Dynamic Programming II

SWE2016-44

Longest Increasing Subsequence

Longest Increasing Subsequence

Find the length of the longest subsequence of a given sequence such that all elements of the subsequence are sorted in increasing order.

Longest Increasing Subsequence

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,

Given sequence LIS = {10, 22, 9, 33, 21, 50, 41, 60}

Longest Increasing Subsequence

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,

Given sequence LIS = {10, 22, 9, 33, 21, 50, 41, 60}

Subsequences: {10}, {10, 22}, {10, 9, 33}, {9, 21, 60}, {50, 60}, …

Longest Increasing Subsequence

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,

Given sequence LIS = {10, 22, 9, 33, 21, 50, 41, 60}

Subsequences: {10}, {10, 22}, {10, 9, 33}, {9, 21, 60}, {50, 60}, …

Increasing Subsequences: {10}, {9, 33, 41}, {33, 41, 60}, {41}, …

Longest Increasing Subsequence

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,

Given sequence LIS = {10, 22, 9, 33, 21, 50, 41, 60}

Subsequences: {10}, {10, 22}, {10, 9, 33}, {9, 21, 60}, {50, 60}, …

Increasing Subsequences: {10}, {9, 33, 41}, {33, 41, 60}, {41}, …

Longest Increasing Subsequences: {10, 22, 33, 50, 60} or {10, 22, 33, 41, 60}

Longest Increasing Subsequence

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,

Given sequence LIS = {10, 22, 9, 33, 21, 50, 41, 60}

Subsequences: {10}, {10, 22}, {10, 9, 33}, {9, 21, 60}, {50, 60}, …

Increasing Subsequences: {10}, {9, 33, 41}, {33, 41, 60}, {41}, …

Longest Increasing Subsequences: {10, 22, 33, 50, 60} or {10, 22, 33, 41, 60}

So, Length of LIS = 5

Optimal Substructure Property

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.

Optimal Substructure Property

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.

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.

Optimal Substructure Property

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.

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, return max(L(i)) where 0<i<n.

Optimal Substructure Property

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.

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, return max(L(i)) where $0<i<n$.

→The LIS problem satisfies the optimal substructure property.

Overlapping Substructure Property

```
                lis(4)
          /        |
       lis(3)    lis(2)   lis(1)
      /            /
   lis(2) lis(1) lis(1)
   /
lis(1)
```

Longest Increasing Subsequence

Initialize LIS value

iterator								
arr[]	10	22	9	33	21	50	41	60
LIS	1	1	1	1	1	1	1	1

Longest Increasing Subsequence

For i=1:

iterator	j	i						
arr[]	10	22	9	33	21	50	41	60
LIS	1	1	1	1	1	1	1	1

Longest Increasing Subsequence

For i=1:

iterator	j	i						
arr[]	10	22	9	33	21	50	41	60
LIS	1	2	1	1	1	1	1	1

Longest Increasing Subsequence

For i=2:

iterator	j		i					
arr[]	10	22	9	33	21	50	41	60
LIS	1	2	1	1	1	1	1	1

Longest Increasing Subsequence

For i=2:

iterator		j	i					
arr[]	10	22	9	33	21	50	41	60
LIS	1	2	1	1	1	1	1	1

Longest Increasing Subsequence

For i=3:

iterator	j			i				
arr[]	10	22	9	33	21	50	41	60
LIS	1	2	1	1	1	1	1	1

Longest Increasing Subsequence

For i=3:

iterator	j			i				
arr[]	10	22	9	33	21	50	41	60
LIS	1	2	1	2	1	1	1	1

Longest Increasing Subsequence

For i=3:

iterator		j		i				
arr[]	10	22	9	33	21	50	41	60
LIS	1	2	1	2	1	1	1	1

Longest Increasing Subsequence

For i=3:

iterator		j		i				
arr[]	10	22	9	33	21	50	41	60
LIS	1	2	1	3	1	1	1	1

Longest Increasing Subsequence

For i=3:

iterator			**j**	**i**				
arr[]	**10**	**22**	**9**	**33**	**21**	**50**	**41**	**60**
LIS	**1**	**2**	**1**	**3**	**1**	**1**	**1**	**1**

