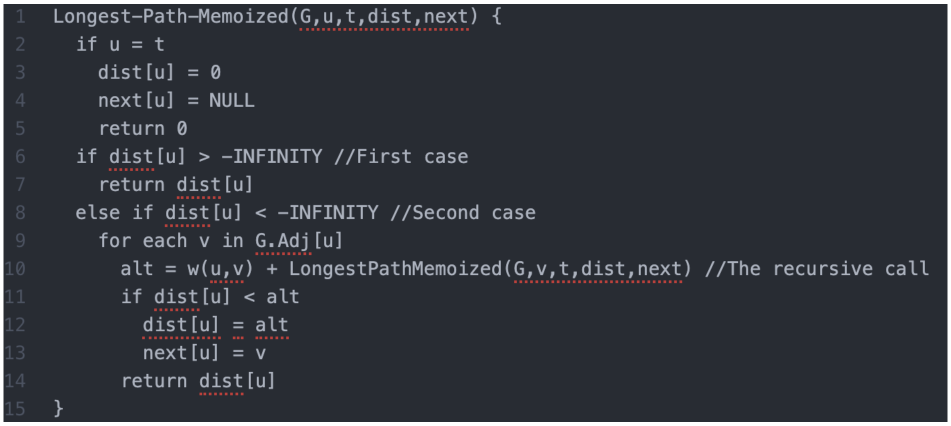
1. Constructing and Extracting the Solution for Longest Path(10 pts)

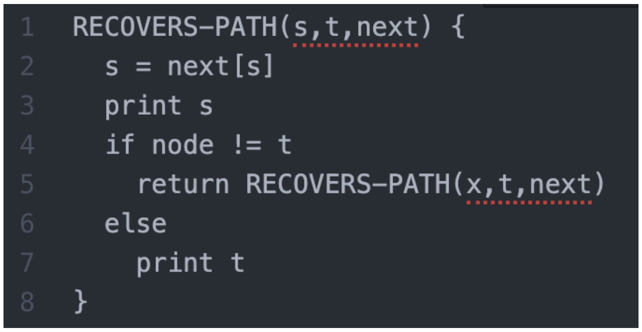
In the following you use the posted solution to LongestPathValueMemoized from class last week.

(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 the next[u]. Assume that all entries of next are initialized to 1 by the caller.



```
Longest-Path-Memoized(G,u,t,dist,next) {
  if u = t
    dist[u] = 0
    next[u] = NULL
    return 0
  if dist[u] > -INFINITY //First case
    return dist[u]
  else if dist[u] < -INFINITY //Second case
    for each v in G.Adj[u]
      alt = w(u,v) + LongestPathMemoized(G,v,t,dist,next) //The recursive call
      if dist[u] < alt
        dist[u] = alt
        next[u] = v
      return dist[u]
}
```

(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.



```
RECOVERS-PATH(s,t,next) {
  s = next[s]
  print s
  if node != t
    return RECOVERS-PATH(x,t,next)
  else
    print t
}
```

(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].

The asymptotic runtime of the total solution is $\theta(|E|)$. This is due to the traversal of the edges.

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

The asymptotic use of space in the solution is θ(|V| + |E|)

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 Topic 12 Lecture Notes Dynamic Programming
X_i = prefix $< x1, ..., xi >$
Y_i = prefix $< y1, ..., yi >$

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:

X_i = suffix $< xi, ..., xm >$
Y_j = suffix $< yi, ..., ym >$

(a) (3 pts) Reformulate **Theorum 15.1** for the suffix version by filling in the blanks below:

Let Z = $< z_1, ..., z_k >$ be any LCS of X = $< x_1, ..., x_m >$ and Y = $< y_1, ..., y_n >$. Then:

1. If $x_1 = y + 1$, this means that $x_1 = y_1 = z_1$. This means it 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 $x_1 \neq y_1$ this means that $z_1 \neq x_1$. This means that Z is an LCS of X_2 and Y.
4. If $x_1 \neq y_1$, this means that $z_1 \neq y_1$. This means that Z is an LCS of X and Y_2.
5. (If the first characters of X and Y don't match each other, the 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.)

