<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
* -104 <= nums[i] <= 104

**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: 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 -

1. 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.
2. 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 2: DFS + 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 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: DP (60 min)**

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);

// 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 5: 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]:

1. Let's pick first element - 1 and form the subsequence **sub1=[1]**.
2. 7 is greater than previous element so extend the sequence by picking it.   **sub1=[1,7]**.
3. Similarly, we pick 8 as well since it's greater than 7.   **sub1=[1,7,8]**
4. 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]**
5. 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]**
6. Similarly, add 6 to only possible list - cur2.   **sub1=[1,7,8], sub2=[1,4,5,6]**
7. 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]**
8. 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 neighter 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]

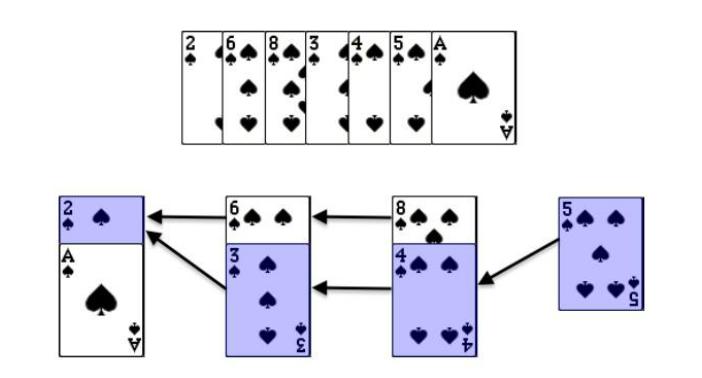
3 is less than previous number, so we can't extend the subsequence sub. We need to find the smallest number >= 3 in sub, it's 6. Then we overwrite it, now sub = [2, 3, 8].

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.