Longest Increasing Subsequence

For i=4:

iterator	j				i			
arr[]	10	22	9	33	21	50	41	60
LIS	1	2	1	3	1	1	1	1

Longest Increasing Subsequence

For i=4:

iterator	j				i			
arr[]	10	22	9	33	21	50	41	60
LIS	1	2	1	3	2	1	1	1

Longest Increasing Subsequence

For i=4:

iterator		j			i			
arr[]	10	22	9	33	21	50	41	60
LIS	1	2	1	3	2	1	1	1

Longest Increasing Subsequence

For i=4:

iterator			j		i			
arr[]	10	22	9	33	21	50	41	60
LIS	1	2	1	3	2	1	1	1

Longest Increasing Subsequence

For i=4:

iterator				j	i			
arr[]	10	22	9	33	21	50	41	60
LIS	1	2	1	3	2	1	1	1

Longest Increasing Subsequence

For i=5:

iterator	j					i		
arr[]	10	22	9	33	21	50	41	60
LIS	1	2	1	3	2	1	1	1

Longest Increasing Subsequence

For i=5:

iterator	j					i		
arr[]	10	22	9	33	21	50	41	60
LIS	1	2	1	3	2	2	1	1

Longest Increasing Subsequence

For i=5:

iterator		j				i		
arr[]	10	22	9	33	21	50	41	60
LIS	1	2	1	3	2	2	1	1

Longest Increasing Subsequence

For i=5:

iterator		j				i		
arr[]	10	22	9	33	21	50	41	60
LIS	1	2	1	3	2	3	1	1

Longest Increasing Subsequence

For i=5:

iterator			j			i		
arr[]	10	22	9	33	21	50	41	60
LIS	1	2	1	3	2	3	1	1

Longest Increasing Subsequence

For i=5:

iterator				j		i		
arr[]	10	22	9	33	21	50	41	60
LIS	1	2	1	3	2	3	1	1

Longest Increasing Subsequence

For i=5:

iterator				j		i		
arr[]	10	22	9	33	21	50	41	60
LIS	1	2	1	3	2	4	1	1

Longest Increasing Subsequence

For i=5:

iterator					j	i		
arr[]	10	22	9	33	21	50	41	60
LIS	1	2	1	3	2	4	1	1

Longest Increasing Subsequence

For i=6:

iterator	j						i	
arr[]	10	22	9	33	21	50	41	60
LIS	1	2	1	3	2	4	1	1

Longest Increasing Subsequence

For i=6:

iterator	j						i	
arr[]	10	22	9	33	21	50	41	60
LIS	1	2	1	3	2	4	2	1

Longest Increasing Subsequence

For i=6:

iterator		j					i	
arr[]	10	22	9	33	21	50	41	60
LIS	1	2	1	3	2	4	2	1

Longest Increasing Subsequence

For i=6:

iterator		**j**					**i**	
arr[]	**10**	**22**	**9**	**33**	**21**	**50**	**41**	**60**
LIS	**1**	**2**	**1**	**3**	**2**	**4**	**3**	**1**

Longest Increasing Subsequence

For i=6:

iterator			j				i	
arr[]	10	22	9	33	21	50	41	60
LIS	1	2	1	3	2	4	3	1

Longest Increasing Subsequence

For i=6:

iterator				j			i	
arr[]	10	22	9	33	21	50	41	60
LIS	1	2	1	3	2	4	3	1

Longest Increasing Subsequence

For i=6:

iterator				j			i	
arr[]	10	22	9	33	21	50	41	60
LIS	1	2	1	3	2	4	4	1

Longest Increasing Subsequence

For i=6:

