<https://leetcode.com/problems/find-the-longest-valid-obstacle-course-at-each-position/description/>

You want to build some obstacle courses. You are given a 0-indexed integer array obstacles of length n, where obstacles[i] describes the height of the ith obstacle.

For every index i between 0 and n - 1 (inclusive), find the length of the longest obstacle course in obstacles such that:

You choose any number of obstacles between 0 and i inclusive.

You must include the ith obstacle in the course.

You must put the chosen obstacles in the same order as they appear in obstacles.

Every obstacle (except the first) is taller than or the same height as the obstacle immediately before it.

Return an array ans of length n, where ans[i] is the length of the longest obstacle course for index i as described above.

**Example 1:**

**Input:** obstacles = [1,2,3,2]

**Output:** [1,2,3,3]

**Explanation:** The longest valid obstacle course at each position is:

- i = 0: [1], [1] has length 1.

- i = 1: [1,2], [1,2] has length 2.

- i = 2: [1,2,3], [1,2,3] has length 3.

- i = 3: [1,2,3,2], [1,2,2] has length 3.

**Example 2:**

**Input:** obstacles = [2,2,1]

**Output:** [1,2,1]

**Explanation:** The longest valid obstacle course at each position is:

- i = 0: [2], [2] has length 1.

- i = 1: [2,2], [2,2] has length 2.

- i = 2: [2,2,1], [1] has length 1.

**Example 3:**

**Input:** obstacles = [3,1,5,6,4,2]

**Output:** [1,1,2,3,2,2]

**Explanation:** The longest valid obstacle course at each position is:

- i = 0: [3], [3] has length 1.

- i = 1: [3,1], [1] has length 1.

- i = 2: [3,1,5], [3,5] has length 2. [1,5] is also valid.

- i = 3: [3,1,5,6], [3,5,6] has length 3. [1,5,6] is also valid.

- i = 4: [3,1,5,6,4], [3,4] has length 2. [1,4] is also valid.

- i = 5: [3,1,5,6,4,2], [1,2] has length 2.

**Constraints:**

n == obstacles.length

1 <= n <= 10^5

1 <= obstacles[i] <= 10^7

**Attempt 1: 2024-09-14**

**Wrong Solution:**

**Below solution just try to leverage Leetcode 300. Longest Increasing Subsequence directly, but failed**

Input

obstacles = [2,2,1]

Output = [1,2,2]

Expected = [1,2,1]

**The reason is two requirements in problem statement:**

**Requirement 1: You must include the ith obstacle in the course.**

**Requirement 2: Every obstacle (except the first) is taller than or the same height as the obstacle immediately before it.**

Why It's Incorrect:

For the given input obstacles = [2, 2, 1], the wrong solution output is [1, 2, 2]. This is because the LIS up to each index is calculated as follows:

For index = 0, the subsequence is [2] → length 1.

For index = 1, the subsequence is [2, 2] → length 2.

**For index = 2, the subsequence is [2, 2, 1], which mistakenly gives length 2 based on pure LIS logic, since up to index 2, we still have index = 0 and 1 two numbers [2, 2] construct the LIS if not include index = 2 number as 1. However, check above requirement 1, we MUST include index = 2 number as 1, and if 1 is on the course, then index = 0 and 1 two numbers [2, 2] violate the requirement 2.**

class Solution {

    public int[] longestObstacleCourseAtEachPosition(int[] obstacles) {

        int len = obstacles.length;

        int[] result = new int[len];

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

            result[i] = lengthOfLIS(obstacles, i);

        }

        return result;

    }

    public int lengthOfLIS(int[] nums, int index) {

        int result = 1;

        int[] dp = new int[index + 1];

        Arrays.fill(dp, 1);

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

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

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

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

                }

            }

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

        }

        return result;

    }

}

**To fix above issue, we have to add one more condition make sure all comparisons MUST include the comparison with current obstacle (nums[j] <= nums[index]) on the course**

**By the way, we cannot remove current condition nums[j] <= nums[i] and only keep (nums[j] <= nums[index]) as below, test out by:**

Input

obstacles = [3,1,5,6,4,2]

Output = [1,1,3,4,3,2]

Expected = [1,1,2,3,2,2]

public int lengthOfLIS(int[] nums, int index) {

    int result = 1;

    int[] dp = new int[index + 1];

    Arrays.fill(dp, 1);

    for(int i = 0; i <= index; i++) {

        for(int j = 0; j < i; j++) {

// Remove nums[j] <= nums[i] is wrong

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

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

            }

        }

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

    }

    return result;

