CS5800: Algorithms — Spring '21 — Virgil Pavlu

Homework 5
Submit via Gradescope

Name: Shri Datta Madhira(NUID: 001557772)
Collaborators:

Instructions:

- Make sure to put your name on the first page. If you are using the LaTeX template we provided, then you can make sure it appears by filling in the `yourname` command.
- Please review the grading policy outlined in the course information page.
- You must also write down with whom you worked on the assignment. If this changes from problem to problem, then you should write down this information separately with each problem.
- Problem numbers (like Exercise 3.1-1) are corresponding to CLRS 3^{rd} edition. While the 2^{nd} edition has similar problems with similar numbers, the actual exercises and their solutions are different, so make sure you are using the 3^{rd} edition.

1. ***(15 points)*** *Exercise 15.4-5.*

Solution:

Step 1: From the recursive code I wrote in the last assignment, I realized that this problem can be solved easily using dynamic programming by tabulating the results we got from the sub-sequences solved before. This problem exhibits dynamic programming because the optimal solution for the main problem depends on the optimal solutions for the sub-problems, i.e., we get the maximum desired length of sub-sequence only when we get the maximum length at each sub-problem. If there is a solution optimal than the solution we have for any sub-problem, then we can plug in that solution and get an even better solution which contradicts the argument.

Step 2: The recurrence objective is,

$$m[i] = \begin{cases} max\{m[i], m[j]+1\} & \text{if array[i]>array[j] when i = 1 to n and j = 0 to i} \end{cases}$$

Step 3:

Algorithm 1: Algorithm to find the longest increasing sub-sequence

```
Function LONGEST-SUB-SEQUENCE(index):
    m = [1]*n
    For i = index+1 to n
        For j = 0 to i
            If array[i] > array[j] And m[i] < m[j] + 1 :
                m[i] = m[j] + 1
    Return max(m)
```

Step 4: Tracing the solution

Algorithm 2: Algorithm to trace the solution for longest increasing sub-sequence

```
Function TRACE-SOLUTION-LIS(m):
    lcs = [0]*max(m)
    k, maxIndex = 0, m.index(max(m))
    For i = 0 to maxIndex+1
        For j = i+1 to maxIndex+1
            If m[i]<m[j] and m[j]-m[i] == 1 :
                If k == 0 :
                    lcs[k].append(nums[i])
                    k += 1
                    break
                If ls[k-1] < nums[i] :
                    lcs[k] = nums[i]
                    k += 1
                    break
    Return lcs
```

Time Complexity: It has 2 for loops to iterate through the sequence. So it takes $O(n^2)$ time.
Space Complexity: It takes $O(n)$ for each of the 'm' and 'lcs' arrays we used.

2. ***(15 points)*** *Exercise 15.2-1.*

Solution:
The given array is $< 5, 10, 3, 12, 5, 50, 6 >$.

Step 1: This problem exhibits optimal substructure because we can split the matrices at some point 'k' as $(A_i, \cdots, A_k)(A_{k+1}, \cdots, Aj)$. We now have two sub-problems. The optimal solution to these sub-problems will become the optimal solution to the main problem. Therefore, the recurrence objective can be written as shown in step 2. We can say that this is the optimal solution because, if there is another solution to a sub-problem that is optimal than the one we have, we can plug in that solution and get a better overall solution, which contradicts that that solution is optimal than the one we have.

Step 2: The recurrence objective is,

$$m[i,j] = \begin{cases} 0 & \text{if i = j,} \\ min\{m[i,k] + m[k+1,j] + p_{i-1}p_k p_j\} & \text{if i < j and i} \leq \text{k < j.} \end{cases}$$

Step 3:

$$m[i,j] = \begin{bmatrix} 0 & 150 & 330 & 405 & 1655 & 2010 \\ x & 0 & 360 & 330 & 2430 & 1950 \\ x & x & 0 & 180 & 930 & 1770 \\ x & x & x & 0 & 3000 & 1860 \\ x & x & x & x & 0 & 1500 \\ x & x & x & x & x & 0 \end{bmatrix}$$

$$s[i,j] = \begin{bmatrix} 0 & 1 & 2 & 2 & 4 & 2 \\ x & 0 & 2 & 2 & 2 & 2 \\ x & x & 0 & 3 & 4 & 4 \\ x & x & x & 0 & 4 & 4 \\ x & x & x & x & 0 & 5 \\ x & x & x & x & x & 0 \end{bmatrix}$$

Example calculation of m[1,6];

$$m[1,6] = min \begin{cases} m[1,1] + m[2,6] + p_0 p_1 p_6 = 0 + 1950 + 300 = 2250 \\ m[1,2] + m[3,6] + p_0 p_2 p_6 = 150 + 1770 + 90 = 2010 \\ m[1,3] + m[4,6] + p_0 p_3 p_6 = 330 + 1800 + 360 = 2490 \\ m[1,4] + m[5,6] + p_0 p_4 p_6 = 405 + 1500 + 150 = 2055 \\ m[1,5] + m[6,6] + p_0 p_5 p_6 = 1655 + 0 + 1500 = 3155 \end{cases} \Rightarrow m[1,6] = 2010,\ s[1,6] = 2$$

Step 4: Tracing the solution
From this we can see that the optimal solution is, $\boldsymbol{(A_1\ A_2)\ (((A_3\ A_4)\ A_5)\ A_6)}$.

