Longest Common Subsequence (LCS)

Name: Cristhian Wiki Sánchez Sauñe

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”.

In order to find out the complexity of brute force approach, we need to first know the number of possible different subsequences of a string with length n, i.e., find the number of subsequences with lengths ranging from 1, 2, .. n-1.

LCS Problem Statement

Recall from theory of permutation and combination that number of combinations with 1 element are ${}^nC_1$. Number of combinations with 2 elements are ${}^nC_2$ and so forth and so on.

$$ {}^nC_0 + {}^nC_1 + {}^nC_2 + \dots {}^nC_n = 2^n $$

So a string of length ***n*** has 2^n-1 different possible subsequences since we do not consider the subsequence with length 0.

LCS Problem Statement

This implies that the time complexity of the brute force approach will be $O(n * 2^n)$.

Take **O(n)** time to check if a subsequence is common to both the strings.

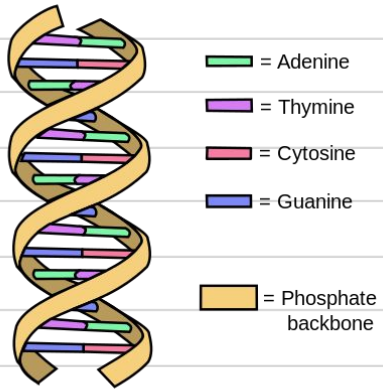
Examples:

- LCS for input Sequences “ABCDGH” and “AEDFHR” is “ADH” of length 3.
- LCS for input Sequences “AGGTAB” and “GXTXAYB” is “GTAB” of length 4.

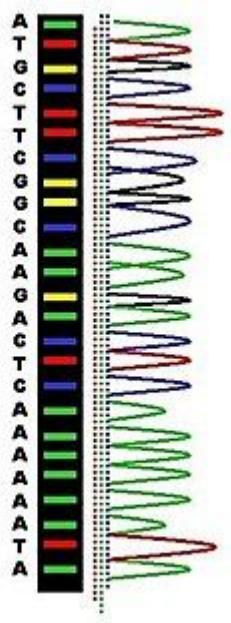
Applications

1. **Molecular biology**

DNA sequences (genes) can be represented as sequences of four letters ACGT, corresponding to the four sub molecules forming DNA. When biologists find a new sequences, they typically want to know what other sequences it is most similar to. One way of computing how similar two sequences are is to find the length of their longest common subsequence.



DNA

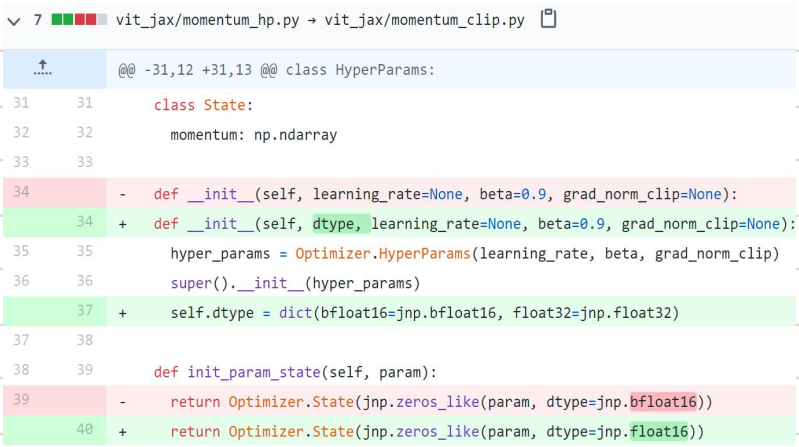


Applications

2. File comparison

Github has an option that allows you to two different versions of the same file, to determine what changes have been made to the file.

It works by finding a longest common subsequence of the lines of the two files; any line in the subsequence has not been changed, so what it displays is the remaining set of lines that have changed.



Dynamic Programming

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]) = *max* (L(X[0..m-2], Y[0..n-1]), L(X[0..m-1], Y[0..n-2]))

Dynamic Programming

Examples:

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

L(“AGGTAB”, “GXTXAYB”) = 1 + L(“AGGTA”, “GXTXAY”)

b) Consider the input strings “DGH” and “FHR. Last characters do not match for the strings. So length of LCS can be written as:

L(“DGH”, “FHR”) = *max* (L(“DG”, “FH**R**”), L(“DG**H**”, “FH”))

