Previous Next

300. Longest Increasing Subsequence

Jan. 17, 2017 | 246.6K views

★★★★☆
Average Rating: 4.17 (187 votes)

Given an unsorted array of integers, find the length of longest increasing subsequence.

Example:

```
Input: [10,9,2,5,3,7,101,18]
Output: 4
Explanation: The longest increasing subsequence is [2,3,7,10
1], therefore the length is 4.
```

Note:

- There may be more than one LIS combination, it is only necessary for you to return the length.
- Your algorithm should run in O(n^2) complexity.

Follow up: Could you improve it to O(n log n) time complexity?

Solution

Approach 1: Brute Force

Algorithm

The simplest approach is to try to find all increasing subsequences and then returning the maximum length of longest increasing subsequence. In order to do this, we make use of a recursive function lengthofLIS which returns the length of the LIS possible from the current element(corresponding to $curpos$) onwards(including the current element). Inside each function call, we consider two cases:

1. The current element is larger than the previous element included in the LIS. In this case, we can include the current element in the LIS. Thus, we find out the length of the LIS obtained by including it. Further, we also find out the length of LIS possible by not including the current element in the LIS. The value returned by the current function call is, thus, the maximum out of the two lengths.

2. The current element is smaller than the previous element included in the LIS. In this case, we can't include the current element in the LIS. Thus, we find out only the length of the LIS possible by not including the current element in the LIS, which is returned by the current function call.

Java

```java
public class Solution {

    public int lengthOfLIS(int[] nums) {
        return lengthofLIS(nums, Integer.MIN_VALUE, 0);
    }

    public int lengthofLIS(int[] nums, int prev, int curpos) {
        if (curpos == nums.length) {
            return 0;
        }
        int taken = 0;
        if (nums[curpos] > prev) {
            taken = 1 + lengthofLIS(nums, nums[curpos], curpos + 1);
        }
        int nottaken = lengthofLIS(nums, prev, curpos + 1);
        return Math.max(taken, nottaken);
    }
}
```

Complexity Analysis

- Time complexity : $O(2^n)$. Size of recursion tree will be 2^n.

- Space complexity : $O(n^2)$. $memo$ array of size $n * n$ is used.

Approach 2: Recursion with Memoization

Algorithm

In the previous approach, many recursive calls had to made again and again with the same parameters. This redundancy can be eliminated by storing the results obtained for a particular call in a 2-d memoization array $memo$. $memo[i][j]$ represents the length of the LIS possible using $nums[i]$ as the previous element considered to be included/not included in the LIS, with $nums[j]$ as the current element considered to be included/not included in the LIS. Here, $nums$ represents the given array.

Java

```java
public class Solution {
    public int lengthOfLIS(int[] nums) {
        int memo[][] = new int[nums.length + 1][nums.length];
        for (int[] l : memo) {
            Arrays.fill(l, -1);
        }
        return lengthofLIS(nums, -1, 0, memo);
    }
    public int lengthofLIS(int[] nums, int previndex, int curpos, int[][] memo) {
        if (curpos == nums.length) {
            return 0;
        }
        if (memo[previndex + 1][curpos] >= 0) {
            return memo[previndex + 1][curpos];
        }
        int taken = 0;
        if (previndex < 0 || nums[curpos] > nums[previndex]) {
            taken = 1 + lengthofLIS(nums, curpos, curpos + 1, memo);
        }

        int nottaken = lengthofLIS(nums, previndex, curpos + 1, memo);
        memo[previndex + 1][curpos] = Math.max(taken, nottaken);
        return memo[previndex + 1][curpos];
    }
}
```

Complexity Analysis

- Time complexity : $O(n^2)$. Size of recursion tree can go upto n^2.

- Space complexity : $O(n^2)$. $memo$ array of $n * n$ is used.

Approach 3: Dynamic Programming

Algorithm

This method relies on the fact that the longest increasing subsequence possible upto the i^{th} index in a given array is independent of the elements coming later on in the array. Thus, if we know the length of the LIS upto i^{th} index, we can figure out the length of the LIS possible by including the $(i+1)^{th}$ element based on the elements with indices j such that $0 \leq j \leq (i+1)$.