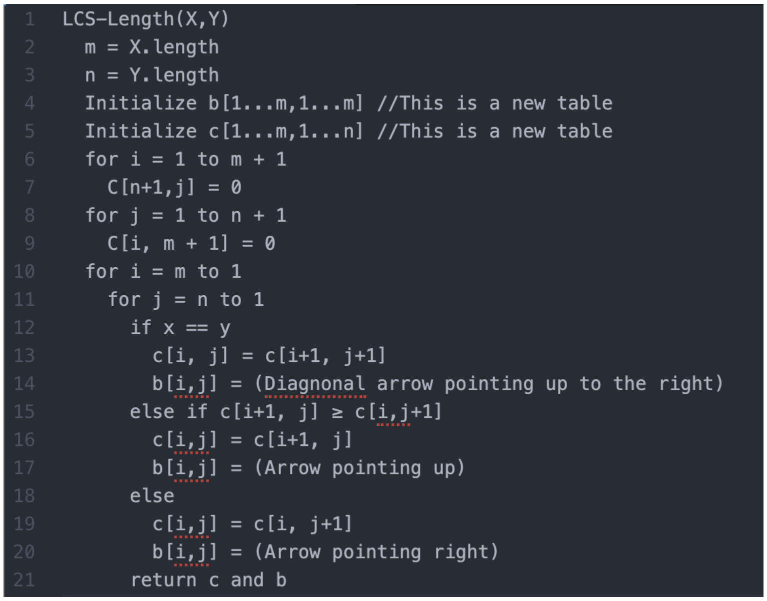
(To keep this problem set from getting too long we will skip the proof, but it is an easy translation of the proof on page 392 of CLRS.)

(b) (5 pts) Now redefine the **Recursive formulation** accordingly (again, fill in the blanks).

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

$$X = \begin{cases} 0, & \text{if } i > m || j > n \\ c[i+1]j+1, & \text{if } i \leqslant m, j \leqslant n \\ & x_i = y_j \\ max(c[i, j+1], c[i+1, j]), & \text{if } i \leqslant m, j \leqslant n \\ & x_i \neq y_i \end{cases} \tag{1}$$

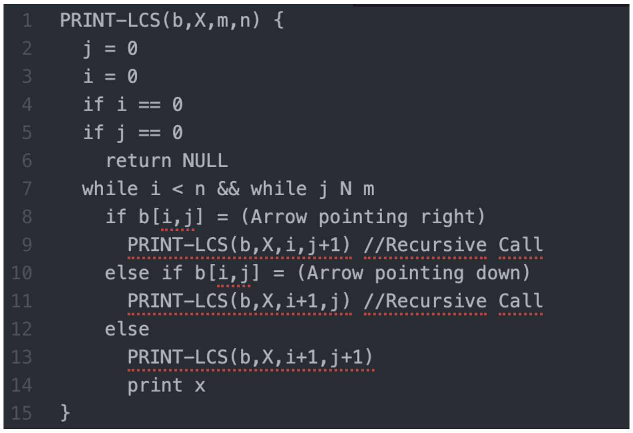
(c) (5 pts) Write pseudocode for LCS-LENGTH according to your **Recursive formulation**. (Hint: the arrows need to be different.)



```
LCS-Length(X,Y)
  m = X.length
  n = Y.length
  Initialize b[1...m,1...m] //This is a new table
  Initialize c[1...m,1...n] //This is a new table
  for i = 1 to m + 1
    C[n+1,j] = 0
  for j = 1 to n + 1
    C[i, m + 1] = 0
  for i = m to 1
    for j = n to 1
      if x == y
        c[i, j] = c[i+1, j+1]
        b[i,j] = (Diagnonal arrow pointing up to the right)
      else if c[i+1, j] ≥ c[i,j+1]
        c[i,j] = c[i+1, j]
        b[i,j] = (Arrow pointing up)
      else
        c[i,j] = c[i, j+1]
        b[i,j] = (Arrow pointing right)
      return c and b
```

*Got help from Matthew Kirts in Section 01.