}

**Below is the correct fix include both conditions**

**Solution 1: Brute Force + Longest Increasing Subsequence**

**Style 1: Call LIS as helper method (10 min, TLE 57/78)**

class Solution {

    public int[] longestObstacleCourseAtEachPosition(int[] obstacles) {

        int len = obstacles.length;

        int[] result = new int[len];

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

            result[i] = lengthOfLIS(obstacles, i);

        }

        return result;

    }

    public int lengthOfLIS(int[] nums, int index) {

        int result = 1;

        int[] dp = new int[index + 1];

        Arrays.fill(dp, 1);

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

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

// Additional condition 'nums[j] <= nums[index]' to make sure

// always consider current obstacle on the course as the last

// element, and it must be no less than all previous choose

// elements on the course

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

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

                }

            }

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

        }

        return result;

    }

}

Time Complexity: O(N^3)

Space Complexity: O(N)

**Style 2: Brute Force + Longest Increasing Subsequence (30 min, purely based on Leetcode 300 solution, TLE 64/78)**

class Solution {

    public int[] longestObstacleCourseAtEachPosition(int[] obstacles) {

        int len = obstacles.length;

        int[] result = new int[len];

        // DP array to store the length of the valid course at each position

        int[] dp = new int[len];

        // Each obstacle alone forms a valid course, so initialize dp with 1

        Arrays.fill(dp, 1);

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

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

                // If we can extend the course from j to i

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

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

                }

            }

            // The result for position i is the value of dp[i]

            result[i] = dp[i];

        }

        return result;

    }

}

Time Complexity: O(N^2)

Space Complexity: O(N)

**Refer to chatGPT**

we can use dynamic programming to find the length of the longest obstacle course for each index i in the input array obstacles . We can iterate over all the obstacles from index 0 to index i and for each obstacle, we can update the length of the longest obstacle course seen so far for that obstacle. We can then update the answer array with the length of the longest obstacle course seen so far for the current obstacle, which is equal to the maximum length seen so far plus 1. By doing this, we can find the length of the longest obstacle course for each index i in the input array obstacles

**Dynamic programming (DP) solution** for LeetCode 1964 that does not involve binary search but still efficiently computes the longest valid obstacle course at each position.

**Approach:**

We use a **DP array** (dp[i]) where each element represents the length of the longest valid obstacle course ending at position i. For each position i, we compare it with all previous obstacles (j < i) and update dp[i] if the sequence is non-decreasing (obstacles[j] <= obstacles[i]).

**Plan:**

Initialize a DP array dp with all elements set to 1, since the minimum length of a valid obstacle course ending at any index is 1 (the obstacle itself).

For each position i, compare obstacles[i] with all previous obstacles (j < i):

If obstacles[j] <= obstacles[i], update dp[i] to dp[j] + 1, meaning that we can extend the valid obstacle course ending at j to i.

The result for each position is the value in dp[i].

**Time Complexity:**

The time complexity of this solution is O(n^2), where n is the number of obstacles. This is because we are using two nested loops to compare each pair of obstacles.

**Code:**

class Solution {

public int[] longestObstacleCourseAtEachPosition(int[] obstacles) {

int len = obstacles.length;

int[] dp = new int[len]; // DP array to store the length of the valid course at each position

int[] result = new int[len]; // Result array

Arrays.fill(dp, 1); // Each obstacle alone forms a valid course, so initialize dp with 1

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

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

// If we can extend the course from j to i (i.e., obstacles[j] <= obstacles[i])

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

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

}

}

result[i] = dp[i]; // The result for position i is the value of dp[i]

}

return result;

}

}

**Explanation:**

Step 1: We initialize the dp array with 1s because each obstacle on its own is a valid obstacle course.

Step 2: We loop through each obstacle i. For each obstacle i, we loop through all previous obstacles j where j < i.

If obstacles[j] <= obstacles[i], we check if extending the sequence from j to i gives a longer valid course. If so, we update dp[i].

Step 3: After checking all js for each i, the length of the longest valid obstacle course ending at position i is stored in dp[i], and we set result[i] = dp[i].

**Example Walkthrough:**

Let’s take the input obstacles = [2, 2, 1] and walk through how the DP array is updated:

Initial state:

dp = [1, 1, 1] (each obstacle alone forms a course of length 1)

result = [0, 0, 0] (empty result array)

Processing **i = 0**:

There's no j < 0, so we skip the inner loop.

dp[0] = 1 (course [2]), and we set result[0] = 1.

Processing **i = 1**:

Compare obstacles[1] = 2 with obstacles[0] = 2:

2 <= 2, so we update dp[1] = dp[0] + 1 = 2.

dp[1] = 2 (course [2, 2]), and we set result[1] = 2.

