CS 381 HW 3

Ankur Dhoot

September 28, 2016

Q1

(a) Using Algorithm LCS-LENGTH (page 394) on X = WASHINGTON and Y = MINNESOTA:

c =

$$\begin{bmatrix}
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 \\
0 & 0 & 0 & 0 & 0 & 0 & 1 & 1 & 1 & 1 \\
0 & 0 & 0 & 0 & 0 & 0 & 1 & 1 & 1 & 1 \\
0 & 0 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 \\
0 & 0 & 1 & 2 & 2 & 2 & 2 & 2 & 2 & 2 \\
0 & 0 & 1 & 2 & 2 & 2 & 2 & 2 & 2 & 2 \\
0 & 0 & 1 & 2 & 2 & 2 & 2 & 2 & 3 & 3 \\
0 & 0 & 1 & 2 & 2 & 2 & 2 & 3 & 3 & 3 \\
0 & 0 & 1 & 2 & 3 & 3 & 3 & 3 & 3 & 3
\end{bmatrix}$$

b =

$$\begin{bmatrix}
\uparrow & \uparrow & \uparrow & \uparrow & \uparrow & \uparrow & \uparrow & \uparrow & \uparrow \\
\uparrow & \uparrow & \uparrow & \uparrow & \uparrow & \uparrow & \uparrow & \uparrow & \nwarrow \\
\uparrow & \uparrow & \uparrow & \uparrow & \uparrow & \nwarrow & \leftarrow & \leftarrow & \uparrow \\
\uparrow & \uparrow & \uparrow & \uparrow & \uparrow & \uparrow & \uparrow & \uparrow & \uparrow \\
\uparrow & \nwarrow & \leftarrow & \leftarrow & \leftarrow & \uparrow & \uparrow & \uparrow & \uparrow \\
\uparrow & \uparrow & \nwarrow & \nwarrow & \leftarrow & \leftarrow & \leftarrow & \leftarrow & \leftarrow \\
\uparrow & \uparrow & \uparrow & \uparrow & \uparrow & \uparrow & \uparrow & \uparrow & \uparrow \\
\uparrow & \uparrow & \uparrow & \uparrow & \uparrow & \uparrow & \uparrow & \nwarrow & \leftarrow \\
\uparrow & \uparrow & \uparrow & \uparrow & \uparrow & \uparrow & \nwarrow & \uparrow & \uparrow \\
\uparrow & \uparrow & \nwarrow & \nwarrow & \leftarrow & \leftarrow & \uparrow & \uparrow & \uparrow
\end{bmatrix}$$

(b) If we were to add a print statement that prints the indices on which PRINT-LCS were called, the trace would be as follows:

PRINT-LCS called with i = 10 and j = 9

PRINT-LCS called with i = 9 and j = 9

PRINT-LCS called with i = 8 and j = 9
PRINT-LCS called with i = 8 and j = 8
PRINT-LCS called with i = 7 and j = 7
PRINT-LCS called with i = 6 and j = 7
PRINT-LCS called with i = 6 and j = 6
PRINT-LCS called with i = 6 and j = 5
PRINT-LCS called with i = 6 and j = 4
PRINT-LCS called with i = 5 and j = 3
PRINT-LCS called with i = 5 and j = 2
PRINT-LCS called with i = 4 and j = 1
PRINT-LCS called with i = 3 and j = 1
PRINT-LCS called with i = 2 and j = 1
PRINT-LCS called with i = 1 and j = 1
PRINT-LCS called with i = 0 and j = 1

Anytime both indices reduce by 1 corresponds to a match on the last character of that subproblem. Therefore, when i = 8, 6, 5 which corresponds to T, N, I. Thus, PRINT-LCS will return INT as the longest common subsequence.

(c) The characters matched in Y = MINNESOTA are at indices 2,4,8 = INT (assuming 1 based indexing as in the book). However, it's clear that indices 2,3,8 = INT as well. Why wasn't 2,3,8 returned as the LCS? PRINT-LCS follows the arrows in the b table, and thus proceeds in a bottom to top, right to left manner. When Y[4] = N, X[6] = N, so we're at position (6,4) in the table. Since c[6, 4] = c[6,3], but position (6,4) is farther right than position (6,3), the algorithm matches (6,4) first. Essentially, the algorithm wants to match characters as far back in the two strings as possible in order to optimize the number of matches. Thus, even though Y[2,3,8] = INT = Y[2,4,8], the algorithm will choose the match that is further back in the string.

Q2

Preliminary Notation:

Let $X_{i,j}$ denote the string with characters $x_i x_{i+1} ... x_j$. Let c[i, j] denote the number of ways one one could break the string $X_{i,j}$ into sequences of words. Thus, our end goal is to compute c[1,n].

