2016. Maximum Difference Between Increasing Elements

Easy Array

Leetcode Link

Problem Description

The problem provides us with an array `nums` containing integers, where our task is to find the maximum difference between two of its elements, `nums[i]` and `nums[j]`, with the condition that `i` is less than `j` (that is `i < j`) and `nums[i]` is less than `nums[j]` (`nums[i] < nums[j]`). The maximum difference is defined as `nums[j] - nums[i]`. The constraints are such that `i` and `j` must be valid indices within the array (`0 <= i < j < n`). If no such pair of indices exists, the function should return `-1`.

Intuition

The solution to this problem is driven by a single pass approach that keeps track of the smallest element found so far while iterating through the array. The idea is to continuously update the smallest element `mi` as we move through the array and simultaneously compute the difference between the current element and this smallest element whenever the current element is larger than `mi`.

The steps are as follows:

- Initialize a variable `mi` to store the minimum value found so far; set it to `inf` (infinity) at the start because we haven't encountered any elements in the array yet.
- Initialize a variable `ans` to store the maximum difference found; set it to `-1` to indicate that no valid difference has been calculated yet.
- Iterate over each element `x` in the `nums` array.
  - If the current element `x` is greater than the current minimum `mi`, then there's potential for a larger difference. Calculate the difference `x - mi` and update `ans` to be the maximum of `ans` and this newly calculated difference.
  - If the current element `x` is not greater than `mi`, then we update `mi` to be the current element `x`, since it's the new minimum.

After iterating through the entire array, `ans` will contain the maximum difference found following the given criteria, or it will remain `-1` if no such pair was found (meaning the array was non-increasing). This approach is efficient as it only requires a single pass through the array, hence the time complexity is O(n), where n is the number of elements in `nums`.

Solution Approach

The implementation of the solution utilizes a simple linear scanning algorithm that operates in a single pass through the array. No complex data structures are needed; only a couple of variables are used to keep track of the state as we iterate. Here is a more detailed breakdown of the approach:

- We have an array `nums` of integers.
- Two variables are initialized:
  - `mi`: This is initially set to `inf`, representing an "infinity" value which ensures that any element we encounter will be less than this value. `mi` serves as the minimum value encounter so far as we iterate through the array.
  - `ans`: This is the answer variable, initialized to `-1`. It will store the maximum difference encountered that satisfies the problem's condition.
- We begin iterating over each element `x` in the `nums` array using a `for` loop.
  - If the current element `x` is greater than `mi`, it means we've found a new pair that could potentially lead to a larger difference. In this case, we find the difference between `x` and `mi` and update `ans` to be the maximum of itself and this new difference. This step uses the `max` function: `ans = max(ans, x - mi)`.
  - If `x` is not greater than `mi`, then the current element `x` becomes the new minimum, which updates `mi` to `x`. It's a crucial step, as this update is necessary to ensure we always have the smallest value seen so far, which allows us to find the largest possible difference.
- After the loop completes, the `ans` variable will hold the maximum difference possible. If `ans` remained `-1`, it implies there was no valid `(i, j)` pair that satisfied the condition (`nums[i] < nums[j]` with `i < j`).

The use of the infinity value for `mi` is a common pattern to simplify code as it avoids the need for special checks on the index being valid or the array being non-empty. Similarly, using `-1` for `ans` follows a typical convention to indicate that a satisfying condition was not found during the computation.

The algorithm's simplicity and the lack of additional data structures contribute to its time efficiency, making it a solution with O(n) time complexity, where `n` is the number of elements in `nums`.

Example Walkthrough

Let's illustrate the solution approach with a small example. Suppose we have the following array `nums`:

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

Our task is to find the maximum difference `nums[j] - nums[i]` with the conditions that `i < j` and `nums[i] < nums[j]`.

We'll initialize our variables `mi` and `ans`:

- `mi` is set to `inf`, which in practical scenarios can be represented by a very large number, assuming the array does not contain larger numbers.
- `ans` is set to `-1` as we've not yet found a valid pair.

Now we begin iterating through the array `nums`:

1. First element (7):
   - `mi` is `inf`, so we update `mi` to be 7.
   - No difference is calculated because this is the first element.
2. Second element (1):
   - 1 is less than `mi` (which is 7), so we now update `mi` to 1.
   - The `ans` is still `-1` because we didn't find a larger element yet.
3. Third element (5):
   - 5 is greater than `mi` (which is 1), so we calculate the difference (5 - 1 = 4).
   - Now, we update `ans` to the maximum of `-1` and 4, which is 4.
4. Fourth element (3):
   - 3 is greater than `mi` (1), so we calculate the difference (3 - 1 = 2).
   - `ans` remains 4 because 4 is greater than 2.
5. Fifth element (6):
   - 6 is greater than `mi` (1), so we calculate the difference (6 - 1 = 5).
   - Update `ans` to 5, since that's greater than the current `ans` of 4.
6. Sixth element (4):
   - 4 is greater than `mi` (1), so we calculate the difference (4 - 1 = 3).
   - `ans` remains 5, as it is still the maximum difference found.

