LCS

Tuesday, December 7, 2021 11:59 AM

Longest Common Subsequence Problem

The Longest Common Subsequence (LCS) problem is finding the longest subsequence present in given two sequences in the same order, i.e., find the longest sequence which can be obtained from the first original sequence by deleting some items and from the second original sequence by deleting other items.

The problem differs from the problem of finding the longest common substring. Unlike substrings, subsequences are not required to occupy consecutive positions within the original string.

For example, consider the two following sequences, X and Y:

```
X: ABCBDAB


Y: BDCABA




The length of the LCS is 4


LCS are BDAB, BCAB, and BCBA
```

Practice this problem

A naive solution is to check if every subsequence of X[1…m] to see if it is also a subsequence of Y[1…n]. As there are 2m subsequences possible of X, the time complexity of this solution would be O(n.2m), where m is the length of the first string and n is the length of the second string.

The LCS problem has optimal substructure. That means the problem can be broken down into smaller, simple “subproblems”, which can be broken down into yet simpler subproblems, and so on, until, finally, the solution becomes trivial.

1. Let’s consider two sequences, X and Y, of length m and n that both end in the same element.

To find their LCS, shorten each sequence by removing the last element, find the LCS of the shortened sequences, and that LCS append the removed element. So, we can say that.

```
LCS(X[1…m], Y[1…n]) = LCS(X[1…m-1], Y[1…n-1]) + X[m]    if
X[m] = Y[n]
```

2. Now suppose that the two sequences does not end in the same symbol.

Then the LCS of X and Y is the longer of the two sequences LCS(X[1…m-1], Y[1…n]) and LCS(X[1…m], Y[1…n-1]). To understand this property, let's consider the two following sequences:

```
X: ABCBDAB (n elements)


Y: BDCABA (m elements)
```

The LCS of these two sequences either ends with B (the last element of the sequence X) or does not.

Case 1: If LCS ends with B, then it cannot end with A, and we can remove A from the sequence Y, and the problem reduces to LCS(X[1…m], Y[1…n-1]).
Case 2: If LCS does not end with B, then we can remove B from sequence X and the problem reduces to LCS(X[1…m-1], Y[1…n]). For example,

```
LCS(ABCBDAB, BDCABA) = maximum (LCS(ABCBDA, BDCABA),
LCS(ABCBDAB, BDCAB))



LCS(ABCBDA, BDCABA) = LCS(ABCBD, BDCAB) + A


LCS(ABCBDAB, BDCAB) = LCS(ABCBDA, BDCA) + B



LCS(ABCBD, BDCAB) = maximum (LCS(ABCB, BDCAB), LCS(ABCBD,
BDCA))


LCS(ABCBDA, BDCA) = LCS(ABCBD, BDC) + A



And so on…
```

The following solution in C++, Java, and Python find the length of LCS of sequences X[0…m-1] and Y[0…n-1] recursively using the LCS problem's optimal substructure property:

- C++
- Java
- Python

```
class Main
{
    // Function to find the length of the longest common
subsequence of
    // sequences `X[0…m-1]` and `Y[0…n-1]`
    public static int LCSLength(String X, String Y, int m,
int n)
    {
        // return if the end of either sequence is reached
        if (m == 0 || n == 0) {
            return 0;
        }

        // if the last character of `X` and `Y` matches
        if (X.charAt(m - 1) == Y.charAt(n - 1)) {
            return LCSLength(X, Y, m - 1, n - 1) + 1;
        }

        // otherwise, if the last character of `X` and `Y`
don't match
        return Integer.max(LCSLength(X, Y, m, n - 1),
                        LCSLength(X, Y, m - 1, n));
    }

    public static void main(String[] args)
    {
        String X = "ABCBDAB", Y = "BDCABA";

        System.out.println("The length of the LCS is "
                + LCSLength(X, Y, X.length(), Y.length()));
    }
}
```

Download Run Code

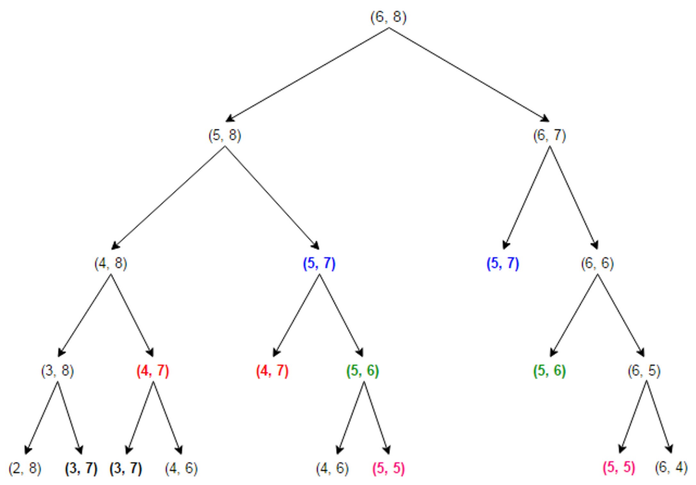
Output:

```
The length of the LCS is 4
```

The worst-case time complexity of the above solution is O(2(m+n)) and occupies space in the call stack, where m and n are the length of the strings X and Y. The worst case happens when there is no common subsequence present in X and Y (i.e., LCS is 0), and each recursive call will end up in two recursive calls.

