ICS 311, Fall 2020, Problem Set 07, Topics 12 & 13

This is a 40 point homework. The extra 10 points are extra credit opportunities (the denominator of 1000 points for the semester remains the same), but the last problem is more difficult.

#1 Constructing and extracting the solution for Longest Path (10 pts)

In the following you use the posted solution to `LongestPathValueMemoized` from class last week.

(a) (3 pts) Rewrite `LongestPathValueMemoized` to `LongestPathMemoized` that takes an additional parameter `next[1..|V|]`, and records the next vertex in the path from any given vertex u in `next[u]`. Assume that all entries of `next` are initialized to nil by the caller.

> **Solution:** Additions in Yellow.

```
Longest-Path-Memoized (G, u, t, dist, next)
  if u = t
      dist[u] = 0
      return 0
  if dist[u] > -∞
      return dist[u]
  else
      for each v in G.Adj[u]
          alt = w(u,v)
              + Longest-Path-Memoized(G, v, t, dist, next)
          if dist[u] < alt
              dist[u] = alt
              next[u] = v   // got max so far via v
      return dist[u]
```

> **Other solutions are possible. The important thing is that next is updated to record how we got the max solution.**

(b) (3 pts) Once that is done, write a procedure that recovers (e.g., prints) the path from s to t by tracing through `next`.

> **Solution:** Something like this:
>
> ```
> Print-Path(s, t, next)
>   u = s
>   print u
>   while u != t
>       u = next[u]
>       print "→" u
> ```

(c) (2 pts) What is the asymptotic runtime of your total solution to the Longest Path problem in terms of |V| and |E|? Include all steps (initializing arrays, `LongestPathMemoized`, and printing the solution). *Hint:* Count in aggregate across all calls rather than trying to figure out how many times the loop runs on a given Adj[u].

> **Solution:** O(V + E). We have Θ(V) for initializations of `dist` and `next`. In aggregate across all calls, the "`for each v in G.Adj[u]`" will process O(E) edges. This is because the algorithm is memoized, so we process the edges out of each vertex at most once. `Print-Path` is O(E) at most.

(d) (2 pts) What is the asymptotic use of space of your solution in terms of |V| and |E|?

> **Solution:** We will give one point for the answer of "Θ(V)" and two points for "O(V+E)". We need Θ(V) space for `dist` and `next`. The other explicit data structures appear to be constant. However, space is required for the run time stack when recursive calls are made, and it is possible for a longest path to involve O(E) of the edges in the graph. If the algorithm were written with an explicit stack rather than recursion this would be more obvious.

#2 LCS by Suffix (15 pts)

In this problem you will redo the derivation of the LCS dynamic programming algorithm to be a variation that works on the suffixes rather than the prefixes. The intention is that revising a published derivation will help you see the approach before we ask you to do one "from scratch" on your own.

In Topics 12 Lecture Notes Dynamic Programming - http://www2.hawaii.edu/~suthers/courses/ics311f20/Notes/Topic-12.html#LCS, the Longest Common Subsequence (LCS) problem is analyzed based on the notation:

Xi = prefix $\langle x1, \ldots, xi \rangle$

Yi = prefix $\langle y1, \ldots, yi \rangle$

But the optimal LCS substructure(s) can also be suffixes rather than prefixes. (Informally: It doesn't matter whether we start looking for the substructures from the beginning of the sequences X and Y of from their end.) In the suffix approach the arrows go in a more natural direction.

Below we step you through the derivation of a symmetric version of LCS based on suffixes:

Xi = suffix $\langle xi, \ldots, xm \rangle$

Yj = suffix $\langle yj, \ldots, yn \rangle$

(a) (3 pts) Reformulate ***Theorem 15.1*** for the suffix version by filling in the "…" below:

Let $Z = \langle z_1, \ldots, z_k \rangle$ be any LCS of $X = \langle x_1, \ldots, x_m \rangle$ and $Y = \langle y_1, \ldots, y_n \rangle$. Then

Solutions are in bold. See comment below if you used i and j.

1. **If $x_1 = y_1$, then $z_1 = x_1 = y_1$, and Z_2 is an LCS of X_2 and Y_2.**
2. (If the first characters of X and Y match, then these first characters are also the first character of the LCS Z, so we can discard the first character of all three and continue recursively on the suffix.)
3. **If $x_1 \neq y_1$, then $z_1 \neq x_1 \Rightarrow Z$ is an LCS of X_2 and Y.**
4. **If $x_1 \neq y_1$, then $z_1 \neq y_1 \Rightarrow Z$ is an LCS of X and Y_2.**
5. (If the first characters of X and Y don't match each other, then the suffix Z must be in the substrings not involving these characters, and furthermore we can use the first character of Z to determine which one it lies in.)

Comment: The above is based on the CLRS approach in which it is assumed that m and n are given new values in the recursion: here we assume that x_1 and y_1 are set to the beginning of the (possibly shorter) string(s) in the recursive case. We will also accept solutions that correctly use x_i and y_j instead of x_1 and y_1, in order to index correctly into arrays called by reference rather than by (copied) value.

