Chapter 47: Longest Common Subsequence

Section 47.1: Longest Common Subsequence Explanation

One of the most important implementations of Dynamic Programming is finding out the Longest Common Subsequence. Let's define some of the basic terminologies first.

Subsequence:

A subsequence is a sequence that can be derived from another sequence by deleting some elements without changing the order of the remaining elements. Let's say we have a string **ABC**. If we erase zero or one or more than one character from this string we get the subsequence of this string. So the subsequences of string **ABC** will be {**"A"**, **"B"**, **"C"**, **"AB"**, **"AC"**, **"BC"**, **"ABC"**, **" "**}. Even if we remove all the characters, the empty string will also be a subsequence. To find out the subsequence, for each characters in a string, we have two options - either we take the character, or we don't. So if the length of the string is **n**, there are **2n** subsequences of that string.

Longest Common Subsequence:

As the name suggest, of all the common subsequencesbetween two strings, the longest common subsequence(LCS) is the one with the maximum length. For example: The common subsequences between **"HELLOM"** and **"HMLD"** are **"H"**, **"HL"**, **"HM"** etc. Here **"HLL"** is the longest common subsequence which has length 3.

Brute-Force Method:

We can generate all the subsequences of two strings using *backtracking*. Then we can compare them to find out the common subsequences. After we'll need to find out the one with the maximum length. We have already seen that, there are **2n** subsequences of a string of length **n**. It would take years to solve the problem if our **n** crosses **20-25**.

Dynamic Programming Method:

Let's approach our method with an example. Assume that, we have two strings **abcdaf** and **acbcf**. Let's denote these with **s1** and **s2**. So the longest common subsequence of these two strings will be **"abcf"**, which has length 4. Again I remind you, subsequences need not be continuous in the string. To construct **"abcf"**, we ignored **"da"** in **s1** and **"c"** in **s2**. How do we find this out using Dynamic Programming?

We'll start with a table (a 2D array) having all the characters of **s1** in a row and all the characters of **s2** in column. Here the table is 0-indexed and we put the characters from 1 to onwards. We'll traverse the table from left to right for each row. Our table will look like:

```
              0     1     2     3     4     5     6
+-----+-----+-----+-----+-----+-----+-----+-----+
| chʳ |     |  a  |  b  |  c  |  d  |  a  |  f  |
+-----+-----+-----+-----+-----+-----+-----+-----+
0  |     |     |     |     |     |     |     |     |
+-----+-----+-----+-----+-----+-----+-----+-----+
1  |  a  |     |     |     |     |     |     |     |
+-----+-----+-----+-----+-----+-----+-----+-----+
2  |  c  |     |     |     |     |     |     |     |
+-----+-----+-----+-----+-----+-----+-----+-----+
3  |  b  |     |     |     |     |     |     |     |
+-----+-----+-----+-----+-----+-----+-----+-----+
4  |  c  |     |     |     |     |     |     |     |
+-----+-----+-----+-----+-----+-----+-----+-----+
5  |  f  |     |     |     |     |     |     |     |
+-----+-----+-----+-----+-----+-----+-----+-----+
```

Here each row and column represent the length of the longest common subsequence between two strings if we take the characters of that row and column and add to the prefix before it. For example: **Table[2][3]** represents the length of the longest common subsequence between **"ac"** and **"abc"**.

The 0-th column represents the empty subsequence of **s1**. Similarly the 0-th row represents the empty subsequence of **s2**. If we take an empty subsequence of a string and try to match it with another string, no matter how long the length of the second substring is, the common subsequence will have 0 length. So we can fill-up the 0-th rows and 0-th columns with 0's. We get:

```
              0     1     2     3     4     5     6
+-----+-----+-----+-----+-----+-----+-----+-----+
| chr |     |  a  |  b  |  c  |  d  |  a  |  f  |
+-----+-----+-----+-----+-----+-----+-----+-----+
0  |     |  0  |  0  |  0  |  0  |  0  |  0  |  0  |
+-----+-----+-----+-----+-----+-----+-----+-----+
1  |  a  |  0  |     |     |     |     |     |     |
+-----+-----+-----+-----+-----+-----+-----+-----+
2  |  c  |  0  |     |     |     |     |     |     |
+-----+-----+-----+-----+-----+-----+-----+-----+
3  |  b  |  0  |     |     |     |     |     |     |
+-----+-----+-----+-----+-----+-----+-----+-----+
4  |  c  |  0  |     |     |     |     |     |     |
+-----+-----+-----+-----+-----+-----+-----+-----+
5  |  f  |  0  |     |     |     |     |     |     |
+-----+-----+-----+-----+-----+-----+-----+-----+
```