iterator					j		i	
arr[]	10	22	9	33	21	50	41	60
LIS	1	2	1	3	2	4	4	1

Longest Increasing Subsequence

For i=6:

iterator						j	i	
arr[]	10	22	9	33	21	50	41	60
LIS	1	2	1	3	2	4	4	1

Longest Increasing Subsequence

Final values:

iterator							j	i
arr[]	10	22	9	33	21	50	41	60
LIS	1	2	1	3	2	4	4	5

Recursive Implementation

```c
int _lis( int arr[], int n, int *max_ref)
{
    /* Base case */
    if (n == 1)
        return 1;

    // 'max_ending_here' is length of LIS ending with arr[n-1]
    int res, max_ending_here = 1;

    /* Recursively get all LIS ending with arr[0], arr[1] ...
       arr[n-2]. If   arr[i-1] is smaller than arr[n-1], and
       max ending with arr[n-1] needs to be updated, then
       update it */
    for (int i = 1; i < n; i++)
    {
        res = _lis(arr, i, max_ref);
        if (arr[i-1] < arr[n-1] && res + 1 > max_ending_here)
            max_ending_here = res + 1;
    }

    // Compare max_ending_here with the overall max. And
    // update the overall max if needed
    if (*max_ref < max_ending_here)
       *max_ref = max_ending_here;

    // Return length of LIS ending with arr[n-1]
    return max_ending_here;
}
```

```c
int lis(int arr[], int n)
{
    // The max variable holds the result
    int max = 1;

    // The function _lis() stores its result in max
    _lis( arr, n, &max );

    // returns max
    return max;
}
```

Recursive Implementation

```
int _lis( int arr[], int n, int *max_ref)
{
    /* Base case */
    if (n == 1)
        return 1;

    // 'max_ending_here' is length of LIS ending with arr[n-1]
    int res, max_ending_here = 1;

    /* Recursively get all LIS ending with arr[0], arr[1] ...
       arr[n-2]. If   arr[i-1] is smaller than arr[n-1], and
       max ending with arr[n-1] needs to be updated, then
       update it */
    for (int i = 1; i < n; i++)
    {
        res = _lis(arr, i, max_ref);
        if (arr[i-1] < arr[n-1] && res + 1 > max_ending_here)
            max_ending_here = res + 1;
    }

    // Compare max_ending_here with the overall max. And
    // update the overall max if needed
    if (*max_ref < max_ending_here)
       *max_ref = max_ending_here;

    // Return length of LIS ending with arr[n-1]
    return max_ending_here;
}
```

```
int lis(int arr[], int n)
{
    // The max variable holds the result
    int max = 1;

    // The function _lis() stores its result in max
    _lis( arr, n, &max );

    // returns max
    return max;
}
```

→ **Time Complextity: $O(2^n)$**

Dynamic Programming

```cpp
#include<bits/stdc++.h>
using namespace std;

/* lis() returns the length of the longest increasing
  subsequence in arr[] of size n */
int lis( int arr[], int n )
{
    int lis[n];

    lis[0] = 1;

    /* Compute optimized LIS values in bottom up manner */
    for (int i = 1; i < n; i++ )
    {
        lis[i] = 1;
        for (int j = 0; j < i; j++ )
            if ( arr[i] > arr[j] && lis[i] < lis[j] + 1)
                lis[i] = lis[j] + 1;
    }

    // Return maximum value in lis[]
    return *max_element(lis, lis+n);
}
```

Tabulation

Dynamic Programming

```cpp
#include<bits/stdc++.h>
using namespace std;

/* lis() returns the length of the longest increasing
  subsequence in arr[] of size n */
int lis( int arr[], int n )
{
    int lis[n];

    lis[0] = 1;

    /* Compute optimized LIS values in bottom up manner */
    for (int i = 1; i < n; i++ )
    {
        lis[i] = 1;
        for (int j = 0; j < i; j++ )
            if ( arr[i] > arr[j] && lis[i] < lis[j] + 1)
                lis[i] = lis[j] + 1;
    }

    // Return maximum value in lis[]
    return *max_element(lis, lis+n);
}
```

Tabulation

→ Time Complextity: $O(n^2)$

Longest Common Subsequence

Longest Common Subsequence

Given two sequences, find the length of longest subsequence present in both of them.

