

Dynamic Programming | Set 4 (Longest Common Subsequence)

We have discussed Overlapping Subproblems and Optimal Substructure properties in [Set 1](#) and [Set 2](#) respectively. We also discussed one example problem in [Set 3](#). Let us discuss Longest Common Subsequence (LCS) problem as one more example problem that can be solved using Dynamic Programming.

LCS Problem Statement: 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. For example, “abc”, “abg”, “bdf”, “aeg”, “acefg”, .. etc are subsequences of “abcdefg”. So a string of length n has 2^n different possible subsequences.

It is a classic computer science problem, the basis of [diff](#) (a file comparison program that outputs the differences between two files), and has applications in bioinformatics.

Examples:

LCS for input Sequences “ABCDGH” and “AEDFHR” is “ADH” of length 3.

LCS for input Sequences “AGGTAB” and “GXTXAYB” is “GTAB” of length 4.

The naive solution for this problem is to generate all subsequences of both given sequences and find the longest matching subsequence. This solution is exponential in term of time complexity. Let us see how this problem possesses both important properties of a Dynamic Programming (DP) Problem.

1) Optimal Substructure:

Let the input sequences be $X[0..m-1]$ and $Y[0..n-1]$ of lengths m and n respectively. And let $L(X[0..m-1], Y[0..n-1])$ be the length of LCS of the two sequences X and Y . Following is the recursive definition of $L(X[0..m-1], Y[0..n-1])$.

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 do not match (or $X[m-1] != Y[n-1]$) then
 $L(X[0..m-1], Y[0..n-1]) = \text{MAX} (L(X[0..m-2], Y[0..n-1]), L(X[0..m-1], Y[0..n-2]))$

Examples:

1) Consider the input strings “AGGTAB” and “GXTXAYB”. Last characters match for the strings. So length of LCS can be written as:

$L(\text{"AGGTAB"}, \text{"GXTXAYB"}) = 1 + L(\text{"AGGTA"}, \text{"GXTXAY"})$

2) Consider the input strings "ABCDGH" and "AEDFHR". Last characters do not match for the strings. So length of LCS can be written as:

$L(\text{"ABCDGH"}, \text{"AEDFHR"}) = \text{MAX} (L(\text{"ABCDG"}, \text{"AEDFHR"}), L(\text{"ABCDGH"}, \text{"AEDFH"}))$

So the LCS problem has optimal substructure property as the main problem can be solved using solutions to subproblems.

2) Overlapping Subproblems:

Following is simple recursive implementation of the LCS problem. The implementation simply follows the recursive structure mentioned above.

```
/* A Naive recursive implementation of LCS problem */
```

```
#include<stdio.h>
```

```
#include<stdlib.h>
```

```
int max(int a, int b);
```

```
/* 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;
}
```

```
/* Driver program to test above function */
```

```
int main()
```

```
{
    char X[] = "AGGTAB";
    char Y[] = "GXTXAYB";
```

```
    int m = strlen(X);
```

```
    int n = strlen(Y);
```

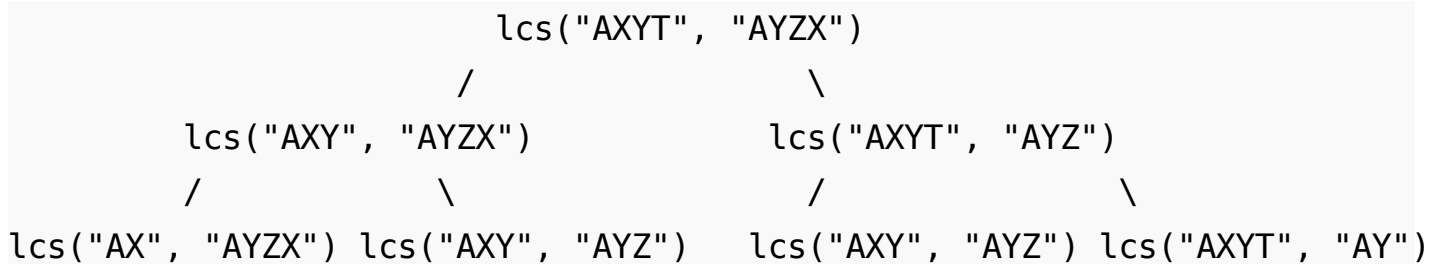
```
    printf("Length of LCS is %d\n", lcs( X, Y, m, n ) );
```

```

    getchar();
    return 0;
}

```

Time complexity of the above naive recursive approach is $O(2^n)$ in worst case and worst case happens when all characters of X and Y mismatch i.e., length of LCS is 0. Considering the above implementation, following is a partial recursion tree for input strings "AXYT" and "AYZX"



In the above partial recursion tree, `lcs("AXY", "AYZ")` is being solved twice. If we draw the complete recursion tree, then we can see that there are many subproblems 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. Following is a tabulated implementation for the LCS problem.

```

/* Dynamic Programming implementation of LCS problem */
#include<stdio.h>
#include<stdlib.h>

```

```

int max(int a, int b);

```

```

/* 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
       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
return L[m][n];
}


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

/* Driver program to test above function */
int main()
{
    char X[] = "AGGTAB";
    char Y[] = "GXTXAYB";

    int m = strlen(X);
    int n = strlen(Y);

    printf("Length of LCS is %d\n", lcs( X, Y, m, n ) );

    getchar();
    return 0;
}
```



Time Complexity of the above implementation is $O(mn)$ which is much better than the worst case time complexity of Naive Recursive implementation.

The above algorithm/code returns only length of LCS. Please see the following post for printing the LCS.

[Printing Longest Common Subsequence](#)