<https://leetcode.com/problems/longest-increasing-subsequence/>

Given an integer array nums, return *the length of the longest* ***strictly increasing subsequence***

.

**Example 1:**

Input: nums = [10,9,2,5,3,7,101,18]

Output: 4

Explanation: The longest increasing subsequence is [2,3,7,101], therefore the length is 4.

**Example 2:**

Input: nums = [0,1,0,3,2,3]

Output: 4

**Example 3:**

Input: nums = [7,7,7,7,7,7,7]

Output: 1

**Constraints:**

1 <= nums.length <= 2500

-10^4 <= nums[i] <= 10^4

**Follow up:** Can you come up with an algorithm that runs in O(n log(n)) time complexity?

**Attempt 1: 2023-04-05**

**Solution 1.1: Native DFS - Divide and Conquer (10 min, TLE)**

**Style 1: 'prev' as actual value**

class Solution {

    public int lengthOfLIS(int[] nums) {

        return helper(nums, 0, -10001);

    }

    private int helper(int[] nums, int index, int prev) {

        if(index >= nums.length) {

            return 0;

        }

        int not\_take = helper(nums, index + 1, prev);

        int take = 0;

        if(nums[index] > prev) {

            take = 1 + helper(nums, index + 1, nums[index]);

        }

        return Math.max(not\_take, take);

    }

}

Time Complexity: O(2^N), where N is the size of nums. At each index, we have choice to either take or not take the element and we explore both ways. So, we 2 \* 2 \* 2...N times = O(2^N)

Space Complexity: O(N), max recursive stack depth.

**Style 2: 'prev' as index (inspired by DFS + Memoization as Solution 2)**

class Solution {

    public int lengthOfLIS(int[] nums) {

        return helper(nums, 0, -1);

    }

    private int helper(int[] nums, int index, int prev) {

        if(index >= nums.length) {

            return 0;

        }

        int not\_take = helper(nums, index + 1, prev);

        int take = 0;

        if(prev == -1 || nums[index] > nums[prev]) {

            take = 1 + helper(nums, index + 1, index);

        }

        return Math.max(not\_take, take);

    }

}

Time Complexity: O(2^N)

Space Complexity: O(N)

**Refer to**

<https://leetcode.com/problems/longest-increasing-subsequence/solutions/1326552/optimization-from-brute-force-to-dynamic-programming-explained/>

❌ ***Solution - I (Brute-Force)***

We need to find maximum increasing subsequence length. In the brute-force approach, we can model this problem as -

If the current element is greater than the previous element, then we can either pick it or don't pick it because we may get a smaller element somewhere ahead which is greater than previous and picking that would be optimal. So we try both options.

If the current element is smaller or equal to previous element, it can't be picked.

class Solution {

public:

    int lengthOfLIS(vector<int>& nums, int i = 0, int prev = INT\_MIN) {

        if(i == size(nums)) return 0;

        return max(lengthOfLIS(nums, i + 1, prev), (nums[i] > prev) + lengthOfLIS(nums, i + 1, max(nums[i], prev)));

    }

};

A better and more understandable way of writing the same code as above -

class Solution {

public:

    int lengthOfLIS(vector<int>& nums) {

        return solve(nums, 0, INT\_MIN);

    }

    int solve(vector<int>& nums, int i, int prev) {

        if(i >= size(nums)) return 0;                                // cant pick any more elements

        int take = 0, dontTake = solve(nums, i + 1, prev);          // try skipping the current element

        if(nums[i] > prev) take = 1 + solve(nums, i + 1, nums[i]);  // or pick it if it is greater than previous picked element

        return max(take, dontTake);                                  // return whichever choice gives max LIS

    }

};

***Time Complexity :*** **O(2^N)**, where N is the size of *nums*. At each index, we have choice to either take or not take the element and we explore both ways. So, we 2 \* 2 \* 2...N times = O(2^N)

***Space Complexity :*** **O(N)**, max recursive stack depth.

**Solution 1.2: Native DFS - For loop inside recursion method (10 min, TLE)**

class Solution {

    public int lengthOfLIS(int[] nums) {

        int maxLength = 0;

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

            // Start a DFS from each element in the array

            maxLength = Math.max(maxLength, helper(nums, i));

        }

        return maxLength;

    }

    private int helper(int[] nums, int i) {

        int maxLength = 1;  // Minimum length is the element itself

        // Explore all previous elements to find an increasing subsequence

        for (int j = 0; j < i; j++) {

            if (nums[j] < nums[i]) {

                // If current element can be part of increasing sequence,

                // recursively find LIS including it

                maxLength = Math.max(maxLength, 1 + helper(nums, j));

            }

        }

        return maxLength;

    }

}

Time Complexity: O(2^N)

Space Complexity: O(N)

**Refer to chatGPT**

**Instead of 'taken' and 'notTaken' solution, can we do with for loop inside recursion method ?**

We can certainly implement the pure DFS solution for LeetCode 300 using a for loop inside the recursion method, similar to how the inner loop operates in the dynamic programming (DP) solution. This approach will still be recursive but will explicitly loop through previous elements, much like the DP solution does with its nested loops.

**How It Works**

In this modified DFS approach, you will:

Use a recursive function to explore each element's possibilities.

Use a for loop inside the recursive function to check all elements before the current element (similar to the DP solution).

Track the longest increasing subsequence length for each position using recursive calls.

**Native DFS with a for Loop Inside Recursion**

Below is the code for the native DFS solution using a for loop inside the recursive method:

class Solution {

public int lengthOfLIS(int[] nums) {

int maxLength = 0;

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

// Start a DFS from each element in the array

maxLength = Math.max(maxLength, dfs(nums, i));

}

return maxLength;

}

private int dfs(int[] nums, int currentIndex) {

int maxLength = 1; // Minimum length is the element itself

// Explore all previous elements to find an increasing subsequence

for (int previousIndex = 0; previousIndex < currentIndex; previousIndex++) {

if (nums[previousIndex] < nums[currentIndex]) {

// If current element can be part of increasing sequence,

// recursively find LIS including it

maxLength = Math.max(maxLength, 1 + dfs(nums, previousIndex));

}

}

return maxLength;

}

}

**Explanation**

Initial Loop: In the lengthOfLIS method, we loop through each element of the array and initiate a DFS from each position. This ensures that we consider all possible subsequences starting from each element.