Longest Common Subsequence

Given two sequences, find the length of longest subsequence present in both of them.

A subsequence is a sequence that appears in the same relative order, but not necessarily contiguous.

Longest Common Subsequence

Given two sequences, find the length of longest subsequence present in both of them.

A subsequence is a sequence that appears in the same relative order, but not necessarily contiguous.

Sequences = “abcdefg”, “abxdfg”

Longest Common Subsequence

Given two sequences, find the length of longest subsequence present in both of them.

A subsequence is a sequence that appears in the same relative order, but not necessarily contiguous.

Sequences = “abcdefg”, “abxdfg”

Common Subsequences = “a”, “b”, “d”, “f”, “g”, “ab”, “df”, “dfg”, “abd”, “abdfg”

Longest Common Subsequence

Given two sequences, find the length of longest subsequence present in both of them.

A subsequence is a sequence that appears in the same relative order, but not necessarily contiguous.

Sequences = “abcdefg”, “abxdfg”

Common Subsequences = “a”, “b”, “d”, “f”, “g”, “ab”, “df”, “dfg”, “abd”, “abdfg”

Longest Common Subsequences (LCS) = “abdfg”

Optimal Substructure Property

Let the input sequences be X[0..m-1] and Y[0..n-1]. And let L(X[0..m-1], Y[0..n-1]) be the length of LCS of the two sequences X and Y.

Optimal Substructure Property

Let the input sequences be X[0..m-1] and Y[0..n-1]. And let L(X[0..m-1], Y[0..n-1]) be the length of LCS of the two sequences X and Y.

If last characters of both sequences match (or X[m-1]==Y[n-1]) then L(X[0..m-1], Y[0..n-1]) = 1 + L(X[0..m-2], Y[0..n-2])

Optimal Substructure Property

Let the input sequences be X[0..m-1] and Y[0..n-1]. And let L(X[0..m-1], Y[0..n-1]) be the length of LCS of the two sequences X and Y.

If last characters of both sequences match (or X[m-1]==Y[n-1]) then L(X[0..m-1], Y[0..n-1]) = 1 + L(X[0..m-2], Y[0..n-2])

If last characters of both sequences don't match (or X[m-1]!=Y[n-1]) then L(X[0..m-1], Y[0..n-1]) = MAX (L(X[0..m-2], Y[0..n-1]), L(X[0..m-1], Y[0..n-2]))

Optimal Substructure Property

Let the input sequences be X[0..m-1] and Y[0..n-1]. And let L(X[0..m-1], Y[0..n-1]) be the length of LCS of the two sequences X and Y.

If last characters of both sequences match (or X[m-1]==Y[n-1]) then L(X[0..m-1], Y[0..n-1]) = 1 + L(X[0..m-2], Y[0..n-2])

If last characters of both sequences don't match (or X[m-1]!=Y[n-1]) then L(X[0..m-1], Y[0..n-1]) = MAX (L(X[0..m-2], Y[0..n-1]), L(X[0..m-1], Y[0..n-2]))

→The LIS problem satisfies the optimal substructure property.