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

	A	G	G	T	A	B
G	-	-	4	-	-	-
X	-	-	-	-	-	-
T	-	-	-	3	-	-
X	-	-	-	-	-	-
A	-	-	-	-	2	-
Y	-	-	-	-	-	-
B	-	-	-	-	-	1

Dynamic Programming

2. **Overlapping Subproblems:**

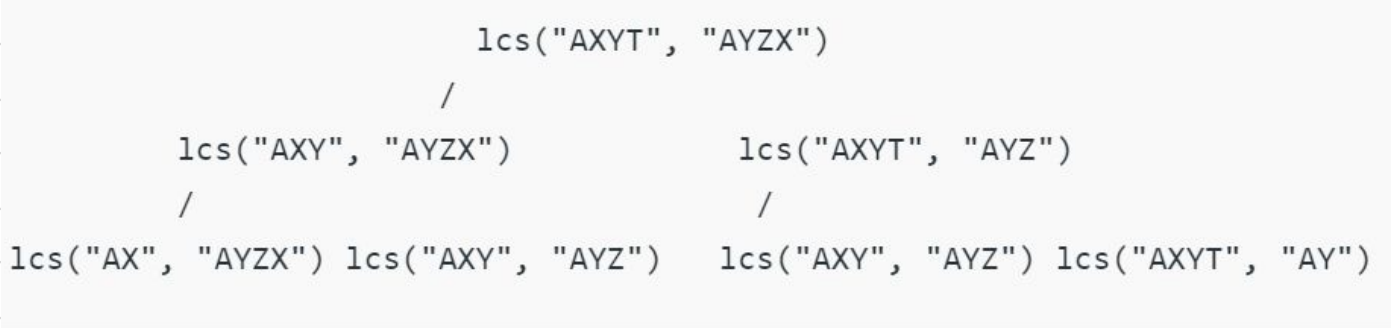
   Following is simple recursive implementation of the LCS problem. The implementation simply follows the recursive structure mentioned above.

```python
# A Naive recursive Python implementation of LCS problem
def lcs(X, Y, m, n):

    if m == 0 or n == 0:
       return 0;
    elif 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));


# Main function
if __name__ == '__main__':
    X = "AGGTAB"
    Y = "GXTXAYB"
    print "Length of LCS is ", lcs(X , Y, len(X), len(Y))
```

Dynamic Programming

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.

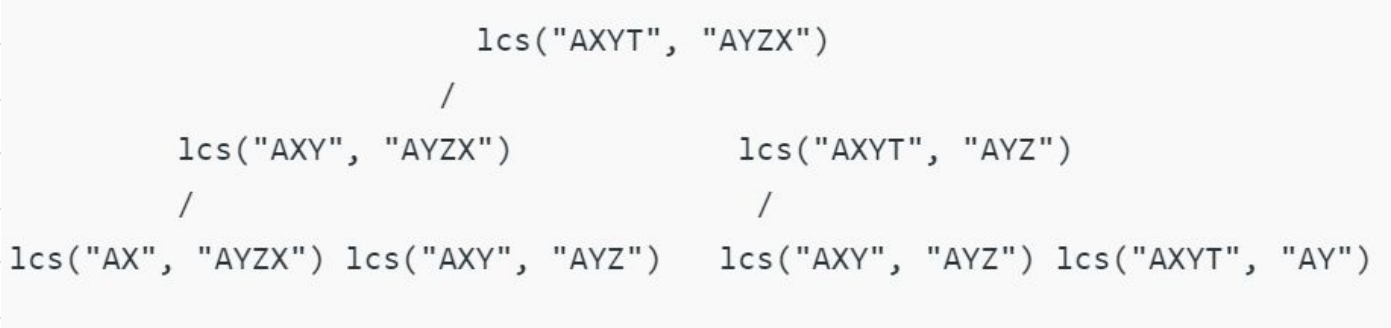
Considering the above implementation, following is a partial recursion tree for input strings “AXYT” and “AYZX”.



Dynamic Programming

In the 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**.



Dynamic Programming

```python
# Dynamic Programming implementation of LCS problem
def lcs(X , Y):
    # find the length of the strings
    m = len(X)
    n = len(Y)

    # declaring the array for storing the dp values
    L = [[None]*(n+1) for i in range(m+1)]

    """Following steps build L[m+1][n+1] in bottom up fashion
    Note: L[i][j] contains length of LCS of X[0..i-1]
    and Y[0..j-1]"""
    for i in range(m+1):
        for j in range(n+1):
            if i == 0 or j == 0 :
                L[i][j] = 0
            elif 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 the length of LCS of X[0..n-1] & Y[0..m-1]
    return L[m][n]
```