Longest Increasing Subsequence

We'll cover the following

Problem Statement

Given a number sequence, find the length of its Longest Increasing Subsequence (LIS). In an increasing subsequence, all the elements are in increasing order (from lowest to highest).

Example 1:

```
Input: {4,2,3,6,10,1,12}
Output: 5
Explanation: The LIS is {2,3,6,10,12}.
```

Example 1:

```
Input: {-4,10,3,7,15}
Output: 4
Explanation: The LIS is {-4,3,7,15}.
```

Basic Solution

A basic brute-force solution could be to try all the subsequences of the given number sequence. We can process one number at a time, so we have two options at any step:

1. If the current number is greater than the previous number that we included, we can increment our count and make a recursive call for the remaining array.
2. We can skip the current number to make a recursive call for the remaining array.

The length of the longest increasing subsequence will be the maximum number returned by the two recurse calls from the above two options.

Code

Here is the code:

Java JS Python3 C++

```python
def find_LIS_length(nums):
  return find_LIS_length_recursive(nums, 0, -1)


def find_LIS_length_recursive(nums, currentIndex,  previousIndex):
  if currentIndex == len(nums):
    return 0

  # include nums[currentIndex] if it is larger than the last included number
  c1 = 0
  if previousIndex == -1 or nums[currentIndex] > nums[previousIndex]:
    c1 = 1 + \
         find_LIS_length_recursive(nums, currentIndex + 1, currentIndex)

  # excluding the number at currentIndex
  c2 = find_LIS_length_recursive(nums, currentIndex + 1, previousIndex)

  return max(c1, c2)


def main():
  print(find_LIS_length([4, 2, 3, 6, 10, 1, 12]))
  print(find_LIS_length([-4, 10, 3, 7, 15]))


main()
```

The time complexity of the above algorithm is exponential $O(2^n)$, where 'n' is the lengths of the input array. The space complexity is $O(n)$ which is used to store the recursion stack.

Top-down Dynamic Programming with Memoization

To overcome the overlapping subproblems, we can use an array to store the already solved subproblems.

The two changing values for our recursive function are the current and the previous index. Therefore, we can store the results of all subproblems in a two-dimensional array. (Another alternative could be to use a hash-table whose key would be a string (currentIndex + "|" + previousIndex)).

Code

Here is the code:

Java JS Python3 C++

```python
def find_LIS_length(nums):
  n = len(nums)
  dp = [[-1 for _ in range(n+1)] for _ in range(n)]
  return find_LIS_length_recursive(dp, nums, 0, -1)
```

```python


def find_LIS_length_recursive(dp, nums, currentIndex, previousIndex):
  if currentIndex == len(nums):
    return 0

  if dp[currentIndex][previousIndex + 1] == -1:
    # include nums[currentIndex] if it is larger than the last included number
    c1 = 0
    if previousIndex == -1 or nums[currentIndex] > nums[previousIndex]:
      c1 = 1 + find_LIS_length_recursive(dp, nums, currentIndex + 1, currentIndex)

    c2 = find_LIS_length_recursive(
      dp, nums, currentIndex + 1, previousIndex)
    dp[currentIndex][previousIndex + 1] = max(c1, c2)

  return dp[currentIndex][previousIndex + 1]


def main():
  print(find_LIS_length([4, 2, 3, 6, 10, 1, 12]))
  print(find_LIS_length([-4, 10, 3, 7, 15]))


main()
```

What is the time and space complexity of the above solution? Since our memoization array `dp[nums.length()][nums.length()]` stores the results for all the subproblems, we can conclude that we will not have more than $N * N$ subproblems (where 'N' is the length of the input sequence). This means that our time complexity will be $O(N^2)$.

The above algorithm will be using $O(N^2)$ space for the memoization array. Other than that we will use $O(N)$ space for the recursion call-stack. So the total space complexity will be $O(N^2 + N)$, which is asymptotically equivalent to $O(N^2)$.

Bottom-up Dynamic Programming

The above algorithm tells us two things:

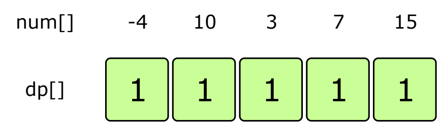
1. If the number at the current index is bigger than the number at the previous index, we increment the count for LIS up to the current index.
2. But if there is a bigger LIS without including the number at the current index, we take that.

So we need to find all the increasing subsequences for the number at index 'i', from all the previous numbers (i.e. number till index 'i-1'), to eventually find the longest increasing subsequence.

If 'i' represents the 'currentIndex' and 'j' represents the 'previousIndex', our recursive formula would look like:

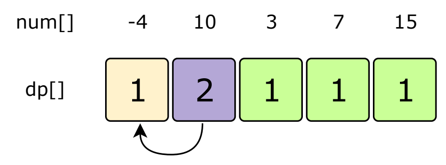
```
    if num[i] > num[j] => dp[i] = dp[j] + 1 if there is no bigger LIS for 'i'
```

Let’s draw this visually for {-4,10,3,7,15}. Start with a subsequence of length ‘1’, as every number will be a LIS of length ‘1’:



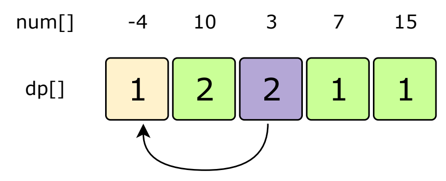
Every number is a LIS of length '1'