Recursive Function (**dfs)**:

Parameter: currentIndex denotes the current position in the array being explored.

**Base Case: The base case is implicitly handled since the for loop inside the dfs method will not execute if there are no preceding elements.**

Loop Inside Recursion: The for loop iterates over all previous indices (previousIndex < currentIndex). This is similar to the inner loop in the DP solution. It checks if nums[previousIndex] < nums[currentIndex], which means the current element can extend an increasing subsequence that ends with the element at previousIndex.

Recursive Call: If the condition is satisfied, it makes a recursive call to continue finding the longest subsequence that includes nums[previousIndex]. It adds 1 to the result of the recursive call to account for including nums[currentIndex].

Updating **maxLength**: The maxLength is updated to store the maximum length of any increasing subsequence found by including or excluding the current element.

Return Value: The main function lengthOfLIS collects the maximum value of maxLength for each starting point to get the overall LIS.

**Key Points**

DFS with Loop: The for loop inside the dfs method mimics the inner for loop in the DP solution. It allows the function to explore all increasing subsequences ending at each index.

Recursive Depth: Each recursive call explores potential subsequences, similar to how a nested loop explores them in the DP solution.

**Efficiency**: This pure DFS with an inner loop is conceptually straightforward but has exponential time complexity O(2^n), as it explores all possible subsequences. It is less efficient than the O(n^2) DP approach or the O(nlogn) optimized DP with binary search.

**Conclusion**

Using a for loop inside the recursion method is a direct and intuitive way to translate the nested loop structure of the DP solution into a DFS-based approach. While it retains the simplicity of recursion, it does so at the cost of efficiency. For large input sizes, this approach would be less practical compared to dynamic programming methods that optimize redundant calculations.

**Is it necessary to separately write a DFS helper method for below solution ? since its not a standard DFS pattern in helper method, it not contains base condition which should directly return**

In the provided solution, the helper method does not exhibit a typical depth-first search (DFS) recursion pattern with a base condition that directly returns. This might seem unusual for a DFS method, but the design is deliberate and serves the specific purpose of solving the Longest Increasing Subsequence (LIS) problem by recursively finding the LIS ending at each index. Let's explore why the helper method is implemented this way and whether it's necessary to separate it as a helper method.

**Understanding the Current Approach**

**DFS Pattern in the** **helper Method**:

The helper method is called for each element i in the array from the lengthOfLIS method.

Inside the helper, the method looks at all previous elements (from 0 to i-1) to check if they are less than nums[i].

If nums[j] < nums[i], it recursively calls helper(nums, j) to compute the LIS that ends at j and tries to extend it by including nums[i].

**Base Condition:**

**The method doesn't explicitly check for out-of-bound conditions or a standard base case like returning 0 directly. Instead, the for loop handles the boundaries implicitly:**

**The loop for (int j = 0; j < i; j++) automatically exits if i is 0, meaning no recursive call is made.**

**The recursion implicitly ends when it reaches the smallest subsequence (a single element) since no previous element will satisfy j < i.**

**Recursive Computation:**

The helper method essentially computes the LIS ending at i by combining results of LISs ending at all previous indices where the subsequence can be extended.

It accumulates results through recursion and maximization.

**Is It Necessary to Have a Separate helper Method?**

Clarity and Modularity: Having a separate helper method makes the solution more modular and easier to understand. It explicitly separates the logic of computing the LIS for a given index, encapsulating the recursive behavior.

Direct Translation from DP Approach: This method mimics the transition from a dynamic programming approach (with nested loops) to a recursive one. It retains the structure and readability.

Code Organization: By moving the logic into a helper method, the main function lengthOfLIS remains clean and focused on initialization and aggregation of results.

**Why the Absence of a Standard Base Condition is Acceptable**

**Implicit Base Case: The method avoids going out of bounds by using a loop (for (int j = 0; j < i; j++)). This loop naturally handles the boundary checks:**

**When i = 0, the loop doesn't run, and the default maxLength = 1 is returned, indicating the minimum LIS is the element itself.**

Single-Element Subsequence: The smallest subsequence considered is always the element itself, handled by initializing maxLength = 1.

Avoiding Redundant Returns: There's no need to explicitly check if i is out of bounds or zero since these conditions are naturally managed by the loop structure and the starting value of maxLength.

**Conclusion**

Not Necessary but Beneficial: It’s not strictly necessary to have a separate helper method. The logic could be incorporated directly into the lengthOfLIS method with nested loops, but that would make the code less modular and harder to maintain.

Design Choice: The use of a helper method is a design choice that enhances code clarity and modularity, aligning with good coding practices. It may not strictly follow the typical DFS recursion patterns with explicit base conditions because the problem's nature and constraints are inherently handled by the design of the recursive calls.

**Overall**, while not a conventional DFS with an explicit base condition, the structure effectively solves the problem and remains clear and maintainable. Thus, keeping a separate helper method is reasonable and aligns with the intent to keep code modular and expressive.

**Refer to**

<https://leetcode.com/problems/longest-increasing-subsequence/solutions/1124603/top-down-backtracking-memoization/>

**Are above two (1.1 & 1.2) Native DFS solutions equal ?**

**Refer to chatGPT**

The two pure DFS solutions for finding the length of the longest increasing subsequence (LIS) provided above are conceptually tackling the same problem, but they are not strictly equivalent in terms of their execution flow. Let's delve into how each approach works, their similarities, and differences.

**Comparing the Two DFS Solutions**

**1. First Solution (Using for Loop Inside Recursion)**

How It Works:

This solution starts a DFS from each element i in the array to calculate the longest increasing subsequence ending at that position.

It uses a for loop to compare the current element nums[i] with all previous elements (nums[j] where j < i).

If nums[j] < nums[i], it considers including nums[i] in the subsequence by making a recursive call helper(nums, j) to find the LIS ending at j.

Characteristics:

It explicitly checks all previous elements in a nested manner.

It only starts new subsequences from each element and looks backward.

The recursion depth is bounded by the length of the array.

This approach simulates the DP nested loops.

**2. Second Solution (Using take and not\_take Decisions)**

How It Works:

This solution uses a top-down recursive approach starting from the beginning of the array.