The LCS problem exhibits overlapping subproblems. A problem is said to have overlapping subproblems if the recursive algorithm for the problem solves the same subproblem repeatedly rather than generating new subproblems.

Let's consider the recursion tree for two sequences of length 6 and 8 whose LCS is 0.



As we can see, the same subproblems (highlighted in the same color) are getting computed repeatedly. We know that problems having optimal substructure and overlapping subproblems can be solved by dynamic programming, in which subproblem solutions are memoized rather than computed repeatedly. This method is demonstrated below in C++, Java, and Python:

- C++
- Java
- Python

```java
import java.util.HashMap;
import java.util.Map;

class Main
{
    // Function to find the length of the longest common
subsequence of substring
    // `X[0…m-1]` and `Y[0…n-1]`
    public static int LCSLength(String X, String Y, int m,
int n,
                                Map<String, Integer>
lookup)
    {
        // return if the end of either string is reached
        if (m == 0 || n == 0) {
            return 0;
        }

        // construct a unique map key from dynamic elements
of the input
        String key = m + "|" + n;

        // if the subproblem is seen for the first time,
solve it and
        // store its result in a map
        if (!lookup.containsKey(key))
        {
            // if the last character of `X` and `Y` matches
            if (X.charAt(m - 1) == Y.charAt(n - 1)) {
                lookup.put(key, LCSLength(X, Y, m - 1, n -
1, lookup) + 1);
            }
            else {
                // otherwise, if the last character of `X`
and `Y` don't match
                lookup.put(key, Integer.max(LCSLength(X, Y,
m, n-1, lookup),
                        LCSLength(X, Y, m - 1, n,
lookup)));
            }
        }

        // return the subproblem solution from the map
        return lookup.get(key);
    }

    public static void main(String[] args)
    {
        String X = "ABCBDAB", Y = "BDCABA";

        // create a map to store solutions to subproblems
        Map<String, Integer> lookup = new HashMap<>();

        System.out.println("The length of the LCS is "
                + LCSLength(X, Y, X.length(), Y.length(),
lookup));
    }
}
```

Download Run Code

Output:

```
The length of the LCS is 4
```

The time complexity of the above top-down solution is O(m.n) and requires O(m.n) extra space, where m and n are the length of the strings X and Y. Note that we can also use an array instead of a map. Check implementation here.

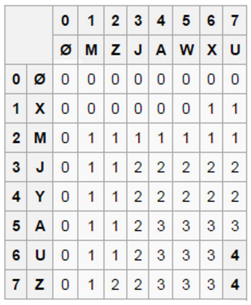
The above *memo*ized version follows the top-down approach since we first break the problem into subproblems and then calculate and store values. We can also solve this problem in a bottom-up manner. In the bottom-up approach, we calculate the smaller values of LCS(i, j) first, then build larger values from them.

```
            | 0                                     if i ==
0 or j == 0


LCS[i][j] = | LCS[i - 1][j - 1] + 1                 if
X[i-1] == Y[j-1]


            | longest(LCS[i - 1][j], LCS[i][j - 1])    if
X[i-1] != Y[j-1]
```

Let X be XMJYAUZ, and Y be MZJAWXU. The longest common subsequence between X and Y is MJAU. The following table is generated by the function LCSLength(), which shows the LCS's length between prefixes of X and Y. The i'th row and j'th column show the LCS's length of substring X[0…i-1] and Y[0…j-1].