1 of 11



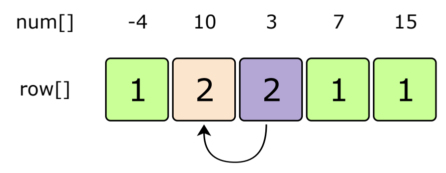
i:1, j:0 => Since num[i] > num[j] and dp[i] <=dp[j], therefore dp[i] = dp[j]+1

2 of 11



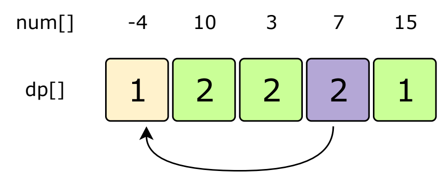
i:2, j:0 => Since num[i] > num[j] and dp[i] <=dp[j], therefore dp[i] = dp[j]+1

3 of 11



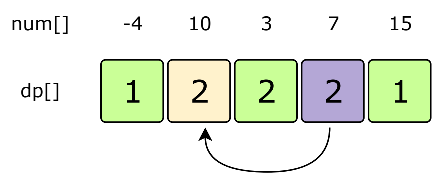
i:2, j:1 => Since num[i] < num[j] so no update

4 of 11



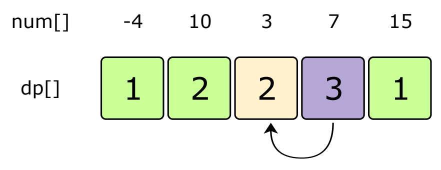
i:3, j:0 => Since num[i] > num[j] and dp[i] <=dp[j], therefore dp[i] = dp[j]+1

5 of 11



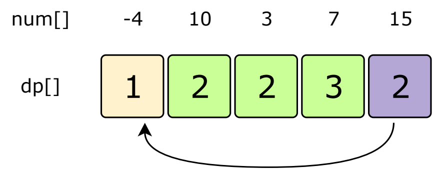
i:3, j:1 => Since num[i] < num[j], so no update

6 of 11



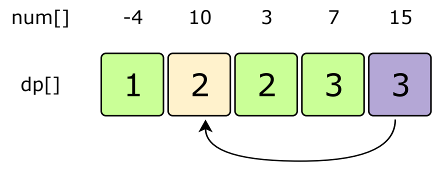
i:3, j:2 => Since num[i] > num[j] and dp[i] <=dp[j], therefore dp[i] = dp[j]+1

7 of 11



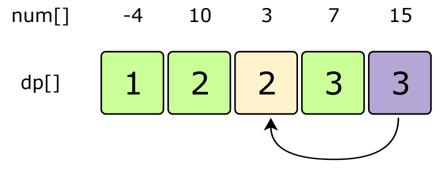
i:4, j:0 => Since num[i] > num[j] and dp[i] <=dp[j], therefore dp[i] = dp[j]+1

8 of 11



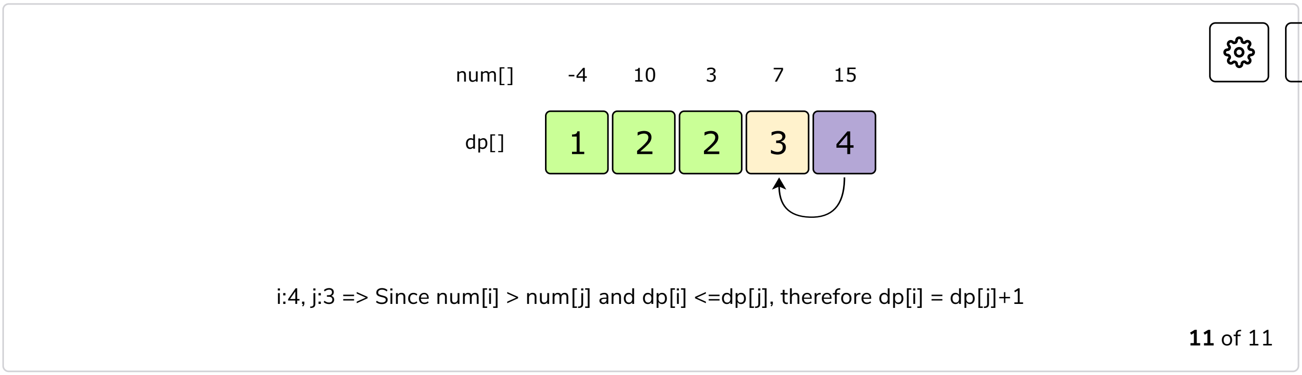
i:4, j:1 => Since num[i] > num[j] and dp[i] <=dp[j], therefore dp[i] = dp[j]+1

9 of 11



i:4, j:2 => Since num[i] > num[j] but dp[i] >dp[j], so no update

10 of 11



i:4, j:3 => Since num[i] > num[j] and dp[i] <=dp[j], therefore dp[i] = dp[j]+1

From the above visualization, we can clearly see that the longest increasing subsequence is of length '4' – as shown by `dp[4]`.

Code

Here is the code for our bottom-up dynamic programming approach:

Java | JS | Python3 | C++

```python
def find_LIS_length(nums):
  n = len(nums)
  dp = [0 for _ in range(n)]
  dp[0] = 1

  maxLength = 1
  for i in range(1, n):
    dp[i] = 1
    for j in range(i):
      if nums[i] > nums[j] and dp[i] <= dp[j]:
        dp[i] = dp[j] + 1
        maxLength = max(maxLength, dp[i])

  return maxLength


def main():
  print(find_LIS_length([4, 2, 3, 6, 10, 1, 12]))
  print(find_LIS_length([-4, 10, 3, 7, 15]))


main()
```

The time complexity of the above algorithm is $O(n^2)$ and the space complexity is $O(n)$.