Let's begin. When we're filling **Table[1][1]**, we're asking ourselves, if we had a string **a** and another string **a** and nothing else, what will be the longest common subsequence here? The length of the LCS here will be 1. Now let's look at **Table[1][2]**. We have string **ab** and string **a**. The length of the LCS will be 1. As you can see, the rest of the values will be also 1 for the first row as it considers only string **a** with **abcd**, **abcda**, **abcdaf**. So our table will look like:

```
              0     1     2     3     4     5     6
+-----+-----+-----+-----+-----+-----+-----+-----+
| chr |     |  a  |  b  |  c  |  d  |  a  |  f  |
+-----+-----+-----+-----+-----+-----+-----+-----+
0  |     |  0  |  0  |  0  |  0  |  0  |  0  |  0  |
+-----+-----+-----+-----+-----+-----+-----+-----+
1  |  a  |  0  |  1  |  1  |  1  |  1  |  1  |  1  |
+-----+-----+-----+-----+-----+-----+-----+-----+
2  |  c  |  0  |     |     |     |     |     |     |
+-----+-----+-----+-----+-----+-----+-----+-----+
3  |  b  |  0  |     |     |     |     |     |     |
+-----+-----+-----+-----+-----+-----+-----+-----+
4  |  c  |  0  |     |     |     |     |     |     |
+-----+-----+-----+-----+-----+-----+-----+-----+
5  |  f  |  0  |     |     |     |     |     |     |
+-----+-----+-----+-----+-----+-----+-----+-----+
```

For row 2, which will now include **c**. For **Table[2][1]** we have **ac** on one side and **a** on the other side. So the length of the LCS is 1. Where did we get this 1 from? From the top, which denotes the LCS **a** between two substrings. So what we are saying is, if **s1[2]** and **s2[1]** are not same, then the length of the LCS will be the maximum of the length of

LCS at the **top**, or at the **left**. Taking the length of the LCS at the top denotes that, we don't take the current character from **s2**. Similarly, Taking the length of the LCS at the left denotes that, we don't take the current character from **s1** to create the LCS. We get:

```
              0     1     2     3     4     5     6
+-----+-----+-----+-----+-----+-----+-----+-----+
| chr |     |  a  |  b  |  c  |  d  |  a  |  f  |
+-----+-----+-----+-----+-----+-----+-----+-----+
0  |     |  0  |  0  |  0  |  0  |  0  |  0  |  0  |
+-----+-----+-----+-----+-----+-----+-----+-----+
1  |  a  |  0  |  1  |  1  |  1  |  1  |  1  |  1  |
+-----+-----+-----+-----+-----+-----+-----+-----+
2  |  c  |  0  |  1  |     |     |     |     |     |
+-----+-----+-----+-----+-----+-----+-----+-----+
3  |  b  |  0  |     |     |     |     |     |     |
+-----+-----+-----+-----+-----+-----+-----+-----+
4  |  c  |  0  |     |     |     |     |     |     |
+-----+-----+-----+-----+-----+-----+-----+-----+
5  |  f  |  0  |     |     |     |     |     |     |
+-----+-----+-----+-----+-----+-----+-----+-----+
```

So our first formula will be:

```
if s2[i] is not equal to s1[j]
    Table[i][j] = max(Table[i-1][j], Table[i][j-1]
endif
```

Moving on, for **Table[2][2]** we have string **ab** and **ac**. Since **c** and **b** are not same, we put the maximum of the top or left here. In this case, it's again 1. After that, for **Table[2][3]** we have string **abc** and **ac**. This time current values of both row and column are same. Now the length of the LCS will be equal to the maximum length of LCS so far + 1. How do we get the maximum length of LCS so far? We check the diagonal value, which represents the best match between **ab** and **a**. From this state, for the current values, we added one more character to **s1** and **s2** which happened to be the same. So the length of LCS will of course increase. We'll put **1 + 1 = 2** in **Table[2][3]**. We get,

```
              0     1     2     3     4     5     6
+-----+-----+-----+-----+-----+-----+-----+-----+
| chr |     |  a  |  b  |  c  |  d  |  a  |  f  |
+-----+-----+-----+-----+-----+-----+-----+-----+
0  |     |  0  |  0  |  0  |  0  |  0  |  0  |  0  |
+-----+-----+-----+-----+-----+-----+-----+-----+
1  |  a  |  0  |  1  |  1  |  1  |  1  |  1  |  1  |
+-----+-----+-----+-----+-----+-----+-----+-----+
2  |  c  |  0  |  1  |  1  |  2  |     |     |     |
+-----+-----+-----+-----+-----+-----+-----+-----+
3  |  b  |  0  |     |     |     |     |     |     |
+-----+-----+-----+-----+-----+-----+-----+-----+
4  |  c  |  0  |     |     |     |     |     |     |
+-----+-----+-----+-----+-----+-----+-----+-----+
5  |  f  |  0  |     |     |     |     |     |     |
+-----+-----+-----+-----+-----+-----+-----+-----+
```