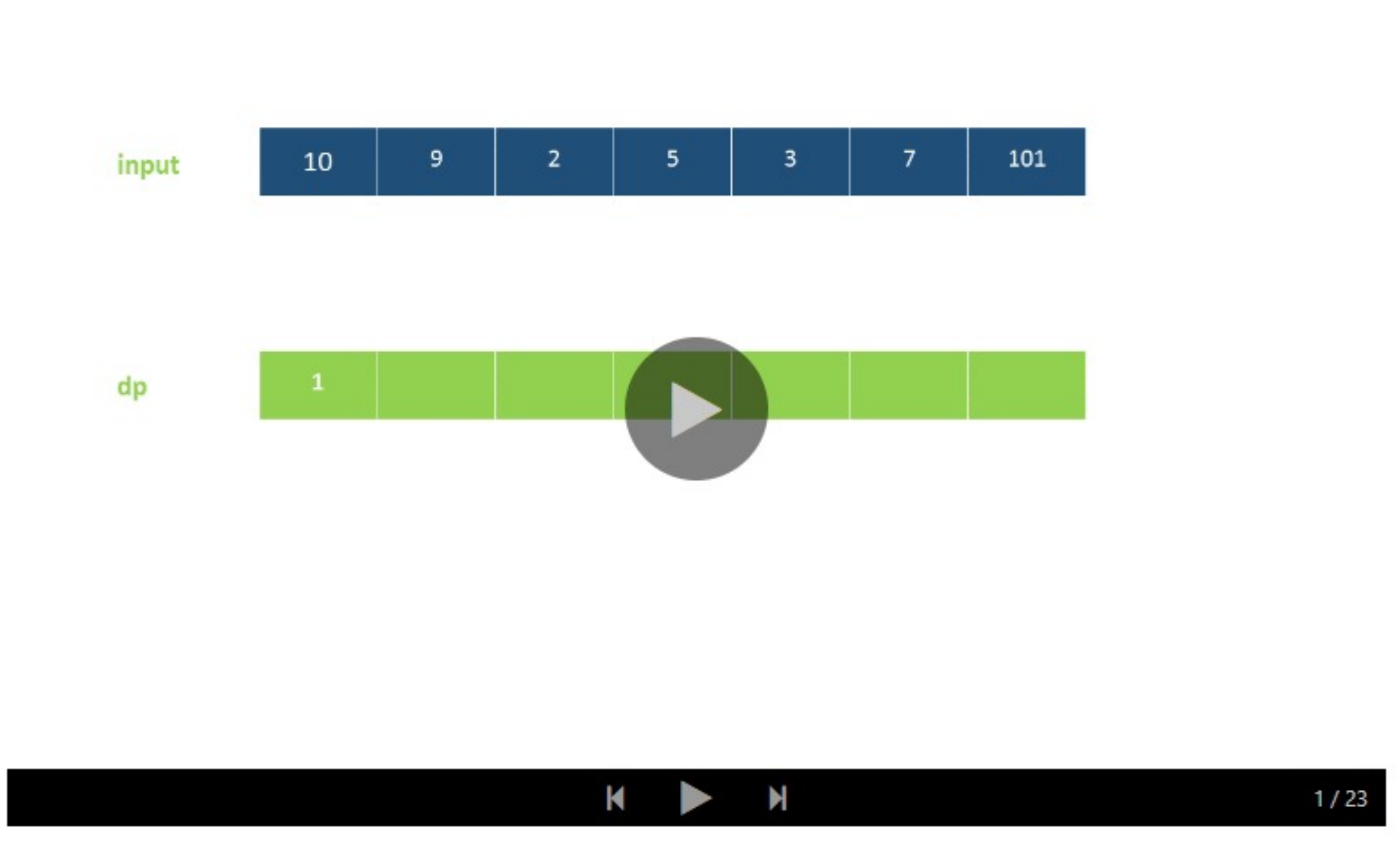
We make use of a dp array to store the required data. $dp[i]$ represents the length of the longest increasing subsequence possible considering the array elements upto the i^{th} index only ,by necessarily including the i^{th} element. In order to find out $dp[i]$, we need to try to append the current element($nums[i]$) in every possible increasing subsequences upto the $(i-1)^{th}$ index(including the $(i-1)^{th}$ index), such that the new sequence formed by adding the current element is also an increasing subsequence. Thus, we can easily determine $dp[i]$ using:

$$dp[i] = \max(dp[j]) + 1, \forall 0 \leq j < i$$

At the end, the maximum out of all the $dp[i]$'s to determine the final result.

$$LIS_{length} = \max(dp[i]), \forall 0 \leq i < n$$

The following animation illustrates the method:



Java

```java
public class Solution {
    public int lengthOfLIS(int[] nums) {
        if (nums.length == 0) {
            return 0;
        }
        int[] dp = new int[nums.length];
        dp[0] = 1;
        int maxans = 1;
        for (int i = 1; i < dp.length; i++) {
            int maxval = 0;
            for (int j = 0; j < i; j++) {
                if (nums[i] > nums[j]) {
                    maxval = Math.max(maxval, dp[j]);
                }
            }
            dp[i] = maxval + 1;
            maxans = Math.max(maxans, dp[i]);
        }
        return maxans;
    }
}
```

Complexity Analysis

- Time complexity : $O(n^2)$. Two loops of n are there.

- Space complexity : $O(n)$. dp array of size n is used.