It makes two choices at each step:

Not take: Skip the current element and move to the next index.

Take: Include the current element in the subsequence if it is greater than the prev element in the sequence.

It passes prev as a parameter to keep track of the last included element in the current subsequence.

Characteristics:

It explores subsequences in a depth-first manner for each element, considering both the inclusion and exclusion of the current element.

This approach can explore all possible subsequences.

The recursion depth can go up to the length of the array.

**Similarities**

Purpose: Both methods aim to find the longest increasing subsequence in the array.

Recursive Nature: Both use recursion to explore different possibilities of subsequences.

No DP Optimization: Both solutions are pure DFS without memoization, meaning they explore the same subsequences multiple times, resulting in exponential time complexity O(2^n)

Base Case: Both methods use base cases that return when no more elements can be considered for forming a subsequence.

**Differences**

**Traversal Direction:**

**First Solution: Starts from each element and looks backward, only considering elements before the current index (i). It explores subsequences ending at i.**

**Second Solution: Starts from the beginning and makes decisions at each step about whether to include the current element (index). It considers the entire array in both directions.**

State Representation:

First Solution: The state is represented by i, the current index, and all recursive calls are based on comparing current and previous elements through a loop.

Second Solution: The state is represented by index (current index) and prev (the last taken element). It directly uses two recursive calls to explore subsequences (take or not\_take).

Flexibility and Control:

First Solution: More controlled, checks each element against all previous ones explicitly. It directly maps to the DP solution's behavior.

Second Solution: More dynamic, exploring each element's two possibilities, allowing for more straightforward inclusion and exclusion logic.

Implementation Simplicity:

First Solution: May feel more intuitive for those familiar with DP approaches due to its explicit checks.

Second Solution: More concise and closer to a backtracking-style approach.

**Are They Equal?**

Theoretically: Both approaches are designed to find the LIS, and they explore overlapping sets of subsequences. Given enough time and without optimization (memoization), they will yield the same result.

**Practically: Their execution patterns differ, which can affect performance. The second solution might explore some subsequences that the first solution would not because it actively makes take/not\_take decisions without being bound by only considering past indices. It checks every possible subsequence combination in a more explicit form, which the first solution indirectly does by checking each previous element.**

**Conclusion**

Both solutions effectively solve the same problem and will output the same result for any input.

They differ in terms of the recursive approach: one uses nested checks (similar to DP), and the other explicitly handles subsequence decisions.

Without optimization, both have exponential time complexity and are not suitable for large input sizes. However, they demonstrate different ways of thinking about the problem, providing insights into converting DP concepts into recursive solutions.

**Solution 2.1: DFS + Divide and Conquer + Memoization (10 min)**

**Style 1: Second dimension based on actual value of 'prev' (MLE)**

class Solution {

    public int lengthOfLIS(int[] nums) {

        // Why define second demension as 20001 ?

        // Because -10^4 <= nums[i] <= 10^4 makes potential minimum

        // element value will be -10^4, we have to balance it with

        // 10^4 + 1, plus potential maximum element value as 10^4,

        // total will be 10^4 + (10^4 + 1) = 2 \* 10^4 + 1

        Integer[][] memo = new Integer[nums.length + 1][20001];

        return helper(nums, 0, -10001, memo);

    }

    private int helper(int[] nums, int index, int prev, Integer[][] memo) {

        if(index >= nums.length) {

            return 0;

        }

        // Why 'prev + 10001' ?

        // Since -10^4 <= nums[i] <= 10^4, for handle extreme smallest 'prev'

        // as -10^4 - 1 = -10001, have to shift 10001 indexes to make index

        // start from 0, that's how 'prev + 10001' comes

        if(memo[index][prev + 10001] != null) {

            return memo[index][prev + 10001];

        }

        int not\_take = helper(nums, index + 1, prev, memo);

        int take = 0;

        if(nums[index] > prev) {

            take = 1 + helper(nums, index + 1, nums[index], memo);

        }

        memo[index][prev + 10001] = Math.max(not\_take, take);

        return memo[index][prev + 10001];

    }

}

Time Complexity : O(N^2)

Space Complexity : O(N^2)

**Style 2: Second dimension based on index of 'prev'**

class Solution {

    public int lengthOfLIS(int[] nums) {

        Integer[][] memo = new Integer[nums.length + 1][nums.length + 1];

        return helper(nums, 0, -1, memo);

    }

    private int helper(int[] nums, int index, int prev\_index, Integer[][] memo) {

        if(index >= nums.length) {

            return 0;

        }

        if(memo[index][prev\_index + 1] != null) {

            return memo[index][prev\_index + 1];

        }

        int not\_take = helper(nums, index + 1, prev\_index, memo);

        int take = 0;

        if(prev\_index == -1 || nums[index] > nums[prev\_index]) {

            take = 1 + helper(nums, index + 1, index, memo);

        }

        memo[index][prev\_index + 1] = Math.max(not\_take, take);

        return memo[index][prev\_index + 1];

    }

}

Time Complexity : O(N^2)

Space Complexity : O(N^2)

**Refer to**

<https://leetcode.com/problems/longest-increasing-subsequence/solutions/1326552/optimization-from-brute-force-to-dynamic-programming-explained/>

✔️ ***Solution - II (Dynamic Programming - Memoization)***

There are many unnecessary repeated calculations in the brute-force approach. We can observe that the length of increasing subsequence starting at ith element with previously picked element prev will always be the same. So we can use **dynamic programming** to store the results for this state and reuse again in the future.