So our second formula will be:

```
if s2[i] equals to s1[j]
    Table[i][j] = Table[i-1][j-1] + 1
endif
```

We have defined both the cases. Using these two formulas, we can populate the whole table. After filling up the table, it will look like this:

```
              0     1     2     3     4     5     6
+-----+-----+-----+-----+-----+-----+-----+-----+
| chr |     |  a  |  b  |  c  |  d  |  a  |  f  |
+-----+-----+-----+-----+-----+-----+-----+-----+
0  |     |  0  |  0  |  0  |  0  |  0  |  0  |  0  |
+-----+-----+-----+-----+-----+-----+-----+-----+
1  |  a  |  0  |  1  |  1  |  1  |  1  |  1  |  1  |
+-----+-----+-----+-----+-----+-----+-----+-----+
2  |  c  |  0  |  1  |  1  |  2  |  2  |  2  |  2  |
+-----+-----+-----+-----+-----+-----+-----+-----+
3  |  b  |  0  |  1  |  2  |  2  |  2  |  2  |  2  |
+-----+-----+-----+-----+-----+-----+-----+-----+
4  |  c  |  0  |  1  |  2  |  3  |  3  |  3  |  3  |
+-----+-----+-----+-----+-----+-----+-----+-----+
5  |  f  |  0  |  1  |  2  |  3  |  3  |  3  |  4  |
+-----+-----+-----+-----+-----+-----+-----+-----+
```

The length of the LCS between **s1** and **s2** will be **Table[5][6] = 4**. Here, 5 and 6 are the length of **s2** and **s1** respectively. Our pseudo-code will be:

```
Procedure LCSlength(s1, s2):
Table[0][0] = 0
for i from 1 to s1.length
    Table[0][i] = 0
endfor
for i from 1 to s2.length
    Table[i][0] = 0
endfor
for i from 1 to s2.length
    for j from 1 to s1.length
        if s2[i] equals to s1[j]
            Table[i][j] = Table[i-1][j-1] + 1
        else
            Table[i][j] = max(Table[i-1][j], Table[i][j-1])
        endif
    endfor
endfor
Return Table[s2.length][s1.length]
```

The time complexity for this algorithm is: **O(mn)** where **m** and **n** denotes the length of each strings.

How do we find out the longest common subsequence? We'll start from the bottom-right corner. We will check from where the value is coming. If the value is coming from the diagonal, that is if **Table[i-1][j-1]** is equal to **Table[i][j] - 1**, we push either **s2[i]** or **s1[j]** (both are the same) and move diagonally. If the value is coming from top, that means, if **Table[i-1][j]** is equal to **Table[i][j]**, we move to the top. If the value is coming from left, that means, if **Table[i][j-1]** is equal to **Table[i][j]**, we move to the left. When we reach the leftmost or topmost column, our search ends. Then we pop the values from the stack and print them. The pseudo-code:

```
Procedure PrintLCS(LCSlength, s1, s2)
temp := LCSlength
S = stack()
i := s2.length
j := s1.length
while i is not equal to 0 and j is not equal to 0
     if Table[i-1][j-1] == Table[i][j] - 1 and s1[j]==s2[i]
```

```
        S.push(s1[j])   //or S.push(s2[i])
        i := i - 1
        j := j - 1
    else if Table[i-1][j] == Table[i][j]
        i := i-1
    else
        j := j-1
    endif
endwhile
while S is not empty
    print(S.pop)
endwhile
```

Point to be noted: if both **Table[i-1][j]** and **Table[i][j-1]** is equal to **Table[i][j]** and **Table[i-1][j-1]** is not equal to **Table[i][j] - 1**, there can be two LCS for that moment. This pseudo-code doesn't consider this situation. You'll have to solve this recursively to find multiple LCSs.

The time complexity for this algorithm is: **O(max(m, n))**.