Processing **i = 2**:

Compare obstacles[2] = 1 with obstacles[0] = 2 and obstacles[1] = 2:

1 < 2, so no updates are made.

dp[2] = 1 (course [1]), and we set result[2] = 1.

Final result: [1, 2, 1].

**Output:**

For the input obstacles = [2, 2, 1], the output will be:

[1, 2, 1]

**And i have a Leecode 300 solution as below, can you compare above Style 2 solution and below Leecode 300 solution, what's the difference and why ?**

class Solution {

    public int lengthOfLIS(int[] nums) {

        int result = 1;

        int len = nums.length;

        int[] dp = new int[len];

        Arrays.fill(dp, 1);

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

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

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

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

                }

            }

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

        }

        return result;

    }

}

**Refer to chatGPT**

Let's compare the two solutions:

**Solution for LeetCode 300 (Longest Increasing Subsequence - LIS):**

The solution you provided for LeetCode 300 finds the **longest increasing subsequence (LIS)** in an array. Here's how it works:

DP Array

(dp[i]): Stores the length of the LIS ending at index i.

Two Loops:

Outer loop (i): Iterates through each element.

Inner loop (j): Compares the current element nums[i] with every previous element nums[j]. If nums[j] < nums[i], it checks if the sequence ending at j can be extended to i, and updates dp[i] accordingly.

Result: The longest increasing subsequence length is the maximum value in the dp array after processing the entire array.

**Solution for LeetCode 1964 (Longest Valid Obstacle Course at Each Position):**

In the dynamic programming (DP) solution I provided for LeetCode 1964, the task is to compute the longest **valid obstacle course** ending at each position in the obstacles array. The valid obstacle course allows non-decreasing sequences, so the solution must compute the longest subsequence where obstacles[j] <= obstacles[i] (instead of strictly less than as in LeetCode 300).

**Key Differences:**

Strict vs Non-decreasing Condition:

In **LeetCode 300** (LIS), the condition is strictly increasing (nums[j] < nums[i]), meaning that only smaller elements can contribute to extending the subsequence.

In **LeetCode 1964**, the condition is non-decreasing (obstacles[j] <= obstacles[i]), meaning that both smaller and equal elements can contribute to the obstacle course.

Result:

LeetCode 300 returns a single number — the length of the longest increasing subsequence in the entire array.

LeetCode 1964 returns an array, where each element represents the length of the longest valid obstacle course **ending at each position**.

Output Format:

LeetCode 300 provides a global result (the longest subsequence for the whole array).

LeetCode 1964 requires a local result at each index, meaning we need to track the longest sequence for each position in the array.

Problem Type:

LeetCode 300 is a global optimization problem, focusing on the overall longest subsequence.

LeetCode 1964 is a local problem, where you calculate the longest valid obstacle course up to each position.

Conditions in the DP Updates:

In **LeetCode 300**, the update happens if nums[j] < nums[i].

In **LeetCode 1964**, the update happens if obstacles[j] <= obstacles[i].

Time Complexity:

Both solutions have a time complexity of **O(n²)**, as both involve nested loops comparing each element with all previous elements.

Use of the **dp Array**:

In both problems, dp[i] represents the length of the longest subsequence ending at index i. The update mechanism is similar, but the condition for extending the subsequence differs (< for LIS, <= for obstacle course).

**Example Comparison:**

Let’s compare the two problems with an example.

**LeetCode 300** (LIS):

Input: nums = [2, 5, 1, 8, 3]

Output: The length of the longest increasing subsequence is 3 (subsequence: [2, 5, 8] or [2, 3, 8]).

**LeetCode 1964** (Obstacle Course):

Input: obstacles = [2, 5, 1, 8, 3]

Output: For each index i, we find the length of the longest non-decreasing subsequence ending at i:

At index 0 (2): [2] → 1

At index 1 (5): [2, 5] → 2

At index 2 (1): [1] → 1

At index 3 (8): [2, 5, 8] → 3

At index 4 (3): [2, 3] → 2

Result: [1, 2, 1, 3, 2]

**Conclusion:**

The solution for **LeetCode 300** (LIS) focuses on strictly increasing sequences, whereas the solution for **LeetCode 1964** focuses on non-decreasing sequences.

LeetCode 300 returns a single value for the entire array, while **LeetCode 1964** returns an array with the length of the longest obstacle course at each position.

The differences arise primarily from the problem requirements: strict increasing vs non-decreasing, and global vs local results. However, both use a similar dynamic programming approach with O(n²) time complexity.

