<https://leetcode.com/problems/non-decreasing-array/description/>

Given an array nums with n integers, your task is to check if it could become non-decreasing by modifying at most one element.

We define an array is non-decreasing if nums[i] <= nums[i + 1] holds for every i (0-based) such that (0 <= i <= n - 2).

**Example 1:**

**Input:** nums = [4,2,3]

**Output:** true

**Explanation:** You could modify the first 4 to 1 to get a non-decreasing array.

**Example 2:**

**Input:** nums = [4,2,1]

**Output:** false

**Explanation:** You cannot get a non-decreasing array by modifying at most one element.

**Constraints:**

n == nums.length

1 <= n <= 10^4

-10^5 <= nums[i] <= 10^5

**Attempt 1: 2023-08-27**

**Solution 1: LIS (10 min, refer to L300)**

class Solution {

    public boolean checkPossibility(int[] nums) {

        int result = 1;

        // dp[i] denotes the LIS ending at index i

        int len = nums.length;

        // Test out by input: nums = {1}

        if(len == 1) {

            return true;

        }

        int[] dp = new int[len];

        Arrays.fill(dp, 1);

        for(int i = 0; i < len; i++) {

            for(int j = 0; j < i; j++) {

                // Compare to L300 strict '<', here

                // non-decreasing can be '<='

                if(nums[j] <= nums[i]) {

                    dp[i] = Math.max(dp[i], dp[j] + 1);

                }

            }

            result = Math.max(result, dp[i]);

        }

        // At most one elememnt modify means LIS length

        // can be either len - 1 or len

        return result >= len - 1;

    }

}

Time Complexity O(N^2)

Space Complexity O(N)

**Refer to**

**L300 Longest Increasing Subsequence classic 1D DP solution**

class Solution {

    public int lengthOfLIS(int[] nums) {

        int result = 1;

        // dp[i] denotes the LIS ending at index i

        int len = nums.length;

        int[] dp = new int[len];

        Arrays.fill(dp, 1);

        for(int i = 0; i < len; i++) {

            // For each element nums[i], if there's an smaller element

            // nums[j] before it, the result will be maximum of current

            // LIS length ending at i: dp[i], and LIS ending at that

            // j + 1: dp[j] + 1. +1 because we are including the current

            // element and extending the LIS ending at j.

            // 假设dp[i]代表加入第i个数能构成的最长升序序列长度，我们就是要在

            // dp[0]到dp[i-1]中找到一个最长的升序序列长度，又保证序列尾值

            // nums[j]小于nums[i]，然后把这个长度加上1就行了。

            // 同时，我们还要及时更新最大长度。

            for(int j = 0; j < i; j++) {

                if(nums[j] < nums[i]) {

                    dp[i] = Math.max(dp[i], dp[j] + 1);

                    // Or we can put here, same effect

                    //result = Math.max(result, dp[i]);

                }

            }

            // Don't forget to update global maximum for each 'i'

            result = Math.max(result, dp[i]);

        }

        return result;

    }

}

Time Complexity : O(N^2)

Space Complexity : O(N)

**Solution 2: Greedy (60 min)**

**Wrong Solution**

**Test out by input**

**nums = [3,4,2,3]**

**Output = true**

**Expected = false**

class Solution {

    public boolean checkPossibility(int[] nums) {

        int n = nums.length;

        int count = 0;

        for(int i = 0; i < n - 1; i++) {

            // Check if any two element relation as decreasing

            // in original nums array

            if(nums[i] > nums[i + 1]) {

                count++;

            }

            if(count > 1) {

                return false;

            }

        }

        return true;

    }

}

**Correct Solution**

class Solution {

    public boolean checkPossibility(int[] nums) {

        int n = nums.length;

        int count = 0;

        for(int i = 0; i < n - 1; i++) {

            // Check if any two element relation as decreasing

            // in original nums array

            if(nums[i] > nums[i + 1]) {

                count++;

                if(count > 1) {

                    return false;

                }

                // Case 1: Modify nums[i] to nums[i + 1]

                // if i == 0 or nums[i - 1] <= nums[i + 1].

                // e.g {1,7(i),3,4} -> convert 7 to 3

                // Case 2: Modify nums[i + 1] to nums[i]

                // if i == len - 2 or nums[i] <= nums[i + 2].