3. ***(15 points)*** *Exercise 15.3-2.*

Solution:

Let the array A be $< 0, 59, 86, 88, 1, 33, 90, 19, 60, 3, 73, 69, 63, 31, 48, 58 >$ of length 16.

```
Step - 1:
<0,59,86,88,1,33,90,19>, <60,3,73,69,63,31,48,58>
Step - 2:
<0,59,86,88>, <1,33,90,19>, <60,3,73,69>, <63,31,48,58>
Step - 3:
<0,59,86>, <88>, <1,33,90>, <19>, <60,3,73>, <69>, <63,31,48>, <58>
Step - 4:
<0,59>,<86>,<88>,<1,33>,<90>,<19>,<60,3>,<73>,<69>,<63,31>,<48>,<58>
Step - 5:
<0>,<59>,<86>,<88>,<1>,<33>,<90>,<19>,<60>,<3>,<73>,<69>,<63>,<31>,<48>,<58>
Step - 6:
<0,59>,<86>,<88>,<1>,<33>,<90>,<19>,<60>,<3>,<73>,<69>,<63>,<31>,<48>,<58>
Step - 7:
<0,59,86>,<88>,<1>,<33>,<90>,<19>,<60>,<3>,<73>,<69>,<63>,<31>,<48>,<58>
Step - 8:
<0,59,86,88>,<1>,<33>,<90>,<19>,<60>,<3>,<73>,<69>,<63>,<31>,<48>,<58>
Step - 9:
<0,59,86,88>,<1,33>,<90>,<19>,<60>,<3>,<73>,<69>,<63>,<31>,<48>,<58>
Step - 10:
<0,59,86,88>,<1,33,90>,<19>,<60>,<3>,<73>,<69>,<63>,<31>,<48>,<58>
Step - 11:
<0,59,86,88>,<1,19,33,90>,<60>,<3>,<73>,<69>,<63>,<31>,<48>,<58>
Step - 12:
<0,59,86,88>,<1,19,33,90>,<3,60>,<73>,<69>,<63>,<31>,<48>,<58>
Step - 13:
<0,59,86,88>,<1,19,33,90>,<3,60,73>,<69>,<63>,<31>,<48>,<58>
Step - 14:
<0,59,86,88>,<1,19,33,90>,<3,60,69,73>,<63>,<31>,<48>,<58>
Step - 15:
<0,59,86,88>,<1,19,33,90>,<3,60,69,73>,<31,63>,<48>,<58>
Step - 16:
<0,59,86,88>,<1,19,33,90>,<3,60,69,73>,<31,48,63>,<58>
Step - 17:
<0,59,86,88>,<1,19,33,90>,<3,60,69,73>,<31,48,58,63>
Step - 18:
<0,1,19,33,59,86,88,90>,<3,31,48,58,60,63,69,73>
Step - 19:
<0,1,3,19,31,33,48,58,59,60,63,69,73,86,88,90>
```

As we can see from the implementation of Merge Sort, at every step the sub-problems are different (**no overlapping sub-problems**). Therefore, even if we do memoization on Merge Sort, it will not improve the running time of the algorithm.

4. ***(20 points)*** *Exercise 15.3-5.*

Solution:
Say we have a rod of length 'n'. Our optimal substructure will yield the lengths at which the rod has to be cut to get the maximum value. And let's assume that we get a solution that contains 4 'i' length pieces.

Now if we put a limit $l_i = 2$ on number of pieces of a particular length 'i' are allowed, the optimal solution to the problem described above will have to look for other ways to cut the rod in order to obey the rule of $number\ of\ i-length\ pieces < l_i$, thereby making it not optimal.

Therefore, the optimal substructure presented in section 15.1 for the rod-cutting problem will not work if a limit is enforced on number of 'i' length pieces.

