1) Constructing and extracting the solution for Longest Path

(a) **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 1 by the caller.**

Algorithm 1 LongestPathMemoized(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) + LongestPathMemoized(G,v,t,dist,next)
        if dist[u] < alt
            dist[u] = alt
            next[u] = v
    return dist[u]
```

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

Algorithm 2 Recover(s, t, next)

```
s = next[s]
Print s
if node ≠ t
    return Recover(x,t,next)
else
    print t
```

(c) **What is the asymptotic runtime of your total solution to the Longest Path problem in terms of $|V|$ and $|E|$? Include all steps**
The run-time will be $\Theta(|E|)$ because it visits the edges without going through the vertices.

(d) **What is the asymptotic use of space of your solution in terms of $|V|$ and $|E|$?**
The asymptotic use of space in terms of $|V|\&|E|$ is $\Theta(|V| + |E|)$.

2) LCS by Suffix

(a) **Reformulate Theorem 15.1 for the suffix version by filling in the below**

1. If $X_m = Y_n$ then $Z_k = X_m = Y_n$ and Z_{k+1} is an LCS of X_{m+1} and Y_{n+1}
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 $x_m \neq y_n$ then $z_k \neq x_m$ implies that Z is an LCS of X_{m+1} and Y
4. If $x_m \neq y_n$ then $z_k \neq y_n$ implies that Z is an LCS of X and Y_{n+1}
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.)

(b) **Now redefine the Recursive formulation accordingly**

$$c[i,j] = \begin{cases} 0 & \text{if } i > m \vee j > n \\ c[i+1, j+1] & \text{if } i \leq m, j \leq n, \wedge x_i = y_i \\ \max(c[i, j+1], c[i+1, j]) & \text{if} i \leq m, j \leq n, \wedge x_i \neq y_i \end{cases}$$

(c) **Write pseudocode for LCS-LENGTH according to your Recursive formulation.**

Algorithm 3 LCS-LENGTH(X,Y)

```
m = X.length
n = Y.length
let b[1...m,1...n] and c[1...m, 1...n] be new tables
for i = 1 to n + 1
    C[m+1, i] = 0
for j = 1 to m + 1
    C[j, n+1] = 0
for i = m down to 1
    for j = n down to 1
        if x_i == y_i
            c[i,j] = c[i+1, j+1]
            b[i,j] = '↘'
        else if c[i+1, j] ≥ c[i, j+1]
            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) **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.**

Algorithm 4 PRINT-LCS(b,X,m,n)

```
result = vector
i = 0
j = 0
while i < m ∧ j < n
  if b[i,j] == "→"
      j = j + 1
  else if b[i,j] == "↓
      i = i + 1
  else
      result.push(X_i)
      i = i + 1
      j = j + 1
return result
```

3) Activity Scheduling with Revenue

(a) **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.**
From page 416 of the CRLS book, the usual cut paste argument shows that the optimal solution A_{ij} includes the two optimal solution for the two sub-problem: $S_{ik} \wedge S_{kj}$.
Then we need to find a set A'_{kj} of mutually compatible activities $\in S_{kj}$ where the collective revenue is maximized.
To begin with find the choices that make the solution optimal such that

$$S_{ij} = \{a_k \in S : f_i \leq s_k \leq f_k \leq s_j\}$$

Then the next step, assume that the optimal solution exists, which will be labeled as A_ij. After that define the subproblems of the following: $S_{ik} \wedge S_{kj}$. Then fide the optimal solution to the subproblem as

$$A_{jk} = A_{ij} \cap S_{ik}$$

$$A_{kj} = A_{ij} \cap S_{kj}$$

Hence, the optimal solution would be

$$A_{ij} = A_{ik} \cup a_k A_{kj}$$

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

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

(c) **Write pseudocode to print out the set of activities chosen.**

Algorithm 5 REVENUE-ACTIVITY-SELECTOR(n)

```
x = [ ]
A = [ ]
for i = 0; i < n + 1; i++
    x[i] = []
    activity[i] = [ ]
    for j = 0; j < n + 1; j ++
        x[i][j] = 0
        activity[i][j] = 0
```

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

Algorithm 6 Print-out-activites(activity)

```
for i to activity.length
    for j = 1 to activity[i].length
        print activity[i,j]
        j = j + 1
    i = i + 1
```

(e) **What is the asymptotic runtime of your solution including (c) and (d)?**
Since there are for loops that are dependent then the run time is $\Theta(n^2)$.