Overlapping Subproblems Property

```
                             lcs("AXYT", "AYZX")
                           /
          lcs("AXY", "AYZX")              lcs("AXYT", "AYZ")
          /                                /
lcs("AX", "AYZX") lcs("AXY", "AYZ")   lcs("AXY", "AYZ") lcs("AXYT", "AY")
```

Longest Common Subsequence

Example: Consider the input strings L_1 with length m and L_2 with length n such that L_1="AGGTAB" and L_2="GXTXAYB"

Longest Common Subsequence

Example: Consider the input strings L_1 with length m and L_2 with length n such that L_1=“AGGTAB” and L_2=“GXTXAYB”

1. **There can be two cases. The last characters match or the last characters do not match.**

Longest Common Subsequence

Example: Consider the input strings L_1 with length m and L_2 with length n such that L_1=“AGGTAB” and L_2=“GXTXAYB”

1. **There can be two cases. The last characters match or the last characters do not match.**
2. **If the last characters match: Increment the length of LCS by 1 and process L_1[m-1] and L_2[n-1].**

Longest Common Subsequence

Example: Consider the input strings L_1 with length m and L_2 with length n such that L_1="AGGTAB" and L_2="GXTXAYB"

1. **There can be two cases. The last characters match or the last characters do not match.**
2. **If the last characters match: Increment the length of LCS by 1 and process $L_1[m-1]$ and $L_2[n-1]$.**
3. **If the last characters do not match: Find max($L_1[m-1]$ $L_2[n]$, $L_1[m]$ $L_2[n-1]$).**

Dynamic Programming

Approach:

- **If the last characters match:**

  LCS[i][j] = LCS[i-1][j-1] + 1

- **If the last characters do not match:**

  LCS[i][j] = max(LCS[i-1][j], LCS[i][j-1])