But it wouldn't be scalable to store the state as (i, prev) because prev element can be any number in [-104, 104]meaning we would need to declare a matrix dp[n][1e8] which won't be possible

**DP with (i, prev) as state which will cause MLE(Memory Limit Exceed)**

class Solution {

public:

    vector<unordered\_map<int, int>> dp;

    int lengthOfLIS(vector<int>& nums) {

        dp.resize(size(nums));

        return solve(nums, 0, INT\_MIN);

    }

    int solve(vector<int>& nums, int i, int prev) {

        if(i >= size(nums)) return 0;

        if(dp[i].count(prev)) return dp[i][prev];

        int take = 0, dontTake = solve(nums, i + 1, prev);

        if(nums[i] > prev) take = 1 + solve(nums, i + 1, nums[i]);

        return dp[i][prev] = max(take, dontTake);

    }

};

Instead, we could store the state of (i, prev\_i), where prev\_i denotes the index of previous chosen element. Thus we would use a dp matrix where dp[i][j] will denote the longest increasing subsequence from index i when previous chosen element's index is j.

class Solution {

public:

    vector<vector<int>> dp;

    int lengthOfLIS(vector<int>& nums) {

        dp.resize(size(nums), vector<int>(1+size(nums), -1));  // dp[i][j] denotes max LIS starting from i when nums[j] is previous picked element

        return solve(nums, 0, -1);

    }

    int solve(vector<int>& nums, int i, int prev\_i) {

        if(i >= size(nums)) return 0;

        if(dp[i][prev\_i+1] != -1) return dp[i][prev\_i+1];

        int take = 0, dontTake = solve(nums, i + 1, prev\_i);

        if(prev\_i == -1 || nums[i] > nums[prev\_i]) take = 1 + solve(nums, i + 1, i); // try picking current element if no previous element is chosen or current > nums[prev\_i]

        return dp[i][prev\_i+1] = max(take, dontTake);

    }

};

***Time Complexity :*** **O(N^2)**

***Space Complexity :*** **O(N^2)**

Depending on the mood of OJ, it may decide to accept your solution or give TLE for the above solution.

**Solution 2.2: DFS + For loop inside recursion method + Memoization (10 min)**

class Solution {

    public int lengthOfLIS(int[] nums) {

        int maxLength = 0;

        Integer[] memo = new Integer[nums.length];

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

            // Start a DFS from each element in the array

            maxLength = Math.max(maxLength, helper(nums, i, memo));

        }

        return maxLength;

    }

    private int helper(int[] nums, int i, Integer[] memo) {

        int maxLength = 1;  // Minimum length is the element itself

        if(memo[i] != null) {

            return memo[i];

        }

        // Explore all previous elements to find an increasing subsequence

        for (int j = 0; j < i; j++) {

            if (nums[j] < nums[i]) {

                // If current element can be part of increasing sequence,

                // recursively find LIS including it

                maxLength = Math.max(maxLength, 1 + helper(nums, j, memo));

            }

        }

        return memo[i] = maxLength;

    }

}

**Refer to**

<https://leetcode.com/problems/longest-increasing-subsequence/solutions/1124603/top-down-backtracking-memoization/>

Idea :-

What is the longest increasing subsequence starting from each index ?

Ex :-

0 1 2 3 4.5

[5,2,3,1,7,4]

MAX(lis(0), lis(1),list(2),list(3) ....list(5)) is the answer

private static int lengthOfLIS(int [] input) {

int max = Integer.MIN\_VALUE;

int [] cache = new int [input.length];

for (int i = 0; i < input.length; ++i) {

max = Math.max(max, lengthOfLIS(input,i, cache));

}

return Math.max(1,max);

}

private static int lengthOfLIS(int[] input, int start, int [] cache) {

if (start >= input.length) return 0;

if (cache[start] != 0) return cache[start];

int max = Integer.MIN\_VALUE;

for (int i = start; i < input.length; ++i) {

if (input[i] > input[start]) {

max = Math.max(max,1 + lengthOfLIS(input, i, cache));

}

}

cache[start] = Math.max(max,1);

return cache[start];

}

**Solution 3: DFS + Memoization + Space Optimized (120 min, too hard to come with)**

class Solution {

    public int lengthOfLIS(int[] nums) {

        // We can do better and further reduce the state stored using DP.

        // It's redundant to store states for all i having prev as its

        // previous element index. The length will always be greatest for

        // the state (prev, prev) since no more elements before prev can

        // be taken. So we can just use a linear DP where dp[i] denotes

        // the LIS starting at index i

        Integer[] memo = new Integer[nums.length + 1];

        return helper(nums, 0, -1, memo);

    }

    private int helper(int[] nums, int index, int prev\_index, Integer[] memo) {

        if(index >= nums.length) {

            return 0;

        }

        if(memo[prev\_index + 1] != null) {

            return memo[prev\_index + 1];

        }

        int not\_take = helper(nums, index + 1, prev\_index, memo);

        int take = 0;

        if(prev\_index == -1 || nums[index] > nums[prev\_index]) {

            take = 1 + helper(nums, index + 1, index, memo);

        }

        memo[prev\_index + 1] = Math.max(not\_take, take);

        return memo[prev\_index + 1];

    }

}

Time Complexity : O(N^2)

Space Complexity : O(N)

**Refer to**

<https://leetcode.com/problems/longest-increasing-subsequence/solutions/1326554/longest-increasing-subsequence-optimization-from-brute-force-to-dp-explained/>

✔️ ***Solution - III (DP - Memoization - Space Optimized)***

We can do better and further reduce the state stored using DP. It's redundant to store states for all i having prev as its previous element index. **The length will always be greatest for the state (prev, prev) since no more elements before prev can be taken. So we can just use a linear DP where dp[i] denotes the LIS starting at index i**

class Solution {

public:

    vector<int> dp;

    int lengthOfLIS(vector<int>& nums) {

        dp.resize(size(nums)+1, -1);

        return solve(nums, 0, -1);

    }

    int solve(vector<int>& nums, int i, int prev\_i) {

        if(i >= size(nums)) return 0;

        if(dp[prev\_i+1] != -1) return dp[prev\_i+1];

        int take = 0, dontTake = solve(nums, i + 1, prev\_i);

        if(prev\_i == -1 || nums[i] > nums[prev\_i])

            take = 1 + solve(nums, i + 1, i);

        return dp[prev\_i+1] = max(take, dontTake);

    }

};

***Time Complexity :*** **O(N^2)**

***Space Complexity :*** **O(N)**

**Solution 4: 2D DP (60 min)**

**Style 1: Native DFS - Divide and Conquer upgrade to 2D DP**

class Solution {

public int lengthOfLIS(int[] nums) {

int len = nums.length;

// DP table where dp[i][j] means LIS starting at index i with previous index j

// We use len + 1 for the second dimension to handle the case where j is -1

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

// Fill DP table with loops, scanning from end to start

for (int i = len - 1; i >= 0; i--) {

for (int j = i - 1; j >= -1; j--) {

int not\_take = dp[i + 1][j + 1]; // Not taking the current element

int take = 0;

// Take the current element if it's greater than the previous element

if (j == -1 || nums[i] > nums[j]) {

take = 1 + dp[i + 1][i + 1];

}

// Store the maximum of taking or not taking the current element

dp[i][j + 1] = Math.max(not\_take, take);

}

}

// The result is stored in dp[0][0] because initially we haven't taken any elements

return dp[0][0];

}

}

**Refer to chatGPT**

To convert the given pure DFS solution into a dynamic programming (DP) solution while scanning the outer loop from the end of the array to the beginning, we need to use a DP table to store the results of subproblems. This way, we can avoid redundant computations by reusing the results from previous calculations.