**Solution 2: Longest Increasing Subsequence + Greedy + Binary Search (Find Upper Boundary) (180 min)**

class Solution {

    public int[] longestObstacleCourseAtEachPosition(int[] obstacles) {

        int len = obstacles.length;

        // dp[i] stores the length of the longest obstacle course ending at index i

        int[] dp = new int[len];

        // LIS\_heights stores the increasing subsequence of obstacle heights

        List<Integer> LIS\_heights = new ArrayList<>();

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

            int insert\_idx = binarySearch(LIS\_heights, obstacles[i]);

            // If insert\_idx is equal to the size of LIS\_heights, it means the current

            // obstacle is greater than all the obstacles encountered so far, so we

            // add it to the end of LIS\_heights.

            if(insert\_idx == LIS\_heights.size()) {

                LIS\_heights.add(obstacles[i]);

            // If insert\_idx is less than the size of LIS\_heights, it means the current

            // obstacle can replace an obstacle in the existing increasing subsequence.

            // We update the obstacle at index insert\_idx in LIS\_heights with the current

            // obstacle. (The update happens because we follow greedy thought try to make

            // all previous obstacles on the course as small as possible, which benefit

            // us to put more obstacles on this course)

            } else {

                LIS\_heights.set(insert\_idx, obstacles[i]);

            }

            // length of the obstacle course is the index plus one

            // (plus one to include current obstacle)

            dp[i] = insert\_idx + 1;

        }

        return dp;

    }

    private int binarySearch(List<Integer> LIS\_heights, int target) {

        int lo = 0;

        int hi = LIS\_heights.size() - 1;

        while(lo <= hi) {

            int mid = lo + (hi - lo) / 2;

            if(LIS\_heights.get(mid) > target) {

                hi = mid - 1;

            } else {

                lo = mid + 1;

            }

        }

        // 'lo - 1' means upper boundary, '+ 1' means insert index of current

        // 'target' value, the insert index of current obstacle should one

        // right position than the last target value position

        return lo - 1 + 1;

    }

}

Time Complexity: O(NlogN)

Space Complexity: O(N)

**Refer to**

<https://leetcode.com/problems/find-the-longest-valid-obstacle-course-at-each-position/solutions/3494753/java-solution-for-find-the-longest-valid-obstacle-course-at-each-position-problem/>

**Intuition**

The given solution uses the concept of the Longest Increasing Subsequence (LIS) to find the length of the longest obstacle course at each position.

**Approach**

Initialize an empty list lis to store the increasing subsequence of obstacle heights encountered so far.

Iterate through each obstacle in the input array.

**For each obstacle, find the index idx to insert it in the lis list using binary search (Find upper boundary since the insert index of current obstacle should one right position than the last target value position). The index idx represents the length of the longest increasing subsequence ending at the current obstacle.**

**If idx is equal to the size of lis, it means the current obstacle is greater than all the obstacles encountered so far, so we add it to the end of lis.**

**If idx is less than the size of lis, it means the current obstacle can replace an obstacle in the existing increasing subsequence. We update the obstacle at index idx in lis with the current obstacle. (The update happens because we follow greedy thought try to make all previous obstacles on the course as small as possible, which benefit us to put more obstacles on this course)**

Set dp[i] (where i is the current index) to idx + 1, representing the length of the longest obstacle course ending at the current index.

Repeat steps 3 to 6 for all obstacles.

Return the dp array, which contains the lengths of the longest obstacle courses at each position.

**Complexity**

Time complexity: The time complexity of the solution is O(n log n) due to the binary search operation performed for each obstacle.

Space complexity: The space complexity is O(n) because we use an additional dp array and the lis list to store intermediate results.

**Code**

class Solution {

    public int[] longestObstacleCourseAtEachPosition(int[] obstacles) {

        int n = obstacles.length;

        // dp[i] stores the length of the longest obstacle course ending at index i

        int[] dp = new int[n];

        // lis stores the increasing subsequence of obstacle heights

        List<Integer> lis = new ArrayList<>();

        for (int i = 0; i < n; i++) {

            int obstacle = obstacles[i];

            // find the index to insert the obstacle in the increasing subsequence

            int idx = binarySearch(lis, obstacle);

            if (idx == lis.size()) {

                lis.add(obstacle);

            } else {

                lis.set(idx, obstacle);

            }

            // length of the obstacle course is the index plus one

            dp[i] = idx + 1;

        }

        return dp;

    }

    private int binarySearch(List<Integer> lis, int target) {

        int left = 0;

        int right = lis.size() - 1;

        while (left <= right) {

            int mid = left + (right - left) / 2;

            if (lis.get(mid) > target) {

right = mid - 1;

            } else {

                left = mid + 1;

            }

        }

        return left;

    }

}

**Refer to**

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

[L704.Binary Search](note://484532A5D2CB4B98A28218674438D5D8)