Lecture 1 exercise solutions

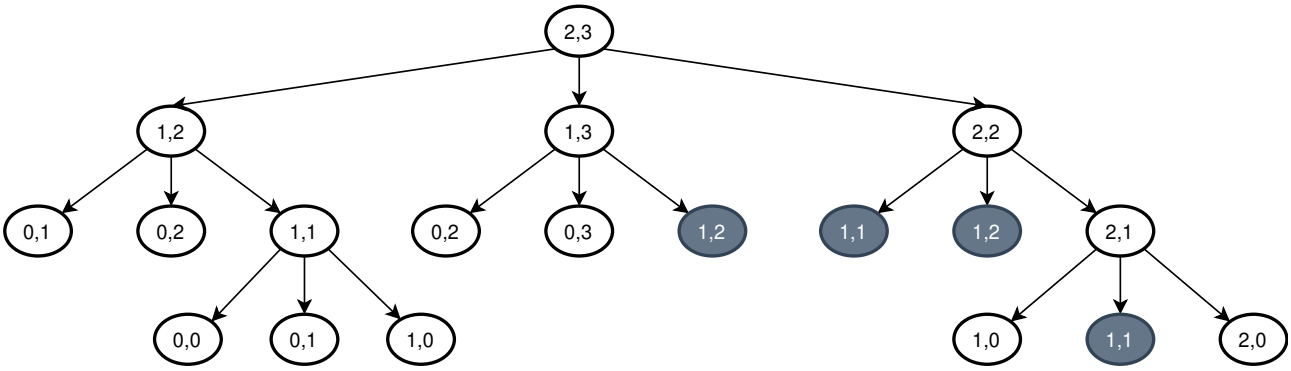
1.

			G	O
	j\i	0	1	2
	0	0	1_D	2_D
L	1	1_I	1_R	2_R
O	2	2_I	2_R	1_C
G	3	3_I	2_C	2_I

The edit distance is 2 and the suggest edits are:

I(2, 'G') R(1, 'L')

Here is the recursion tree for the memoized version of the algorithm. Grey nodes show roots of subtrees that are avoided (already computed and saved in the table) compared to the simple recursive version.



19 recursive calls are made (including the top call) by the memoized algorithm. 37 recursive calls are made by a straightforward (non-memoized) recursive algorithm (we add all the avoided descendant nodes of the grey nodes: 19 + 6+3+6+3 = 37).

2.

Question 2: The second line of the recurrence ($c(i+1, a)$, **if** den[i] > a) is not coded in the pseudocode.

Question 3. If we do not need to print the coins used, just their number, for CoinChange1, we can "forget" the beginning of the array as the algorithm progresses. More specifically, when we are at index a in the array, we will not need to access any indexes smaller than a – den[1]. Remember, den[1] is the largest available denomination. Thus, we can code the algorithm to use at most $min(A$, den[1]) memory. Think how to implement this.

CoinChange2 can save more memory. There, we need to keep track of only ***two*** rows in the table: the current i-th row and the "previous" i+1-st row. Thus, space usage is reduced to $\Theta(A)$.

3.

Here is the recurrence we use:

$$c[i,j]=\begin{cases}0 & \text{if } S_{ij}=\emptyset,\\ \max\limits_{a_k\in S_{ij}}\{c[i,k]+c[k,j]+1\} & \text{if } S_{ij}\neq\emptyset.\end{cases}$$

The memoized version of the algorithm (remember activities are sorted on the end time and there are special sentinel activities $A[0]$ and $A[n+1]$, which do not overlap with anything):

```
ActivitySel1m(A, i, j)
01 if c[i,j] == ∞ then
02    c[i,j] = 0
03    for k = i+1 to j-1 do
04       if not overlaps(A[k], A[i]) and
            not overlaps(A[k], A[j]) then
05          sol = ActivitySel1m(A,i,k) +
                  ActivitySel1m(A,k,j) + 1
06          if sol > c[i,j] then c[i,j] = sol
07 return c[i,j]
```