**Original DFS Solution**

class Solution {

public int lengthOfLIS(int[] nums) {

return helper(nums, 0, -1);

}

private int helper(int[] nums, int index, int prev) {

if (index >= nums.length) {

return 0;

}

int not\_take = helper(nums, index + 1, prev);

int take = 0;

if (prev == -1 || nums[index] > nums[prev]) {

take = 1 + helper(nums, index + 1, index);

}

return Math.max(not\_take, take);

}

}

**Converting to a DP Solution**

**Create a DP table: We'll create a 2D DP array dp[i][j] where i represents the current index in the array and j represents the index of the previous element in the LIS (we'll use an offset since j can be -1).**

Initialization: We'll initialize the DP table with default values (such as 0).

Filling the DP table: We'll fill the DP table using nested loops, where the outer loop scans from the end of the array to the beginning.

Use index tracking: Instead of using prev as a direct parameter, we'll use it to track indices in the array, allowing us to simulate the recursive calls with iterative DP.

**DP Solution Implementation**

Here's how we can implement the DP solution:

class Solution {

public int lengthOfLIS(int[] nums) {

int len = nums.length;

// DP table where dp[i][j] means LIS starting at index i with previous index j

// We use len + 1 for the second dimension to handle the case where prev is -1

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

// Fill DP table with loops, scanning from end to start

for (int i = len - 1; i >= 0; i--) {

for (int prev = i - 1; prev >= -1; prev--) {

int not\_take = dp[i + 1][prev + 1]; // Not taking the current element

int take = 0;

// Take the current element if it's greater than the previous element

if (prev == -1 || nums[i] > nums[prev]) {

take = 1 + dp[i + 1][i + 1];

}

// Store the maximum of taking or not taking the current element

dp[i][prev + 1] = Math.max(not\_take, take);

}

}

// The result is stored in dp[0][0] because initially we haven't taken any elements

return dp[0][0];

}

}

**Explanation of the DP Solution**

DP Table Setup: We use a 2D dp table:

dp[i][prev + 1] holds the length of the longest increasing subsequence starting from index i with prev as the previous element. We use prev + 1 to handle the case where prev is -1.

Outer Loop (**i from len - 1 to 0):** We scan from the end of the array to the start. This ensures that when we calculate the value of dp[i][prev + 1], all future states (dp[i + 1][...]) are already computed. **(顶底之术)**

Inner Loop (**prev from i - 1 to -1):** **This loop checks all possible previous elements. The value of prev starts from i - 1 and goes down to -1, which indicates that no element has been chosen yet.**

Decision Making:

not\_take: This is the value if we do not include nums[i] in our current subsequence.

take: This is the value if we include nums[i] in our current subsequence. We only consider this if nums[i] > nums[prev] or if prev == -1 (no previous element).

Return Value: The final answer, representing the length of the longest increasing subsequence, is stored in dp[0][0]. **(顶底之术)**

**Conclusion**

This approach efficiently converts the DFS solution into a DP solution by replacing recursive calls with iterative table updates.

It maintains the problem's original logic while using a 2D DP table to avoid redundant calculations.

The outer loop scanning from the end to the beginning ensures that each state is computed only after all dependent states are known, simulating the recursive call structure but in a bottom-up manner.

**Style 2: Native DFS - For loop inside recursion method upgrade to 2D DP**

**其实这个2D DP有点勉强，严格来说只有内层for循环能应用顶底之术，因为在原 Native DFS - For loop inside recursion method 的解法中外层for循环并不适用于标准意义上的顶底之术，因为这就是一个纯粹的多端DP入口才必须使用的for循环，代表多个不同的开始，所以外层没必要进入顶底之术**

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

// Start a DFS from each element in the array

maxLength = Math.max(maxLength, helper(nums, i));

}

**抛去外层for循环以后实际helper方法内部的for循环中顶是j，底是0 (每次进入下一层以后 j 就变成了for循环的 i，而且每次从 0 开始循环到 i，所以 j 作为顶，0 作为底)**

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

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

// If current element can be part of increasing sequence,

// recursively find LIS including it

maxLength = Math.max(maxLength, 1 + helper(nums, j));

}

}

**最终解法 Refer to chatGPT**

class Solution {

    public int lengthOfLIS(int[] nums) {

        int n = nums.length;

        if (n == 0) return 0;

        int[][] dp = new int[n][n];

        // Initialize the diagonal elements, as each single element is an LIS of length 1.

        for (int i = 0; i < n; i++) {

            dp[i][i] = 1;

        }

        int maxLength = 1;

        // Fill dp table

// 外层依然沿用 i for loop from 1 to n - 1, 因为外层for循环并不适用于标准意义上的顶底之术，因为

// 这就是一个纯粹的多端DP入口才必须使用的for循环，代表多个不同的开始，所以外层没必要进入顶底之术

        for (int i = 1; i < n; i++) {

// 抛去外层for循环以后实际helper方法内部的for循环中顶是j，底是0 (每次进入下一层以后 j 就变成

// 了for循环的 i，而且每次从 0 开始循环到 i，所以 j 作为顶，0 作为底)

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

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

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

                }

            }

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

        }

        return maxLength;

    }

}

**Solution 5: 1D DP (60 min)**

**Style 1: Native DFS - Divide and Conquer upgrade to 1D DP**

class Solution {

public int lengthOfLIS(int[] nums) {

int len = nums.length;

if (len == 0) return 0;

// 1D DP array to store the maximum length of LIS ending at each position

int[] dp = new int[len];

// Initialize all values of dp array to 1 (each element is a subsequence of length 1)

