Dynamic Programming

Content

3. Longest Increasing Subsequence

The Longest Increasing Subsequence (LIS) problem is to find the length of the longest subsequence of a given sequence such that all elements of the subsequence are sorted in increasing order. For example, the length of LIS for {10, 22, 9, 33, 21, 50, 41, 60, 80} is 6 and LIS is {10, 22, 33, 50, 60, 80}.

Input: arr[] = {3, 10, 2, 1, 20}

Output: Length of LIS = 3

The longest increasing subsequence is 3, 10, 20

Input: arr[] = {3, 2}

Output: Length of LIS = 1

The longest increasing subsequences are {3} and {2}

Input: arr[] = {50, 3, 10, 7, 40, 80}

Output: Length of LIS = 4

The longest increasing subsequence is {3, 7, 40, 80}.

Method 1: Recursion

Optimal Substructure - Let arr[0..n-1] be the input array and L(i) be the length of the LIS ending at index i such that arr[i] is the last element of the LIS.

Then, L(i) can be recursively written as:

L(i) = 1 + max(L(j)) where 0 < j < i and arr[j] < arr[i]; or

L(i) = 1, if no such j exists.

To find the LIS for a given array, we need to return max(L(i)) where 0 < i < n.

Formally, the length of the longest increasing subsequence ending at index i, will be 1 greater than the maximum of lengths of all longest increasing subsequences ending at indices before i, where arr[j] < arr[i] (j < i).

Thus, we see the LIS problem satisfies the optimal substructure property as the main problem can be solved using solutions to subproblems.

The recursive tree given below will make the approach clearer:

Input : arr[] = {3, 10, 2, 11}

f(i): Denotes LIS of subarray ending at index 'i'

Time Complexity: The time complexity of this recursive approach is exponential as there is a case of overlapping subproblems as explained in the recursive tree diagram above.

Auxiliary Space: O(1). No external space used for storing values apart from the internal stack space.

Method 2: Dynamic Programming.

We can see that there are many subproblems in the above recursive solution which are solved again and again. So this problem has Overlapping Substructure property and recomputation of same subproblems can be avoided by either using Memoization or Tabulation.

The simulation of approach will make things clear:

Input : arr[] = {3, 10, 2, 11}

LIS[] = {1, 1, 1, 1} (initially)

Iteration-wise simulation :

1. arr[2] > arr[1] {LIS[2] = max(LIS [2], LIS[1]+1)=2}
2. arr[3] < arr[1] {No change}
3. arr[3] < arr[2] {No change}
4. arr[4] > arr[1] {LIS[4] = max(LIS [4], LIS[1]+1)=2}
5. arr[4] > arr[2] {LIS[4] = max(LIS [4], LIS[2]+1)=3}
6. arr[4] > arr[3] {LIS[4] = max(LIS [4], LIS[3]+1)=3}

We can avoid the recomputation of subproblems by using tabulation.

Implementation - https://github.com/lavishabhambri/Weekly-Algo-Newsletter/blob/main/Dynamic%20Programming/Codes/longest_increasing_subsequence.cpp

4. Longest Bitonic Subsequence

In this problem you are given a number n, representing the number of elements of the array. This n is followed by n numbers representing the contents of an array of length n.

All you are required to do is to print the length of the longest bitonic subsequence of the array.

Note: Bitonic subsequence begins with elements in increasing order, followed by elements in decreasing order.
For example: Sample Input: 10 10 22 9 33 21 50 41 60 80 1 Sample Output: 7

As we can see that 10, 22, 33, 50, 60, 80, 1 is the longest Bitonic Subsequence; it increases from 10 to 80 and then decreases to 1. And since its length is 7, the output becomes 7.

In this problem we need to calculate the length of the longest occurring bitonic subsequence in an array.
To solve this problem we will use the same trick, which we used while solving Longest Increasing Subsequence.
First of all, while travelling the array from left to right we will calculate the length of Increasing Subsequence corresponding to each element.
For example, in the sample input array, while travelling from left to right, for element 33, we will store the count as 3; corresponding to 33, which represents length of increasing subsequence (10, 22 and 33).

After completing the traversal from left to right, we will now traverse the array again but this time from right to left and this time we will store the length of decreasing subsequence corresponding to each element.
For example, in the sample input array, while travelling from right to left, for element 33, we will store the count as 3; corresponding to 33, which represents length of decreasing subsequence (33, 21 and 1).
Then we will again travel the array from left to right to calculate the length of bitonic subsequence, corresponding to each element by using the length of increasing subsequence and decreasing subsequence corresponding to that element.
For each element, we can calculate the length of bitonic subsequence as:
Length of bitonic subsequence = length of increasing subsequence + length of decreasing subsequence -1.For example, for element 33, we have 3 as the length of both increasing subsequence (10, 22 and 33) and for decreasing subsequence (33, 21 and 1). So the length of the bitonic subsequence (10, 22, 33, 21, 9) for this point becomes 5 (3 + 3 - 1 = 5).
We basically subtract 1 because that particular element is counted twice, once in the length of increasing subsequence and then in the length of decreasing subsequence.
And while travelling the array last time i.e. for calculating the length of bitonic subsequence for each point, we will maintain a variable omax (Overall Maximum) which will store the length of longest bitonic subsequence so far.

Time Complexity - $O(n^2)$-> A nested for loop is used 2 times and a single for loop once, which condenses to $O(n^2)$.

Space Complexity - O(n) -> Two n sized arrays are used which condenses to O(n).

Implementation -
https://github.com/lavishabhambri/Weekly-Algo-Newsletter/blob/main/Dynamic%20Programming/Codes/longest_bitonic_subarray.cpp