(To keep this problem set from getting too long we will skip the proof, but it is an easy translation of the proof on page 392 of CLRS.)

(b) (5 pts) Now redefine the **Recursive formulation** accordingly (again, fill in the "___"):

Define $c[i, j]$ = length of LCS of X_i and Y_j . We want to find c[___, ___].

$$c[i,j] = \begin{cases} 0 & \text{if } i = \text{___ or } j = \text{___} \\ c[\text{___} , \text{___}] & \text{if } i \text{ ___ }, j \text{ ___ and } x_i = y_j \\ \max(\text{___}) & \text{if } i \text{ ___ }, j \text{ ___ and } x_i \neq y_j \end{cases}$$

Solution:

Define $c[i, j]$ = length of LCS of X_i and Y_j . We want to find c[**1,1**].

$$c[i,j] = \begin{cases} 0 & \text{if } i = \mathbf{m+1} \text{ or } j = \mathbf{n+1} \\ c[\mathbf{i+1, j+1}] + 1 & \text{if } \mathbf{i \le m}, \mathbf{j \le n} \text{ and } x_i = y_j \\ \max(\mathbf{c[i+1, j], c[i, j+1]}) & \text{if } \mathbf{i \le m}, \mathbf{j \le n} \text{ and } x_i \neq y_j \end{cases}$$

Note that the CLRS formulation started with $c[0,0]$ even though there is no x_0 or y_0. *Hint:* You do not need $c[0,0]$, but your c[] does need indices that go beyond the range of the indices in X and Y.

(c) (5 pts) Write pseudocode for LCS-LENGTH according to your **Recursive formulation**. (*Hint*: the arrows need to be different.)

Solution: should modify the existing code as follows (in yellow):

```
LCS-LENGTH(X, Y, m, n)
for i ← 1 to m do
    c[i, n + 1] ← 0
for j ← 1 to n do
    c[m + 1, j] ← 0
for i ← m downto 1 do
    for j ← n downto 1 do
       if xi = yj then
          c[i, j] ← c[i + 1, j + 1] + 1
          b[i, j] ← "↘"
     else if c[i + 1, j] ≥ c[i, j + 1] then
          c[i, j] ← c[i + 1, j]
```

```
            b[i, j] ← "↓"
        else
            c[i, j] ← c[i, j + 1]
            b[i, j] ← "→"
return c and b
```

(d) (2 pts) In the Notes, the longest subsequence could be only 'printed' with PRINT-LCS and the pseudocode needed recursion. With your 'suffix' method, you can do better and simpler: Write LCS(b,X,m,n) pseudocode that returns the subsequence directly. Use a vector to store the result. A vector is like an array but grows as needed.

Solution: Yours may differ but the logic should follow the arrows from the upper left b[1,1].

```
LCS(b,X,m,n):
S ← []
i ← 1
j ← 1
k ← 1
while (i ≤ m) or (j ≤ n) do
    if b[i, j] = "↘" then
     Sk ← xi
     k ← k + 1
        i ← i + 1
     j ← j + 1
    else if b[i, j] = "↓"
        i ← i + 1
     else
     j ← j + 1
return S
```

#3 Activity Scheduling with Revenue (15 pts)

Activity scheduling problem from the greedy algorithm class: Suppose that different activities earn different amounts of revenue. In addition to their start and finish times s_i and f_i, each activity a_i has revenue r_i, and our objective is now to **maximize the total revenue:**

$$\sum_{a_i \in A} r_i$$

In class you found out that we can't use a greedy algorithm to maximize the revenue from activities, and we noted that dynamic programming will apply. Here you will develop the DP solution following the same steps as for the other problems (e.g., problem 3 above), but you are responsible for the details. Your analysis in (a) and (b) below should mirror that of section 16.1 of CLRS.

(a) (3 pts) Describe the structure of an optimal solution A_{ij} for S_{ij}, as defined in CLRS, and use a cut and paste argument to show that the problem has optimal substructure.