Arrays.fill(dp, 1);

int maxLength = 1;

// Outer for loop direction no need change (compare to 2D DP's right to left)

for (int i = len - 1; i >= 0; i--) {

// Inner for loop direction change from left to right (compare to 2D DP solution's right to left)

for (int j = i + 1; j < len; j++) {

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

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

}

}

// Update the global maximum length

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

}

return maxLength;

}

}

**Refer to chatGPT**

**To convert the given 2D DP solution into a 1D DP solution without using a temporary DP array, we can utilize a single DP array that represents the longest increasing subsequence (LIS) ending at each position. The idea is to update the DP array in such a way that we avoid overwriting values that are yet to be used in the same iteration.**

**Key Changes**

Use a 1D DP Array: Instead of maintaining a 2D array, we maintain a 1D array where **dp[i] represents the LIS ending at the current index i.**

**Reverse Iteration: By iterating from the end of the array to the start, we ensure that the necessary values are updated in a way that avoids overwriting before they are used.**

**1D DP Solution Implementation**

Here’s the converted solution:

class Solution {

public int lengthOfLIS(int[] nums) {

int len = nums.length;

if (len == 0) return 0;

// 1D DP array to store the maximum length of LIS ending at each position

int[] dp = new int[len];

// Initialize all values of dp array to 1 (each element is a subsequence of length 1)

Arrays.fill(dp, 1);

int maxLength = 1;

// Outer for loop direction no need change (compare to 2D DP's right to left)

for (int i = len - 1; i >= 0; i--) {

// Inner for loop direction change from left to right (compare to 2D DP solution's right to left)

for (int j = i + 1; j < len; j++) {

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

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

}

}

// Update the global maximum length

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

}

return maxLength;

}

}

**Explanation**

Initialization:

We initialize the dp array with 1 because each element alone is an increasing subsequence of length 1.

The maxLength variable keeps track of the longest increasing subsequence found during iteration.

Filling the DP Array:

The outer loop iterates from the last element to the first element.

**The inner loop checks all subsequent elements (j > i) to see if they can be appended to the current subsequence ending at nums[i].**

If nums[j] > nums[i], it means nums[j] can extend the LIS ending at nums[i]. We update dp[i] by considering the length of dp[j] plus one more element (nums[i]).

Avoiding Overwriting:

By iterating from right to left, we ensure that when we update dp[i], all values of dp[j] (where j > i) have already been computed correctly, which avoids overwriting any needed values.

Result:

The maxLength variable stores the maximum value found in the dp array, representing the length of the longest increasing subsequence in the given nums array.

**Conclusion**

This conversion effectively demonstrates how to transform a 2D DP solution into a 1D DP solution while maintaining the logic of the original approach. The solution scans from end to start to avoid overwriting and thus eliminates the need for an additional temporary DP array. The space complexity is reduced to O(n), making it more efficient compared to the original 2D DP solution with O(n^2) space complexity.

**Style 2: Classic 1D DP**

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)

**Refer to**

<https://segmentfault.com/a/1190000003819886>

思路由于这个最长上升序列不一定是连续的，对于每一个新加入的数，都有可能跟前面的序列构成一个较长的上升序列，或者跟后面的序列构成一个较长的上升序列。比如

1,3,5,2,8,4,6，对于6来说，可以构成1,3,5,6，也可以构成2,4,6。因为前面那个序列长为4，后面的长为3，所以我们更愿意6组成那个长为4的序列，所以对于6来说，它组成序列的长度，实际上是之前最长一个升序序列长度加1，注意这个最长的序列的末尾是要小于6的，不然我们就把1,3,5,8,6这样的序列给算进来了。这样，我们的递推关系就隐约出来了，**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。**

**代码**

public class Solution {

    public int longestIncreasingSubsequence(int[] nums) {

        // write your code here

        if(nums.length == 0){

            return 0;

        }

        // 构建最长升序序列长度的数组

        int[] lis = new int[nums.length];

        lis[0] = 1;

        int max = 0;

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

            // 找到dp[0]到dp[i-1]中最大的升序序列长度且nums[j]<nums[i]

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

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

                    lis[i] = Math.max(lis[i], lis[j]);

                }

            }

            // 加1就是该位置能构成的最长升序序列长度

            lis[i] += 1;

            // 更新全局长度

            max = Math.max(max, lis[i]);

        }

        return max;

    }

}

**比较好理解的版本**

public class Solution {

    public int longestIncreasingSubsequence(int[] nums) {

        if(nums.length == 0){

            return 0;

        }

        int[] lis = new int[nums.length];

        int max = 0;

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

            int localMax = 0;

            // 找出当前点之前的最大上升序列长度

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

                if (lis[j] > localMax && nums[j] <= nums[i]){

                    localMax = lis[j];

                }

            }

            // 当前点则是该局部最大上升长度加1

            lis[i] = localMax + 1;

            // 用当前点的长度更新全局最大长度

            max = Math.max(max, lis[i]);

        }

        return max;

    }

}

**Refer to**

<https://leetcode.com/problems/longest-increasing-subsequence/solutions/1326554/longest-increasing-subsequence-optimization-from-brute-force-to-dp-explained>

✔️ ***Solution - IV (Dynamic Programming - Tabulation)***

We can solve it iteratively as well. Here, we use dp array where dp[i] denotes the LIS ending at index i. We can always pick a single element and hence all dp[i] will be initialized to 1.

**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.**

class Solution {

public:

    int lengthOfLIS(vector<int>& nums) {

        int ans = 1, n = size(nums);

        vector<int> dp(n, 1);

        for(int i = 0; i < n; i++)

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

                if(nums[i] > nums[j])

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

        return ans;

    }

};

***Time Complexity :*** **O(N^2)**

***Space Complexity :*** **O(N)**

**Solution 6: Binary Search (120 min)**

class Solution {

    public int lengthOfLIS(int[] nums) {

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

        for(int cur : nums) {

            if(list.size() == 0 || list.get(list.size() - 1) < cur) {

                list.add(cur);

            } else {

                int index = binarySearch(list, cur);

                list.set(index, cur);

            }

        }

        return list.size();

    }

    // Find the index of first element larger than 'target'

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

        int start = 0;

        int end = list.size() - 1;

        while(start <= end) {

            int mid = start + (end - start) / 2;

            if(list.get(mid) >= target) {

                end = mid - 1;

            } else {

                start = mid + 1;

            }

        }

        return start;

    }

}

Time Complexity: O(N \* logN), where N <= 2500 is the number of elements in array nums.

