equal
    // number to the left of the current number is sought (marked by the index on the stack).
    for (let i = n - 1; i >= 0; i--) {
        while (indexStack.length > 0 && nums[indexStack[indexStack.length - 1]] <= nums[i]) {
            // Update maximum ramp width if the current ramp is wider.
            maxRampWidth = Math.max(maxRampWidth, i - indexStack[indexStack.length - 1]);
            indexStack.pop(); // Pop from the stack as we've found a valid ending point for this starting index.
        }
        // There's no need to continue if the stack is empty.
        if (indexStack.length === 0) {
            break;
        }
    }

    return maxRampWidth; // Return the maximum width of a ramp found.
}

// Example usage:
// const nums = [6,0,8,2,1,5];
// const rampWidth = maxWidthRamp(nums);

```

Time and Space Complexity

The time complexity of the code can be considered as `O(N)`, where `N` is the length of the input `nums` list. It is because the first loop iterates over all elements once to build the stack, and the second loop iterates over all elements in the worst case if the stack isn't emptied early. However, each element is added and then popped from the stack at most once, leading to a linear time complexity over the two loops.

The space complexity of the code is `O(N)` in the worst case when the elements in the `nums` list are strictly decreasing. In such a case, each `nums` index would be appended to the stack, thus the stack could grow to the same size as the `nums` list. In the general case, the space complexity depends on the input but will not exceed `O(N)`.