(d) (2 pts) In the Notes, the longest subsequence could only be "printed" with PRINT-LCS and the pseudocode needed recursion. With your "suffix" method, you can do better and simplier: 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.



```
PRINT-LCS(b,X,m,n) {
  j = 0
  i = 0
  if i == 0
  if j == 0
    return NULL
  while i < n && while j N m
    if b[i,j] = (Arrow pointing right)
      PRINT-LCS(b,X,i,j+1) //Recursive Call
    else if b[i,j] = (Arrow pointing down)
      PRINT-LCS(b,X,i+1,j) //Recursive Call
    else
      PRINT-LCS(b,X,i+1,j+1)
      print x
}
```

*Got help from Matthew Kirts in Section 01.

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_1 has a 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.

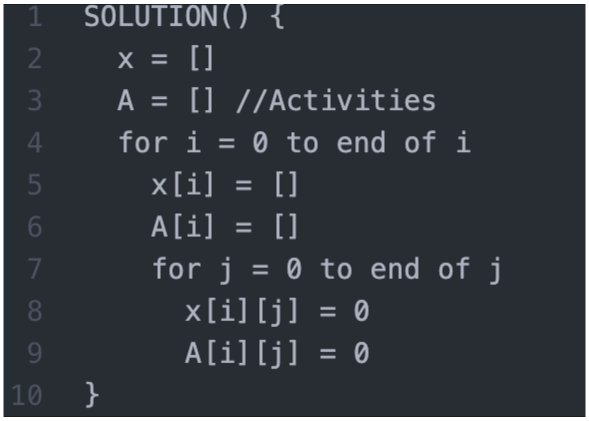
(a) (3 pts) Describe the structure of an optimal solution for A_{ij} for S_{ij}, as defined in CLRS and use a cut and paste argument to show that the problem has optimal substructure.

By referring to the cut and paste argument in CLRS, we need to show the optimal solution for A_{ij} has to be the optimal solution for S_{ik} and S_{kj}. In order to do this, we would find the set of compatible activities in S_{kj} where the revenue is maximized. In other words, the optimal solution A_{ij} would consist of the sub components A_{ik}, a_k, and A_{kj}.

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

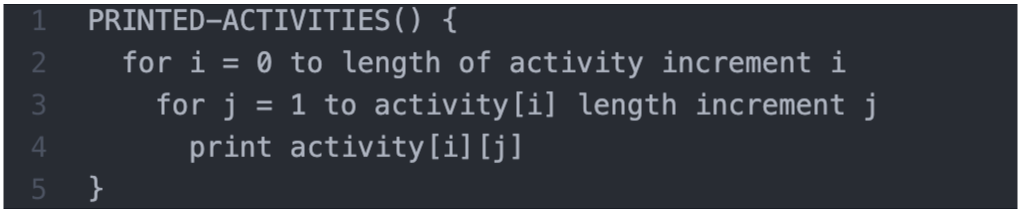
$$X = \begin{cases} 0, & \text{if } S_{ij} = 0 \\ max(val[i,k] + val[k,j] + r_k), & \text{if } S_{ij} \neq 0 \end{cases} \tag{2}$$

(c) (6 pts) Translate that definition into pseudocode that computes the optimal solution. Hints: Create fictitious activities where a_0 with $f_0 = 0$ and a_{n+1} with $s_{n+1=\infty}$. Define tables val[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 increasing difference of j-i.



```
SOLUTION() {
  x = []
  A = [] //Activities
  for i = 0 to end of i
    x[i] = []
    A[i] = []
    for j = 0 to end of j
      x[i][j] = 0
      A[i][j] = 0
}
```

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



```
PRINTED-ACTIVITIES() {
  for i = 0 to length of activity increment i
    for j = 1 to activity[i] length increment j
      print activity[i][j]
}
```

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

The asymptotic runtime of this solution is $O(n^2)$, due to the nested for loops and iteration through each one.