                // e.g {4,7(i),3,9} -> convert 3 to 7

                if(i == 0 || nums[i - 1] <= nums[i + 1]) {

                    nums[i] = nums[i + 1];

                } else {

                    nums[i + 1] = nums[i];

                }

            }

            // Relocate return immediately after count > 1 make it fast

            //if(count > 1) {

            //    return false;

            //}

        }

        return true;

    }

}

Time Complexity O(N)

Space Complexity O(N)

**Refer to**

<https://leetcode.com/problems/non-decreasing-array/solutions/106826/java-c-simple-greedy-like-solution-with-explanation/comments/109180>

Great solution. Below is a little explanation for those having trouble understanding.

The problem requires that every number has to be equal or greater than previous number.

If we encounter a failing condition where the number is not greater or equal to previous (smaller than previous) we need to make a correction. **Correction can be made in either of two ways**:

Make the previous number smaller or equal to current number

Make the current number equal to previous number

We can do (1) as long as the number at position i-2 is equal or lower than the current element. (if i-2 is valid)

In case 1 below we can do this at (3) (i = 2) as the element 1 (i = 0) fulfills 1 <= 3. We can replace 7 with 3.

However, this cannot be done in case 2 as 4 <= 3 does not satisfy.

**Correction with technique (1) takes priority** as there is no risk in lowering the value but there is a risk associated if the value is increased. (Consider scenario in case 1 if we replace 3 with 7, it will fail to satisfy the condition for the last element)

We have to make correction with (2) if we cannot achieve it by (1). In which case we increase the value of current element by matching previous element. In case 2, we replace 3 with 7.

Also we only compare condition with the previous element only because as we move forward we know the previous numbers are already validated .

Case 1:

7

/\ 4

/ \ /

/ \/

/ 3

1

Case 2:

9

/

7 /

/\ /

/ \ /

/ \ /

4 \ /

\/

3(i)

**Refer to chatGPT**

LeetCode 665, titled **"Non-decreasing Array,"** requires you to determine if it is possible to modify at most one element of a given array to make it non-decreasing.

**Problem Statement:**

Given an integer array nums, your task is to return true if you can make it non-decreasing by modifying at most one element. Otherwise, return false.

A non-decreasing array is an array where nums[i] <= nums[i + 1] for every i (0 <= i < nums.length - 1).

**Approach:**

We will iterate through the array and check where the condition nums[i] > nums[i + 1] is violated.

If this happens more than once, it is impossible to fix the array by modifying only one element, so we return false.

If this happens only once, there are two cases to consider for modification:

Modify nums[i] to nums[i + 1] if i == 0 or nums[i - 1] <= nums[i + 1].

Modify nums[i + 1] to nums[i] if i == len - 2 or nums[i] <= nums[i + 2].

If any of the above conditions is met, the modification is possible, and we continue checking. Otherwise, we return false.

**Java Solution:**

class Solution {

public boolean checkPossibility(int[] nums) {

int count = 0; // Count of modifications needed

for (int i = 0; i < nums.length - 1; i++) {

if (nums[i] > nums[i + 1]) {

count++;

if (count > 1) {

return false; // More than one modification needed

}

// If this is the first element or the element before it is less than or equal to nums[i+1]

if (i == 0 || nums[i - 1] <= nums[i + 1]) {

nums[i] = nums[i + 1]; // Modify nums[i] to nums[i + 1]

} else {

nums[i + 1] = nums[i]; // Modify nums[i + 1] to nums[i]

}

}

}

return true;

}

}

**Explanation:**

Initialization: A count variable is used to keep track of the number of modifications needed.

Loop through the array: The loop iterates through the array, comparing each element with the next one.

Check for non-decreasing property violation: If nums[i] > nums[i + 1], the non-decreasing property is violated:

Increase the count.

If count becomes greater than 1, return false immediately since more than one modification would be needed.

Modification decision:

If possible, modify nums[i] to nums[i + 1] to fix the violation.

If modifying nums[i] isn't suitable, modify nums[i + 1] to nums[i].

Return **true** if only one or no modifications are needed after the loop.

This approach ensures that the solution runs efficiently with a time complexity of O(n), where n is the length of the input array.

[L300.Longest Increasing Subsequence](note://20C954956F6E4AA6A3575EA3146A1FCA)