Here, `overlaps(a,b) = a.f > b.s and b.f > a.s`. The bottom-up solution would have a nested loop structure similar to Matrix-Chain-Order in Section 15.2 of CLRS3. To remember the choices made, one would have to create a table **T** and add `T[i,j] = k` in line 6 of **`ActivySel1m`**. To print out the activities, the following recursive procedure could be used:

```
PrintActivities(c, T, i, j)  // Initial call: PrintActivities(c,T,1,n)
01 if c[i,j] > 0 then
02    PrintActivities(c, T, i, T[i,j])
03    Print T[i,j]
04    PrintActivities(c, T, T[i,j], j)
```

CLRS4 14-2

How a subproblem is defined? How many parameters define it? Which choices have to be considered in each step of the algorithm?

Having a string, any algorithm looking for a longest palindrome subsequence (LPS) is going to match some pairs of symbols (one from each of the two ends) and it is going to skip some symbols that it chooses not to include in the palindrome. Thus, given a string, we can choose either to skip a symbol from one end or to skip a symbol from another end, or to match the two end symbols (if they are equal).

When considering what subproblem each of the choices generates, it is clear that a subproblem is a substring of the original string defined by two parameters—where it starts and where it ends. Here is a recurrence expressing the length of the LPS of $s[i\,..\,j]$. Note, that we include a special trivial case—one symbol string is always a palindrome.

$$P(i,j) = \begin{cases} 0 & \text{if } i > j, \\ 1 & \text{if } i = j, \\ 2 + P(i+1, j-1) & \text{if } i < j \ \wedge \ s[i] = s[j], \\ \max(P(i+1,j), P(i,j-1)) & \text{otherwise.} \end{cases}$$

One could wonder if it is always optimal to match the two end symbols if they are equal (the third line of the recurrence). It is easy to prove by contradiction that this is optimal. Skipping both of the two symbols is obviously suboptimal. Now suppose $s[i] = s[j]$, but we skip $s[j]$ and match $s[i]$ with $s[k]$, where $s[i] = s[k]$ and $k < j$. Then the LPS found by the algorithm would have the length $2 + P(i+1, k-1)$. This can not be larger than $2 + P(i+1, j-1)$, because a substring of a string can not have longer LPS than the string itself.

Here is a bottom-up algorithm to compute the length of the LPS. We use a 2D array of solutions, P. Let i index the rows of P and j index the columns of P. Then, according to the first line of the recurrence, the left-bottom corner of P is all zeros. In the pseudocode we only fill one diagonal, $j = i - 1$, with zeros (line 2), as these are the only cells from the left-bottom corner of P that will be accessed by the rest of the algorithm.

```
LPS(s[1..n])
 1  for i = 2 to n
 2      P[i, i-1] = 0          // the first line of the recurrence
 3  for i = 1 to n
 4      P[i, i] = 1            // the second line of the recurrence
 5  for i = n-1 downto 1       // pay attention to the order of filling the table: bottom to top, left to right
 6      for j = i+1 to n       // only above the diagonal
 7          if s[i] == s[j]
 8              P[i, j] = 2 + P[i+1, j-1]
 9          elseif P[i+1, j] > P[i, j-1]
10              P[i, j] = P[i+1, j]
11          else P[i, j] = P[i, j-1]
12  return P[1, n]
```

Here is a simple recursive algorithm to print the longest palindrome subsequence using table P. Note that we did not record the choices we made in a separate table, but it is easy to see what is the choice for a subproblem (i, j) by making the same check as in line 9 of LPS.

```
PRINT-LPS(s, P, i, j)        // First call: PRINT-LPS(s, P, 1, n)
 1  if i > j return
 2  if i == j
 3       print s[i]
 4  elseif s[i] == s[j]
 5       print s[i]
 6       PRINT-LPS(s, P, i + 1, j − 1)
 7       print s[j]
 8  elseif P[i + 1, j] > P[i, j − 1]
 9       PRINT-LPS(s, P, i + 1, j)
10  else PRINT-LPS(s, P, i, j − 1)
```

The worst-case running time of LPS is clearly $\Theta(n^2)$. The worst-case running time of PRINT-LPS can be described by the recurrence $T(n) = T(n-1) + \Theta(1)$, which solves to $\Theta(n)$.