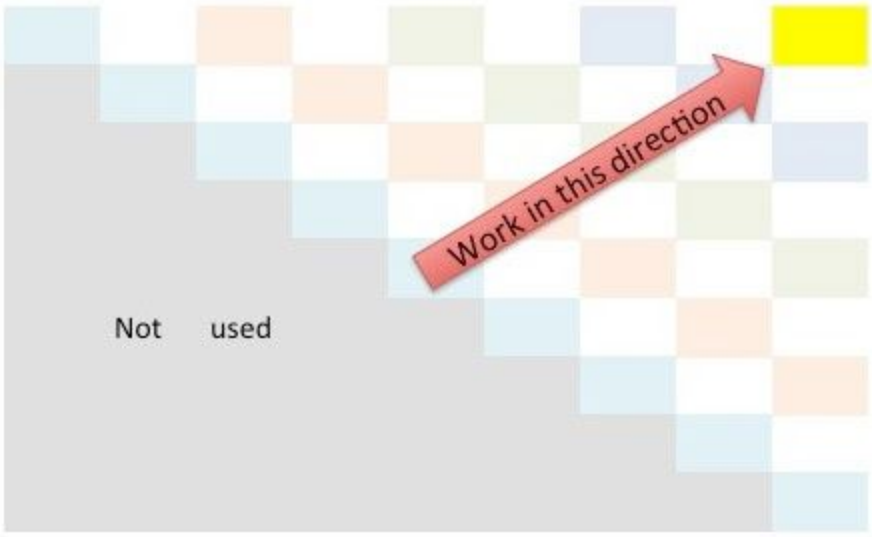
> **Solution:** We follow the analysis of section 16.1. Define the value of a set of compatible events as the sum of values of events in that set. Let S_{ij} be defined as in Section 16.1. An *optimal solution* to S_{ij} is a subset of mutually compatible events of S_{ij} that has maximum value. Let A_{ij} be an optimal solution to S_{ij}, and suppose that A_{ij} includes event a_k. Let A_{ik} and A_k be defined as in Section 16.1 (the remaining portions of the optimal solution on either side of a_k.) Then $A_{ij} = A_{ik} \cup \{a_k\} \cup A_{kj}$, so the value of maximum value set A_{ij} is equal to the value of these three elements.
>
> Then we can apply a cut and paste argument to show that the optimal solution A_{ij} must also include an optimal solution to the two subproblems for S_{ik} and S_{kj}. If we could find a solution for S_{ik} that was of greater value than the value of A_{ik} (or similarly a solution for S_{kj} of value greater than that of A_{kj}), then we could replace A_{ik} (or A_{kj}) with the greater valued subproblem solution and get a greater valued solution to S_{ij}, contradicting the optimality of A_{ij}.

(b) (3 pts) Write a recursive definition of the value val[i,j] of the optimal solution for S_{ij}.

> **Solution:**
>
> $$\text{val}[i,j] = \begin{cases} 0 & \text{if } S_{ij} = \{\} \\ \max_{a_k \in S_{ij}} \{ \text{val}[i,k] + \text{val}[k,j] + r_k \} & \text{if } S_{ij} \neq \{\} \end{cases}$$

(c) (6 pts) Translate that definition into pseudocode that computes the optimal solution. *Hints:* Create fictitious activities a_0 with $f_0 = 0$ and a_{n+1} with $s_{n+1} = \infty$. Define tables val[0..n+1,0..n+1] for the values and activity[0..n+1,0..n+1], where activity[i,j] is the activity *k* that is chosen for A_{ij}. Use a bottom-up approach, filling in the tables for smallest problems first and then by increasing the difference of j-i.

As with any dynamic programming problem, we need to understand the structure of the table in which we store solutions to sub-problems. The smallest problem is on the diagonal, when j=i so the interval is of length 0 and fits no activities. This leads to the initialization val[i,i] = 0. As illustrated below, we then fill in the cells next to the diagonal, for intervals of length j-i=1. Then we fill in the next diagonal, for intervals of length j-i=2, looking up the optional solution in the j-i=1 diagonal if there is space left for subproblems. This continues for each successive diagonal, looking up optimal solutions on the previous diagonals (which optimal substructure allows us to do), until we write the best value at the upper right cell.



Solution (yours may differ in details):

```
Max-Value-Activity-Selector(s, f, v, n)
  let val[0..n+1,0..n+1] and val[0..n+1,0..n+1] be new tables
  for i = 0 to n
      val[i,i] = 0
      val[i, i+1] = 0
  val[n+1, n+1] = 0
  for l = 2 to n + 1
      for i = 0 to n - l + 1
          j = i + 1
          val[i,j] = 0
          k = j - 1
          while f[i] < f[k]
              // if it fits and it is of better value ...
```

```
                if f[i] <= s[k] and
                   f[k] <= s[j] and
                   val[i,k] + val[k,j] + rk > val[i,j]
                    // ... save it
                    val[i,j] = val[i,k] + val[k,j] + rk
                    act[i,j] = k
                k = k - 1
  print "Maximum value set of activities has value: ", val[0,
n+1]
  print "The activities are:"
  Print-Activites(val, act, 0, n+1)
```

(d) (2 pts) Write pseudocode to print out the set of activities chosen.

Solution:

```
Print-Activities(val, act, i, j)
  if val[i,j] > 0
      k = act[i,j]
      print k, " "
      Print-Activities(val, act, i, k)
      Print-Activities(val, act, k, j)
```

(e) (1 pt) What is the asymptotic runtime of your solution including (c) and (d)?

Solution:

Max-Value-Activity-Selector requires $O(n^3)$ time due to the two nested **for** loops and the **while** loop nested within them. The l loop runs n times; the i loop runs up to n times dependent on l, and the while loop potentially runs through all i activities (there is no explicit exit test for k because it will exit at fictitious activity f[0] = 0).