Dynamic Programming

LCS	ø	A	G	G	T	A	B
ø	0	0	0	0	0	0	0
G	0						
X	0						
T	0						
X	0						
A	0						
Y	0						
B	0						

Dynamic Programming

LCS	ø	A	G	G	T	A	B
ø	0	0	0	0	0	0	0
G	0	0					
X	0						
T	0						
X	0						
A	0						
Y	0						
B	0						

Dynamic Programming

LCS	ø	A	G	G	T	A	B
ø	0	0	0	0	0	0	0
G	0	0	1				
X	0						
T	0						
X	0						
A	0						
Y	0						
B	0						

Dynamic Programming

LCS	ø	A	G	G	T	A	B
ø	0	0	0	0	0	0	0
G	0	0	1	1			
X	0						
T	0						
X	0						
A	0						
Y	0						
B	0						

Dynamic Programming

LCS	ø	A	G	G	T	A	B
ø	0	0	0	0	0	0	0
G	0	0	1	1	1		
X	0						
T	0						
X	0						
A	0						
Y	0						
B	0						

Dynamic Programming

LCS	ø	A	G	G	T	A	B
ø	0	0	0	0	0	0	0
G	0	0	1	1	1	1	
X	0						
T	0						
X	0						
A	0						
Y	0						
B	0						

Dynamic Programming

LCS	ø	A	G	G	T	A	B
ø	0	0	0	0	0	0	0
G	0	0	1	1	1	1	1
X	0						
T	0						
X	0						
A	0						
Y	0						
B	0						

Dynamic Programming

LCS	ø	A	G	G	T	A	B
ø	0	0	0	0	0	0	0
G	0	0	1	1	1	1	1
X	0	0					
T	0						
X	0						
A	0						
Y	0						
B	0						

Dynamic Programming

LCS	ø	A	G	G	T	A	B
ø	0	0	0	0	0	0	0
G	0	0	1	1	1	1	1
X	0	0	1				
T	0						
X	0						
A	0						
Y	0						
B	0						

Dynamic Programming

LCS	ø	A	G	G	T	A	B
ø	0	0	0	0	0	0	0
G	0	0	1	1	1	1	1
X	0	0	1	1	1	1	1
T	0						
X	0						
A	0						
Y	0						
B	0						

Dynamic Programming

LCS	ø	A	G	G	T	A	B
ø	0	0	0	0	0	0	0
G	0	0	1	1	1	1	1
X	0	0	1	1	1	1	1
T	0	0					
X	0						
A	0						
Y	0						
B	0						

Dynamic Programming

LCS	ø	A	G	G	T	A	B
ø	0	0	0	0	0	0	0
G	0	0	1	1	1	1	1
X	0	0	1	1	1	1	1
T	0	0	1				
X	0						
A	0						
Y	0						
B	0						

Dynamic Programming

LCS	ø	A	G	G	T	A	B
ø	0	0	0	0	0	0	0
G	0	0	1	1	1	1	1
X	0	0	1	1	1	1	1
T	0	0	1	1			
X	0						
A	0						
Y	0						
B	0						

Dynamic Programming

LCS	ø	A	G	G	T	A	B
ø	0	0	0	0	0	0	0
G	0	0	1	1	1	1	1
X	0	0	1	1	1	1	1
T	0	0	1	1	2		
X	0						
A	0						
Y	0						
B	0						

Dynamic Programming

LCS	ø	A	G	G	T	A	B
ø	0	0	0	0	0	0	0
G	0	0	1	1	1	1	1
X	0	0	1	1	1	1	1
T	0	0	1	1	2	2	2
X	0						
A	0						
Y	0						
B	0						

Dynamic Programming

LCS	ø	A	G	G	T	A	B
ø	0	0	0	0	0	0	0
G	0	0	1	1	1	1	1
X	0	0	1	1	1	1	1
T	0	0	1	1	2	2	2
X	0	0	1	1	2	2	2
A	0						
Y	0						
B	0						

Dynamic Programming

LCS	ø	A	G	G	T	A	B
ø	0	0	0	0	0	0	0
G	0	0	1	1	1	1	1
X	0	0	1	1	1	1	1
T	0	0	1	1	2	2	2
X	0	0	1	1	2	2	2
A	0	1	1	1	2	3	3
Y	0						
B	0						

Dynamic Programming

LCS	ø	A	G	G	T	A	B
ø	0	0	0	0	0	0	0
G	0	0	1	1	1	1	1
X	0	0	1	1	1	1	1
T	0	0	1	1	2	2	2
X	0	0	1	1	2	2	2
A	0	1	1	1	2	3	3
Y	0	1	1	1	2	3	3
B	0						

Dynamic Programming

LCS	ø	A	G	G	T	A	B
ø	0	0	0	0	0	0	0
G	0	0	1	1	1	1	1
X	0	0	1	1	1	1	1
T	0	0	1	1	2	2	2
X	0	0	1	1	2	2	2
A	0	1	1	1	2	3	3
Y	0	1	1	1	2	3	3
B	0	1	1	1	2	3	4

Dynamic Programming

LCS	ø	A	G	G	T	A	B
ø	0	0	0	0	0	0	0
G	0	0	1	1	1	1	1
X	0	0	1	1	1	1	1
T	0	0	1	1	2	2	2
X	0	0	1	1	2	2	2
A	0	1	1	1	2	3	3
Y	0	1	1	1	2	3	3
B	0	1	1	1	2	3	4

Dynamic Programming

LCS	ø	A	G	G	T	A	B
ø	0	0	0	0	0	0	0
G	0	0	1	1	1	1	1
X	0	0	1	1	1	1	1
T	0	0	1	1	2	2	2
X	0	0	1	1	2	2	2
A	0	1	1	1	2	3	3
Y	0	1	1	1	2	3	3
B	0	1	1	1	2	3	4

Dynamic Programming

LCS	ø	A	G	G	T	A	B
ø	0	0	0	0	0	0	0
G	0	0	1	1	1	1	1
X	0	0	1	1	1	1	1
T	0	0	1	1	2	2	2
X	0	0	1	1	2	2	2
A	0	1	1	1	2	3	3
Y	0	1	1	1	2	3	3
B	0	1	1	1	2	3	4

Dynamic Programming

LCS	ø	A	G	G	T	A	B
ø	0	0	0	0	0	0	0
G	0	0	1	1	1	1	1
X	0	0	1	1	1	1	1
T	0	0	1	1	2	2	2
X	0	0	1	1	2	2	2
A	0	1	1	1	2	3	3
Y	0	1	1	1	2	3	3
B	0	1	1	1	2	3	4