Space Complexity: O(N), we can achieve O(1) in space by overwriting values of sub into original nums array.

**Refer to**

<https://leetcode.com/problems/longest-increasing-subsequence/solutions/1326554/longest-increasing-subsequence-optimization-from-brute-force-to-dp-explained>

✔️ ***Solution - V (Binary Search)***

In the brute-force approach, we were not sure if an element should be included or not to form the longest increasing subsequence and thus we explored both options. The problem lies in knowing if an element must be included in the sequence formed till now. Let's instead try an approach where we include element whenever possible to maximize the length and if it's not possible, then create a new subsequence and include it.

Consider an example - [1,7,8,4,5,6,-1,9]:

Let's pick first element - 1 and form the subsequence **sub1=[1]**.

7 is greater than previous element so extend the sequence by picking it.   **sub1=[1,7]**.

Similarly, we pick 8 as well since it's greater than 7.   **sub1=[1,7,8]**

Now we cant extend it further. We can't simply discard previous sequence and start with 4 nor can we discard 7,8 and place 4 instead of them because we don't know if future increasing subsequence will be of more length or not. So we keep both previous subsequence as well as try picking 4 by forming a new subsequence. It's better to form new subsequence and place 4 after 1 to maximize new sequence length. So we have **sub1=[1,7,8]** and **sub2=[1,4]**

Can we add 5 in any of the sequence? Yes we can add it to sub2. If it wasn't possible we would have tried the same approach as in 4th step and created another subsequence list.   **sub1=[1,7,8], sub2=[1,4,5]**

Similarly, add 6 to only possible list - cur2.   **sub1=[1,7,8], sub2=[1,4,5,6]**

Now, -1 cant extend any of the existing subsequence. So we need to form another sequence. Notice we cant copy and use any elements from existing subsequences before -1 either, since -1 is lowest. **sub1=[1,7,8], sub2=[1,4,5,6], sub3=[-1]**

Now, 9 can be used to extend all of the list. At last, we get  **sub1=[1,7,8,9], sub2=[1,4,5,6,9], sub3=[-1,9]**

We finally pick the maximum length of all lists formed till now. This approach works and gets us the correct LIS but it seems like just another **inefficient approach because it's costly to maintain multiple lists and search through all of them when including a new element or making a new list**. Is there a way to speed up this process? Yes. We can just maintain a single list and mark multiple lists inside it. Again, an example will better explain this.

Consider the same example as above - [1,7,8,4,5,6,-1,9]:

Pick first element - 1 and form the subsequence **sub=[1]**.

7 is greater than 1 so extend the existing subsequence by picking it. **sub=[1,7]**.

Similarly, we pick 8 as well since it's greater than 7.  **sub=[1,7,8]**

Now comes the main part. We can't extend any existing sequence with 4. So we need to create a new subsequence following 4th step previous approach but this time we will create it inside sub itself by replacing smallest element larger than 4 (Similar to 4th step above where we formed a new sequence after picking smaller elements than 4 from existing sequence).

    [1,    4,      8]

          ^sub2  ^sub1

This replacement technique works because replaced elements dont matter to us

We only used end elements of existing lists to check if they can be extended otherwise form newer lists

And since we have replaced a bigger element with smaller one it wont affect the

step of creating new list after taking some part of existing list (see step 4 in above approach)

Now, we can't extend with 5 either. We follow the same approach as step 4.

    [1,    4,    5]

                ^sub2

Think of it as extending sub2 in 5th step of above appraoch

Also, we can see sub2 replaced sub1 meaning any subsequence formed with sub2 always has better chance of being LIS than sub1.

We get 6 now and we can extend the sub list by picking it.

    [1,    4,    5,    6]

                      ^sub2

Cant extend with -1. So, Replace -

    [-1,    4,    5,  6]

            ^sub3      ^sub2

We have again formed a new list internally by replacing smallest element larger than -1 from exisiting list

We get 9 which is greater than the end of our list and thus can be used to extend the list

    [-1,    4,    5,    6,    9]

            ^sub3            ^sub2

Finally the length of our maintained list will denote the LIS length = `5`. Do note that it wont give the LIS itself but just correct length of it.

The optimization which improves this approach over DP is applying **Binary search** when we can't extend the sequence and need to replace some element from maintained list - sub. The list always remains sorted and thus binary search gives us the correct index of element in list which will be replaced by current element under iteration.

Basically, we will compare end element of sub with element under iteration cur. If cur is bigger than it, we just extend our list. Otherwise, we will simply apply binary search to find the smallest element >= cur and replace it. Understanding the explanation till now was the hard part...the approach is very easy to code🙂 .

I have used the input array itself as my maintained list. Use an auxiliary array if you're restricted from modifying the input.

class Solution {

public:

    int lengthOfLIS(vector<int>& A) {

        int len = 0;

        for(auto cur : A)

            if(len == 0 || A[len-1] < cur) A[len++] = cur;            // extend

            else \*lower\_bound(begin(A), begin(A) + len, cur) = cur;    // replace

        return len;

    }

};

***Time Complexity :*** **O(NlogN)**

***Space Complexity :*** **O(1)**

**Refer to**

<https://leetcode.com/problems/longest-increasing-subsequence/solutions/1326308/c-python-dp-binary-search-bit-segment-tree-solutions-picture-explain-o-nlogn/>

**✔️ Solution 2: Greedy with Binary Search**

Let's construct the idea from following example.

Consider the example nums = [2, 6, 8, 3, 4, 5, 1], let's try to build the increasing subsequences starting with an empty one: sub1 = [].

Let pick the first element, sub1 = [2].

6 is greater than previous number, sub1 = [2, 6]

8 is greater than previous number, sub1 = [2, 6, 8]

3 is less than previous number, we can't extend the subsequence sub1, but we must keep 3 because in the future there may have the longest subsequence start with [2, 3], sub1 = [2, 6, 8], sub2 = [2, 3].

With 4, we can't extend sub1, but we can extend sub2, so sub1 = [2, 6, 8], sub2 = [2, 3, 4].