At the end of the iteration, `ans` contains the value `5`, which is the maximum difference obtained by subtracting an earlier, smaller number from a later, larger number in the array, satisfying `nums[j] - nums[i]` where `i < j` and `nums[i] < nums[j]`.

Therefore, the maximum difference in the given array `nums` is `5`.

Python Solution

```python
class Solution:
    def maximumDifference(self, nums: List[int]) -> int:
        # Initialize the minimum value seen so far to infinity
        min_val = float('inf')
        # Initialize the maximum difference as -1 (default if not found)
        max_diff = -1

        # Iterate over each number in the list
        for num in nums:
            # If the current number is greater than the minimum value seen,
            # there is a potential for a new maximum difference
            if num > min_val:
                # Update the maximum difference if the current difference
                # is greater than the previous maximum difference
                max_diff = max(max_diff, num - min_val)
            else:
                # If the current number is not greater than the minimum value seen,
                # update the minimum value to the current number
                min_val = num

        # Return the maximum difference found; if no valid difference was found,
        # this will return -1
        return max_diff
```

Java Solution

```java
class Solution {
    public int maximumDifference(int[] nums) {
        // Initialize the minimum value to a very large value
        int minVal = Integer.MAX_VALUE;
        // Initialize the answer to -1, assuming there is no positive difference found
        int maxDiff = -1;

        // Loop through each number in the input array
        for (int num : nums) {
            // If the current number is greater than the minimum value found so far
            if (num > minVal) {
                // Update the maximum difference with the greater value between the current maximum difference
                // and the difference between the current number and the minimum value found so far
                maxDiff = Math.max(maxDiff, num - minVal);
            } else {
                // If the current number is not greater than the minimum value found so far,
                // then update the minimum value to the current number
                minVal = num;
            }
        }

        // Return the maximum difference found, or -1 if no positive difference exists
        return maxDiff;
    }
}
```

C++ Solution

```cpp
class Solution {
public:
    int maximumDifference(vector<int>& nums) {
        // Initialize min_value with a large number well above any expected input.
        int min_value = INT_MAX; // INT_MAX is more idiomatic than 1 << 30.

        // Initialize the answer with -1, indicating no positive difference found yet.
        int max_difference = -1;

        // Iterate over each number in the input vector 'nums'.
        for (int num : nums) {
            // If the current number is larger than the min_value seen so far,
            // update max_difference with the greater of current max_difference and
            // the difference between current number and min_value.
            if (num > min_value) {
                max_difference = max(max_difference, num - min_value);
            } else {
                // If the current number is smaller than min_value, update min_value
                // with the current number.
                min_value = num;
            }
        }

        // Return the maximum positive difference found, or -1 if no such difference exists.
        return max_difference;
    }
};
```

Typescript Solution

```typescript
function maximumDifference(nums: number[]): number {
    // Get the length of the nums array
    const numElements = nums.length;
    // Initialize the minimum value to the first element of the nums array
    let minimumValue = nums[0];
    // Initialize the maximum difference as -1; this will change if a greater difference is found
    let maxDifference = -1;

    // Loop through the array starting from the second element
    for (let index = 1; index < numElements; index++) {
        // Calculate the current difference and update the maxDifference if the current difference is greater
        maxDifference = Math.max(maxDifference, nums[index] - minimumValue);
        // Update the minimumValue with the smallest number encountered so far
        minimumValue = Math.min(minimumValue, nums[index]);
    }

    // If no positive maximum difference was found, return -1; otherwise, return the maxDifference
    return maxDifference > 0 ? maxDifference : -1;
}
```

Time and Space Complexity

The time complexity of the provided code can be analyzed by looking at the operations within the main loop. The code iterates through the list of numbers once, performing constant-time operations within each iteration, such as comparison, subtraction, and assignment. Thus, the time complexity is `O(n)` where `n` is the length of the input list `nums`.

For space complexity, the code uses a fixed amount of additional memory to store the variables `mi` and `ans`. Regardless of the size of the input list, this does not change, which means the space complexity is constant, or `O(1)`.