Dynamic Programming

LCS	ø	A	G	G	T	A	B
ø	0	0	0	0	0	0	0
G	0	0	1	1	1	1	1
X	0	0	1	1	1	1	1
T	0	0	1	1	2	2	2
X	0	0	1	1	2	2	2
A	0	1	1	1	2	3	3
Y	0	1	1	1	2	3	3
B	0	1	1	1	2	3	4

Dynamic Programming

LCS	ø	A	G	G	T	A	B
ø	0	0	0	0	0	0	0
G	0	0	1	1	1	1	1
X	0	0	1	1	1	1	1
T	0	0	1	1	2	2	2
X	0	0	1	1	2	2	2
A	0	1	1	1	2	3	3
Y	0	1	1	1	2	3	3
B	0	1	1	1	2	3	4

Dynamic Programming

LCS	ø	A	G	G	T	A	B
ø	0	0	0	0	0	0	0
G	0	0	1	1	1	1	1
X	0	0	1	1	1	1	1
T	0	0	1	1	2	2	2
X	0	0	1	1	2	2	2
A	0	1	1	1	2	3	3
Y	0	1	1	1	2	3	3
B	0	1	1	1	2	3	4

Dynamic Programming

LCS	ø	A	G	G	T	A	B
ø	0	0	0	0	0	0	0
G	0	0	1	1	1	1	1
X	0	0	1	1	1	1	1
T	0	0	1	1	2	2	2
X	0	0	1	1	2	2	2
A	0	1	1	1	2	3	3
Y	0	1	1	1	2	3	3
B	0	1	1	1	2	3	4

Dynamic Programming

LCS	ø	A	G	G	T	A	B
ø	0	0	0	0	0	0	0
G	0	0	1	1	1	1	1
X	0	0	1	1	1	1	1
T	0	0	1	1	2	2	2
X	0	0	1	1	2	2	2
A	0	1	1	1	2	3	3
Y	0	1	1	1	2	3	3
B	0	1	1	1	2	3	4

Dynamic Programming

LCS	ø	A	G	G	T	A	B
ø	0	0	0	0	0	0	0
G	0	0	1	1	1	1	1
X	0	0	1	1	1	1	1
T	0	0	1	1	2	2	2
X	0	0	1	1	2	2	2
A	0	1	1	1	2	3	3
Y	0	1	1	1	2	3	3
B	0	1	1	1	2	3	4

GTAB → Length of LCS = 4

Recursive Implementation

```c
/* Returns length of LCS for X[0..m-1], Y[0..n-1] */
int lcs( char *X, char *Y, int m, int n )
{
    if (m == 0 || n == 0)
        return 0;
    if (X[m-1] == Y[n-1])
        return 1 + lcs(X, Y, m-1, n-1);
    else
        return max(lcs(X, Y, m, n-1), lcs(X, Y, m-1, n));
}

/* Utility function to get max of 2 integers */
int max(int a, int b)
{
    return (a > b)? a : b;
}
```

Recursive Implementation

```c
/* Returns length of LCS for X[0..m-1], Y[0..n-1] */
int lcs( char *X, char *Y, int m, int n )
{
    if (m == 0 || n == 0)
        return 0;
    if (X[m-1] == Y[n-1])
        return 1 + lcs(X, Y, m-1, n-1);
    else
        return max(lcs(X, Y, m, n-1), lcs(X, Y, m-1, n));
}

/* Utility function to get max of 2 integers */
int max(int a, int b)
{
    return (a > b)? a : b;
}
```