Basic Idea:

Similar to the rod cutting problem, we'll make an initial cut at index m that corresponds to the first word in a sequence. If $X_{1,m}$ is a word, then we need to know how many sequences of words are possible in $X_{m+1,n}$. But by definition, this is just c[m+1,n]. We then try all possible cuts for a first word. Of those cuts which yield a valid first word, we sum the number

of sequences possible starting with that word (i.e the number of possible sequences in $X_{m+1,n}$ for each m that yields a valid first word). This sum then gives us the desired total.

We'll let the size of the subproblem be the length of the string for which we're trying to compute the number of word sequences. Based on the idea above, we can write the recursive formula

c[i, j] =

$$\begin{cases} 1 & j = i + 1 \\ I_{i,i}c[i+1,j] + I_{i,i+1}c[i+2,j]... + I_{i,j}c[j+1,j] & j \leq i \end{cases}$$

where $I_{i,j}$ is the indicator variable (i.e = 0 or 1) denoting whether $X_{i,j}$ is a word. As can be seen, a subproblem only depends on solutions to smaller subproblems. Thus, we can use dynamic programming!!

Algorithm 1 Compute total number of word sequences into which X can be broken

```
1:  function WORD-SEQUENCES(String X)
2:      n = X.length
3:      let c[1...n+1, 1...n] be a new table
4:      for s = 1 to n do   ▷ initialize table, empty string has one sequence
5:          c[s + 1, s] = 1
6:      end for
7:      for j = 1 to n do                              ▷ j is the subproblem size
8:          for i = 1 to n - j + 1 do        ▷ i is starting index of subproblem
9:              end = i + j - 1                ▷ end is last index of subproblem
10:             c[i, end] = 0
11:             for k = 0 to j - 1 do        ▷ iterate over all possible first cuts
12:                 if isWord(X_{i,i+k}) then
13:                     c[i, end] += c[i + k + 1, end]
14:                 end if
15:             end for
16:         end for
17:     end for
18:     return c[1,n]
19: end function
```

We initialize the number of sequences of an empty string to be 1. We do this so that if the entire subproblem is a word, we increment the c[i,j] entry by 1 since there is one sequence possible of the remaining characters (the empty sequence) following that word.

As can be seen, subproblems are computed smallest to largest so that the table has solutions to the smaller subproblems available for constant time lookup in every iteration. The counter j denotes the size of the subproblem.

The counter i denotes the starting index of the subproblem. The counter k ranges over the subproblem size and denotes how far from the starting index we make the cut for the first word. If the cut we make yields a valid word, then the number of sequences possible starting with that first word is just the number of sequences possible in the rest of the string (since we fixed the first word). Adding this over all valid first words gives us each c table entry. At the end, we return c[1,n] since this is the desired total.

Runtime:

We compute all c[i,j] entries in the table of which there are O(n^2). Each table entry, c[i,j], computation takes O(j - i) time. We iterate through all possible cuts (of which there are j - i) and check if the cut yields a valid word (which is constant time by assumption). If we have a valid word, we use preexisting table entries to add to our sum which also takes O(1) time. Thus, each c[i,j] entries takes O(j-i) time which is also O(n). Thus, we compute O(n^2) entries each in time O(n) for a grand total run time of O(n^3).

But wait. There's more!!! If we look at the recurrence defined above, in order to compute c[1,n], we only need subproblems of the form c[i,n]. These subproblems in turn only need subsubproblems of the form c[i', n].

Thus, we don't need to compute all c[i,j] entries in the table. We can get away with just all problems of the form c[i,n]. We can get rid of the second dimension and let c[i] denote the number of word sequences possible starting at index i in the string X (and ending at the end of the string).

Algorithm 2 Compute total number of word sequences into which X can be broken, even more efficient

```
 1: function WORD-SEQUENCES(String X)
 2:     n = X.length
 3:     let c[1...n+1] be a new table
 4:     c[n+1] = 1
 5:     for j = 1 to n do                          ▷ j is the subproblem size
 6:         start = n - j + 1
 7:         c[start] = 0
 8:         for k = 0 to j - 1 do          ▷ iterate over all possible first cuts
 9:             if isWord(X_{start,start+k}) then
10:                 c[start] += c[start + k + 1]
11:             end if
12:         end for
13:     end for
14:     return c[1]
15: end function
```

The new algorithm is almost identical to the previous, except for one major difference: we have no for loop with counter i since for a given j we

know that the only subproblem we need to solve is c[n - j + 1]. We solve subproblems for "right to left". That is, we first solve c[n], then c[n-1], then c[n-2], until we get to c[1] which is our desired quantity.

Runtime: We compute n entries in the c array. All subproblems needed to solve a given problem have already been solved and stored in the array. The maximum number of subproblems we ever need is n (when computing c[1]). Thus, we fill n entries in the array, each in time at most O(n) leading to an O(n^2) runtime.