		0	1	2	3	4	5	6	7
		Ø	M	Z	J	A	W	X	U
0	Ø	0	0	0	0	0	0	0	0
1	X	0	0	0	0	0	0	1	1
2	M	0	1	1	1	1	1	1	1
3	J	0	1	1	2	2	2	2	2
4	Y	0	1	1	2	2	2	2	2
5	A	0	1	1	2	3	3	3	3
6	U	0	1	1	2	3	3	3	**4**
7	Z	0	1	2	2	3	3	3	**4**

This is demonstrated below in C++, Java, and Python:

- C++
- Java
- Python

```java
class Main
{
    // Function to find the length of the longest common subsequence of substring
    // `X[0…m-1]` and `Y[0…n-1]`
    public static int LCSLength(String X, String Y)
    {
        int m = X.length(), n = Y.length();

        // lookup table stores solution to already computed subproblems,
        // i.e., `T[i][j]` stores the length of LCS of substring
        // `X[0…i-1]` and `Y[0…j-1]`
        int[][] T = new int[m + 1][n + 1];

        // fill the lookup table in a bottom-up manner
        for (int i = 1; i <= m; i++)
        {
            for (int j = 1; j <= n; j++)
            {
                // if the current character of `X` and `Y` matches
                if (X.charAt(i - 1) == Y.charAt(j - 1)) {
                    T[i][j] = T[i - 1][j - 1] + 1;
                }
                // otherwise, if the current character of `X` and `Y` don't match
                else {
                    T[i][j] = Integer.max(T[i - 1][j], T[i][j - 1]);
                }
            }
        }

        // LCS will be the last entry in the lookup table
        return T[m][n];
    }

    public static void main(String[] args)
    {
        String X = "XMJYAUZ", Y = "MZJAWXU";

        System.out.println("The length of the LCS is " + LCSLength(X, Y));
    }
}
```

Download Run Code

Output:

```
The length of the LCS is 4
```

The time complexity of the above bottom-up solution is O(m.n) and requires O(m.n) extra space, where m and n are the length of the strings X and Y. The space complexity of the above solution can be improved to O(n) as calculating LCS of a row of the LCS table requires only the solutions to the current row and the previous row.

Applications of LCS problem

The longest common subsequence problem forms the basis of data comparison programs such as the diff utility and use in the field of bioinformatics. It is also widely used by revision control systems such as Git.

Also See:

- Longest Common Subsequence of k–sequences
- Longest Common Subsequence (LCS) | Space optimized version
- Longest Common Subsequence | Finding all LCS

From <https://www.techiedelight.com/longest-common-subsequence/>

Brute Force

```
class Solution {
    public int longestCommonSubsequence(String text1, String text2) {
        return longestCommonSubsequence(text1, text2, 0, 0);
    }

    private int longestCommonSubsequence(String text1, String text2, int i, int j) {
        if (i == text1.length() || j == text2.length())
            return 0;
        if (text1.charAt(i) == text2.charAt(j))
            return 1 + longestCommonSubsequence(text1, text2, i + 1, j + 1);
        else
            return Math.max(
                longestCommonSubsequence(text1, text2, i + 1, j),
                longestCommonSubsequence(text1, text2, i, j + 1)
            );
    }
}
```

Top-down DP

From <https://leetcode.com/problems/longest-common-subsequence/discuss/590781/From-Brute-Force-To-DP>

We might use memoization to overcome overlapping subproblems.
Since there are two changing values, i.e. `i` and `j` in the recursive function `longestCommonSubsequence`, we might apply a two-dimensional array.

```
class Solution {
    private Integer[][] dp;
    public int longestCommonSubsequence(String text1, String text2) {
        dp = new Integer[text1.length()][text2.length()];
        return longestCommonSubsequence(text1, text2, 0, 0);
    }

    private int longestCommonSubsequence(String text1, String text2, int i, int j) {
        if (i == text1.length() || j == text2.length())
            return 0;

        if (dp[i][j] != null)
            return dp[i][j];

        if (text1.charAt(i) == text2.charAt(j))
            return dp[i][j] = 1 + longestCommonSubsequence(text1, text2, i + 1, j + 1);
        else
            return dp[i][j] = Math.max(
                longestCommonSubsequence(text1, text2, i + 1, j),
                longestCommonSubsequence(text1, text2, i, j + 1)
            );
    }
}
```

Bottom-up DP

For every `i` in text1, `j` in text2, we will choose one of the following two options:

- if two characters match, length of the common subsequence would be 1 plus the length of the common subsequence till the `i-1` and `j-1` indexes
- if two characters doesn't match, we will take the longer by either skipping `i` or `j` indexes

```
class Solution {
    public int longestCommonSubsequence(String text1, String text2) {
        int[][] dp = new int[text1.length() + 1][text2.length() + 1];
        for (int i = 1; i <= text1.length(); i++) {
            for (int j = 1; j <= text2.length(); j++) {
                if (text1.charAt(i - 1) == text2.charAt(j - 1))
                    dp[i][j] = 1 + dp[i - 1][j - 1];
                else
                    dp[i][j] = Math.max(dp[i - 1][j], dp[i][j - 1]);
            }
        }
        return dp[text1.length()][text2.length()];
    }
}
```