With 5, we can't extend sub1, but we can extend sub2, so sub1 = [2, 6, 8], sub2 = [2, 3, 4, 5].

With 1, we can't extend neither sub1 nor sub2, but we need to keep 1, so sub1 = [2, 6, 8], sub2 = [2, 3, 4, 5], sub3 = [1].

Finally, length of longest increase subsequence = len(sub2) = 4.

In the above steps, we need to keep different sub arrays (sub1, sub2..., subk) which causes poor performance. But we notice that we can just keep one sub array, when new number x is not greater than the last element of the subsequence sub, we do binary search to find the smallest element >= x in sub, and replace with number x.

Let's run that example nums = [2, 6, 8, 3, 4, 5, 1] again:

Let pick the first element, sub = [2].

6 is greater than previous number, sub = [2, 6]

8 is greater than previous number, sub = [2, 6, 8]

3 is less than previous number, so we can't extend the subsequence sub. We need to find the smallest number >= 3 in sub, it's

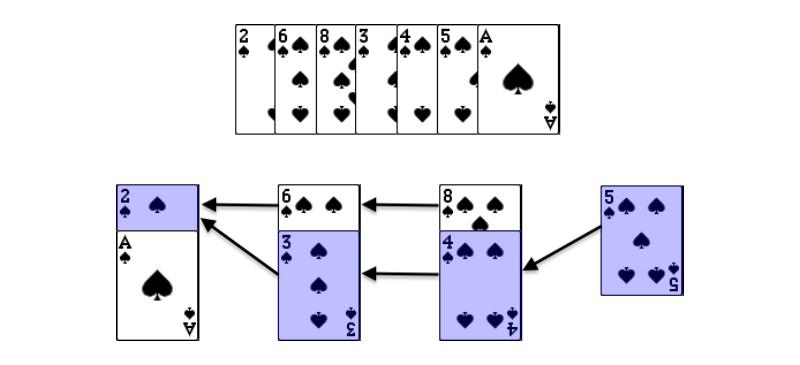
6. Then we overwrite it, now sub = [2, 3, 8].

4 is less than previous number, so we can't extend the subsequence sub. We overwrite 8 by 4, so sub = [2, 3, 4].

5 is greater than previous number, sub = [2, 3, 4, 5].

1 is less than previous number, so we can't extend the subsequence sub. We overwrite 2 by 1, so sub = [1, 3, 4, 5].

Finally, length of longest increase subsequence = len(sub) = 4.



class Solution { // 8 ms, faster than 91.61%

public:

    int lengthOfLIS(vector<int>& nums) {

        vector<int> sub;

        for (int x : nums) {

            if (sub.empty() || sub[sub.size() - 1] < x) {

                sub.push\_back(x);

            } else {

                auto it = lower\_bound(sub.begin(), sub.end(), x); // Find the index of the first element >= x

                \*it = x; // Replace that number with x

            }

        }

        return sub.size();

    }

};

**Complexity**

Time: O(N \* logN), where N <= 2500 is the number of elements in array nums.

Space: O(N), we can achieve O(1) in space by overwriting values of sub into original nums array.

**Why Leetcode 300 and Leetcode 1671 not suitable for Monotonic Stack ?**

LeetCode 300 requires finding the longest increasing subsequence in an array. While a monotonic stack is useful for handling problems with nearest greater or smaller elements efficiently, it is not well-suited for the type of cumulative, global comparisons needed for the longest increasing subsequence. Here's why:

**Global Sequence Tracking vs. Local Neighbors:**

LIS Problem: The LIS problem is inherently about global sequence tracking. Each element in the array must consider all previous elements to determine if it can extend an increasing subsequence. This requires keeping track of the longest valid subsequence up to each point, which isn't something a monotonic stack is designed to handle efficiently.

Monotonic Stack: Typically, it is designed for scenarios where each element only needs to know about its nearest greater or smaller neighbors. It doesn't maintain information about cumulative subsequences or handle longer-range comparisons efficiently.

**Dynamic Programming and Binary Search:**

The conventional approach for solving the LIS problem uses dynamic programming (DP) or a combination of DP and binary search. This allows for efficiently tracking the minimum possible tail value for increasing subsequences of different lengths.

The binary search technique is used to keep the DP array sorted, replacing elements to maintain the smallest possible value for the longest increasing subsequence found so far. This approach ensures a time complexity of O(nlogn), which is optimal for this problem.

**Subsequence Formation:**

In LIS: We care about the cumulative formation of subsequences. For each element, we want to know where it fits in the overall increasing order. A monotonic stack, which focuses on immediate neighbor relationships (like the nearest greater or smaller), doesn’t directly help with forming the longest sequence because it doesn’t track subsequence lengths efficiently.

**Data Structure Requirements:**

The LIS problem is solved by maintaining a sorted list of elements that represent the smallest possible end elements of increasing subsequences of various lengths. A monotonic stack cannot efficiently maintain this sorted list, especially when elements need to be replaced to maintain optimal subsequence formation.

**Example Scenarios Where Monotonic Stack Fits**

Monotonic stacks are perfect for problems where:

You need to find the **next greater** or **next smaller** element for each element in the array (e.g., LeetCode 496, "Next Greater Element I").

Problems like **"Largest Rectangle in Histogram"** (LeetCode 84) that need efficient processing of elements to calculate areas based on nearest smaller elements.

**Summary**

For **LeetCode 300** and **LeetCode 1671**:

Problem Requirements: Both problems require global sequence tracking, i.e., considering the entire sequence to maintain the longest subsequence formation.

Monotonic Stack Limitation: It focuses on immediate, local relationships rather than forming or tracking long-range subsequences.

Optimal Approaches: Dynamic programming (DP) combined with binary search (for optimization) is preferred for both LIS and LDS problems because these methods efficiently handle cumulative and global comparisons needed for subsequence tracking.

**Conclusion**

Both LeetCode 300 and LeetCode 1671 involve problems that require global tracking of subsequences and managing cumulative information. Monotonic stacks excel in scenarios requiring efficient local element comparisons, such as finding nearest greater or smaller elements. However, they are not well-suited for problems needing global comparisons to form or track subsequences, which is why conventional approaches using DP and binary search are used instead.