Approach 4: Dynamic Programming with Binary Search

Algorithm

In this approach, we scan the array from left to right. We also make use of a dp array initialized with all 0's. This dp array is meant to store the increasing subsequence formed by including the currently encountered element. While traversing the $nums$ array, we keep on filling the dp array with the elements encountered so far. For the element corresponding to the j^{th} index ($nums[j]$), we determine its correct position in the dp array(say i^{th} index) by making use of Binary Search(which can be used since the dp array is storing increasing subsequence) and also insert it at the correct position. An important point to be noted is that for Binary Search, we consider only that portion of the dp array in which we have made the updates by inserting some elements at their correct positions(which remains always sorted). Thus, only the elements upto the i^{th} index in the dp array can determine the position of the current element in it. Since, the element enters its correct position(i) in an ascending order in the dp array, the subsequence formed so far in it is surely an increasing subsequence. Whenever this position index i becomes equal to the length of the LIS formed so far(len), it means, we need to update the len as $len = len + 1$.

Note: dp array does not result in longest increasing subsequence, but length of dp array will give you length of LIS.

Consider the example:

input: [0, 8, 4, 12, 2]

dp: [0]

dp: [0, 8]

dp: [0, 4]

dp: [0, 4, 12]

dp: [0 , 2, 12] which is not the longest increasing subsequence, but length of dp array results in length of Longest Increasing Subsequence.

Java

```java
public class Solution {
    public int lengthOfLIS(int[] nums) {
        int[] dp = new int[nums.length];
        int len = 0;
        for (int num : nums) {
            int i = Arrays.binarySearch(dp, 0, len, num);
            if (i < 0) {
                i = -(i + 1);
            }
            dp[i] = num;
            if (i == len) {
                len++;
            }
        }
        return len;
    }
}
```

Note: Arrays.binarySearch() method returns index of the search key, if it is contained in the array, else it returns (-(insertion point) - 1). The insertion point is the point at which the key would be inserted into the array: the index of the first element greater than the key, or a.length if all elements in the array are less than the specified key.

Complexity Analysis

- Time complexity : $O(n \log n)$. Binary search takes $\log n$ time and it is called n times.

- Space complexity : $O(n)$. dp array of size n is used.

Rate this article: ★★★★☆

Previous Next

Comments: 98

Sort By

Type comment here... (Markdown is supported)

Preview Post

nkeng ★311 October 23, 2018 12:24 AM

explanation for last approach is very hard to follow

295 Share Reply

SHOW 11 REPLIES

chrislzm ★1461 April 2, 2018 1:47 PM

Wow, I love the last solution. Who figured this out/what was the intuition?

I kind of have to agree with previous comments about the language used to write this article. A lot Leetcode articles (like this one) attempt to use really formal English, but (I think) because the author is obviously not a native English speaker, it makes the article even harder to understand.

Read More

126 Share Reply

SHOW 4 REPLIES

winner_never_quit ★980 February 25, 2019 7:28 PM

Article mistake: In the 1st approach, there is no memo array is used. So the space complexity is not O(n^2) but O(n), which is the recursion depth.

77 Share Reply

SHOW 1 REPLY

skyfly1010 ★37 December 28, 2018 8:16 AM

Do not understand the last approach!!!!!

36 Share Reply

SHOW 2 REPLIES

sitarama_raju ★29 January 29, 2019 10:44 PM

In approach 3, one equation seems to be incorrect:
dp[i]=max(dp[j])+1,∀0≤j<i — this should also include one more check nums[i] > nums[j]

29 Share Reply

SHOW 3 REPLIES

hongyi192 ★21 August 28, 2018 7:35 AM

wikipedia has a better explanation for the optimal solution
https://en.wikipedia.org/wiki/Longest_increasing_subsequence#Efficient_algorithm

21 Share Reply

SHOW 1 REPLY

denis21 ★15 June 22, 2018 12:15 PM

The last solution is amazing, so it's basically incrementally building the longest incrementing subsequence array.

15 Share Reply

SHOW 2 REPLIES

qqwei ★78 April 21, 2019 12:08 AM

Well I don't see why the fourth approach is regarded as a DP. But it is a really cool idea.

12 Share Reply

jz678 ★110 December 23, 2017 6:07 PM

Your english is very limited and makes the post hard to understand

17 Share Reply

SHOW 1 REPLY

RKG ★50 March 29, 2018 12:43 PM

Nice explanation. I liked the last approach, but using last approach my submission time is 3ms. I replaced Arrays.binarySearch() with my own implementation of binary search and my time improved to just 1ms. Reason: Arrays.binarySearch() library method is optimized for large inputs, so for smaller inputs library uses insertion sort algo rather than binary search.

7 Share Reply

SHOW 1 REPLY

‹ 1 2 3 4 5 6 ... 9 10 ›