

Dynamic Programming | Set 12

(Longest Palindromic Subsequence)

Given a sequence, find the length of the longest palindromic subsequence in it. For example, if the given sequence is "BBABCBCAB", then the output should be 7 as "BABCBAB" is the longest palindromic subsequence in it. "BBBBB" and "BBCBB" are also palindromic subsequences of the given sequence, but not the longest ones.

The naive solution for this problem is to generate all subsequences of the given sequence and find the longest palindromic 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 and can efficiently solved using Dynamic Programming.

1) Optimal Substructure:

Let $X[0..n-1]$ be the input sequence of length n and $L(0, n-1)$ be the length of the longest palindromic subsequence of $X[0..n-1]$.

If last and first characters of X are same, then $L(0, n-1) = L(1, n-2) + 2$.

Else $L(0, n-1) = \text{MAX}(L(1, n-1), L(0, n-2))$.

Following is a general recursive solution with all cases handled.

```
// Every single character is a palindrom of length 1
L(i, i) = 1 for all indexes i in given sequence

// IF first and last characters are not same
If (X[i] != X[j]) L(i, j) = max{L(i + 1, j), L(i, j - 1)}

// If there are only 2 characters and both are same
Else if (j == i + 1) L(i, j) = 2

// If there are more than two characters, and first and last
// characters are same
Else L(i, j) = L(i + 1, j - 1) + 2
```

2) Overlapping Subproblems

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

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

// A utility function to get max of two integers
int max (int x, int y) { return (x > y)? x : y; }

// Returns the length of the longest palindromic subsequence in seq
int lps(char *seq, int i, int j)
{
    // Base Case 1: If there is only 1 character
    if (i == j)
        return 1;

    // Base Case 2: If there are only 2 characters and both are same
    if (seq[i] == seq[j] && i + 1 == j)
        return 2;
```

```
// If the first and last characters match
if (seq[i] == seq[j])
    return lps (seq, i+1, j-1) + 2;

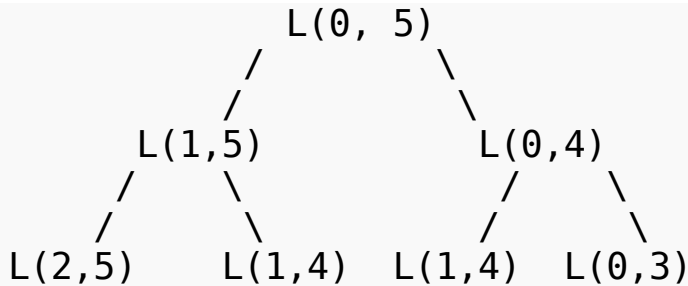
// If the first and last characters do not match
return max( lps(seq, i, j-1), lps(seq, i+1, j) );
}

/* Driver program to test above functions */
int main()
{
    char seq[] = "GEEKSFORGEESK";
    int n = strlen(seq);
    printf ("The length of the LPS is %d", lps(seq, 0, n-1));
    getchar();
    return 0;
}
```

Output:

The length of the LPS is 5

Considering the above implementation, following is a partial recursion tree for a sequence of length 6 with all different characters.



In the above partial recursion tree, $L(1, 4)$ 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. Since same subproblems are called again, this problem has Overlapping Subproblems property. So LPS problem has both properties (see [this](#) and [this](#)) of a dynamic programming problem. Like other typical [Dynamic Programming\(DP\) problems](#), recomputations of same subproblems can be avoided by constructing a temporary array $L[][]$ in bottom up manner.

Dynamic Programming Solution

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

// A utility function to get max of two integers
int max (int x, int y) { return (x > y)? x : y; }

// Returns the length of the longest palindromic subsequence in seq
int lps(char *str)
{
    int n = strlen(str);
    int i, j, cl;
    int L[n][n]; // Create a table to store results of subproblems

    // Strings of length 1 are palindrome of length 1
    for (i = 0; i < n; i++)
        L[i][i] = 1;

    // Build the table. Note that the lower diagonal values of table are
    // useless and not filled in the process. The values are filled in a
    // manner similar to Matrix Chain Multiplication DP solution (See
    // http://www.geeksforgeeks.org/archives/15553). cl is length of
```

```

// substring
for (cl=2; cl<=n; cl++)
{
    for (i=0; i<n-cl+1; i++)
    {
        j = i+cl-1;
        if (str[i] == str[j] && cl == 2)
            L[i][j] = 2;
        else if (str[i] == str[j])
            L[i][j] = L[i+1][j-1] + 2;
        else
            L[i][j] = max(L[i][j-1], L[i+1][j]);
    }
}

return L[0][n-1];
}

```

```

/* Driver program to test above functions */
int main()
{
    char seq[] = "GEEKS FOR GEEKS";
    int n = strlen(seq);
    printf ("The length of the LPS is %d", lps(seq));
    getchar();
    return 0;
}

```

Output:

The length of the LPS is 7

Time Complexity of the above implementation is $O(n^2)$ which is much better than the worst case time complexity of Naive Recursive implementation.

This problem is close to the **Longest Common Subsequence (LCS) problem**. In fact, we can use LCS as a subroutine to solve this problem. Following is the two step solution that uses LCS.

- 1) Reverse the given sequence and store the reverse in another array say rev[0..n-1]
- 2) LCS of the given sequence and rev[] will be the longest palindromic sequence.

This solution is also a $O(n^2)$ solution.