→ **Time Complextity: $O(2^n)$**

Dynamic Programming

```c
/* Returns length of LCS for X[0..m-1], Y[0..n-1] */
int lcs( char *X, char *Y, int m, int n )
{
    int L[m + 1][n + 1];
    int i, j;

    /* Following steps build L[m+1][n+1] in
       bottom up fashion. Note that L[i][j]
       contains length of LCS of X[0..i-1]
       and Y[0..j-1] */
    for (i = 0; i <= m; i++)
    {
        for (j = 0; j <= n; j++)
        {
        if (i == 0 || j == 0)
            L[i][j] = 0;

        else if (X[i - 1] == Y[j - 1])
            L[i][j] = L[i - 1][j - 1] + 1;

        else
            L[i][j] = max(L[i - 1][j], L[i][j - 1]);
        }
    }

    /* L[m][n] contains length of LCS
    for X[0..n-1] and Y[0..m-1] */
    return L[m][n];
}
```

Tabulation

Dynamic Programming

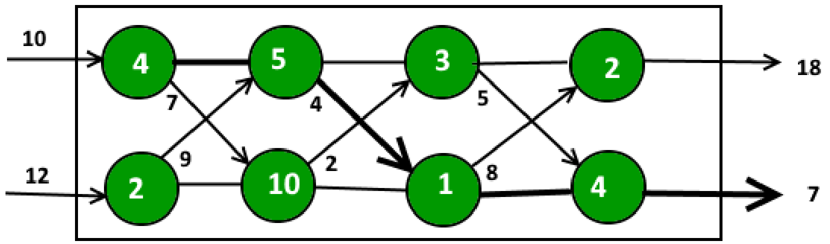
```c
/* Returns length of LCS for X[0..m-1], Y[0..n-1] */
int lcs( char *X, char *Y, int m, int n )
{
    int L[m + 1][n + 1];
    int i, j;

    /* Following steps build L[m+1][n+1] in
       bottom up fashion. Note that L[i][j]
       contains length of LCS of X[0..i-1]
       and Y[0..j-1] */
    for (i = 0; i <= m; i++)
    {
        for (j = 0; j <= n; j++)
        {
        if (i == 0 || j == 0)
            L[i][j] = 0;

        else if (X[i - 1] == Y[j - 1])
            L[i][j] = L[i - 1][j - 1] + 1;

        else
            L[i][j] = max(L[i - 1][j], L[i][j - 1]);
        }
    }

    /* L[m][n] contains length of LCS
    for X[0..n-1] and Y[0..m-1] */
    return L[m][n];
}
```

Tabulation

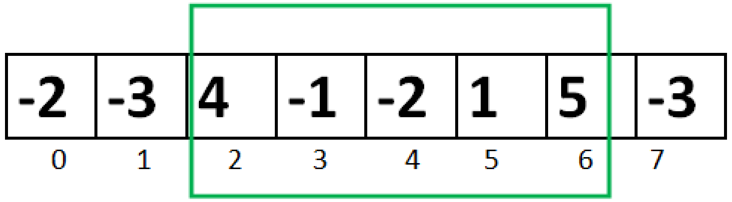
→ Time Complextity: O(mn)

Other Dynamic Programming Questions

- **Assembly Line Scheduling**



- **Largest Sum Contiguous Subarray**



4 + (-1) + (-2) + 1 + 5 = 7

Maximum Contiguous Array Sum is 7

Other Dynamic Programming Questions

- **0-1 Knapsack Problem**

0-1 Knapsack Problem

value[] = {60, 100, 120};
weight[] = {10, 20, 30};
W = 50;

Solution: 220

Weight = 10; Value = 60;
Weight = 20; Value = 100;
Weight = 30; Value = 120;
Weight = (20+10); Value = (100+60);
Weight = (30+10); Value = (120+60);
Weight = (30+20); Value = (120+100);
Weight = (30+20+10) > 50

- **Building Bridges**

```
8     1     4     3     5     2     6     7
<---- Cities on the other bank of river---->
--------------------------------------------
  <--------------- River--------------->
--------------------------------------------
1     2     3     4     5     6     7     8
<------- Cities on one bank of river------->
```

Reference

- Charles Leiserson and Piotr Indyk, “*Introduction to Algorithms*”, September 29, 2004

- https://www.geeksforgeeks.org