5. ***(20 points)*** *Problem 15-5.*

Solution:

(a) **Step 1:**
This problem exhibits the optimal substructure required to implement dynamic programming. The cost of an operation at position 'i' can be taken as the optimal cost of all the operations for all the sub-problems for 'i-1' positions before that and the cost of the current operation.
Step 2: The recurrence objective is,

$$m[i,j] = \begin{cases} m[i-1,j-1] + cost(\textbf{\textit{Copy}}), & \text{x[i]==y[j] then z[j]=x[i]} \\ m[i-1,j-1] + cost(\textbf{\textit{Replace}}), & \text{x[i]} \neq \text{y[j]} \\ m[i-1,j] + cost(\textbf{\textit{Delete}}), & \text{nothing given in question} \\ m[i,j-1] + cost(\textbf{\textit{Insert}}), & \text{z[k]=c} \\ m[i-2,j-2] + cost(\textbf{\textit{Twiddle}}), & \text{z[j]=x[i+1],z[j+1]=x[i]} \\ m[i-1,j-1] + cost(\textbf{\textit{Kill}}), & \text{there are not examined(x) and o/p has been achieved.} \end{cases}$$

Step 3:
m[1 $\cdots$ m, 1 $\cdots$ n]: sub-problem dependency table. Each index can be any of the below operations. s[1 $\cdots$ m, 1 $\cdots$ n]: table to store the action done at each step.

```
Algorithm for edit-distance for optimal operation sequence
def edit-distance(x,y,0,0):
    for i in range(len(x),0,-1):
        for j in range(len(y),0,-1):
            if #condition:
                m[i,j] = cost(condition)
                s[i,j] = copy/insert/replace/delete/twiddle/kill
                return z[i] + edit-distance(x,y,i-1,j-1)
```

Time Complexity: It takes $O(n^3)$ time.
Space Complexity: $\Theta(n^2)$ to store 'm' and 's'.

(b) If the problem of edit distance is cast as optimal alignment, we would have spaces in 'x' and 'y' in edit distance also. So, basing on the scores given by optimal alignment we will give scores to edit-distance transformation operations.

cost(**COPY**): Since we copy an element if x[i] == y[i], which satisfies 1^{st} condition, we give cost as 1. **cost(COPY) = 1**.

cost(**INSERT**): If x has space and we want a letter that is in y, we will do the insert operation. Then cost of the insert operation will be 2 (3rd condition). **cost(INSERT) = 2**.

cost(**DELETE**): The same argument as insert operation goes for delete operation. If x does not have a space and y does not, then we have to delete the element in x. So, basing on the 3^{rd} condition, we get 2. **cost(DELETE) = 2**.

cost(**REPLACE**): The opposite argument as COPY goes for REPLACE. We replace an element if x[i] $\neq$ y[i], which satisfies 2^{nd} condition. We give the score as 1. **cost(REPLACE) = 1**.

cost(**TWIDDLE**): Twiddle is a copy operation but with two elements. Swapped or not does not matter for cost calculation. So, cost(TWIDDLE) = 2*cost(COPY) = 2. **cost(TWIDDLE) = 2**.

6. *(30 points)* *(Note: you should decide to use Greedy or DP on this problem) Prof. Curly is planning a cross-country road-trip from Boston to Seattle on Interstate 90, and he needs to rent a car. His first inclination was to call up the various car rental agencies to find the best price for renting a vehicle from Boston to Seattle, but he has learned,much to his dismay, that this may not be an optimal strategy. Due to the plethora of car rental agencies and the various price wars among them, it might actually be cheaper to rent one car from Boston to Cleveland with Hertz, followed by a second car from Cleveland to Chicago with Avis, and so on, than to rent any single car from Boston to Seattle.Prof. Curly is not opposed to stopping in a major city along Interstate 90 to change rental cars; however, he does not wish to backtrack, due to time constraints. (In other words, a trip from Boston to Chicago, Chicago to Cleveland, and Cleveland to Seattle is out of the question.) Prof. Curly has selected n major cities along Interstate 90 and ordered them from East to West, where City 1 is Boston and City n is Seattle. He has constructed a table T[i, j] which for all i < j contains the cost of the cheapest single rental car from City i to City j. Prof. Curly wants to travel as cheaply as possible.Devise an algorithm which solves this problem, argue that your algorithm is correct,and analyze its running time and space requirements. Your algorithm or algorithms should output both the total cost of the trip and the various cities at which rental cars must be dropped off and/or picked up.*

Solution:

Step 1: We want the cheapest services from Boston to Seattle. This problem exhibits an optimal substructure. Say, the cheapest way is Boston -> Cleveland -> Seattle. Then, the sub-paths Boston -> Cleveland and Cleveland -> Seattle must also be optimally cheap. If there exists a better sub-path then this path can be substituted in the optimal path leading to a better path. But this negates the optimality of the solution.

Step 2: Based on the optimal sub-structure, we can write the recurrence objective as,

$$m[i,j] = \begin{cases} 0 & \text{if i = j,} \\ min\{m[i,k] + T[k,j]\} & \text{if i < j and (i,j) } \in \text{ cities.} \end{cases}$$

Step 3: Computing the table

Algorithm 3: Algorithm to find the cheapest services from Boston to Seattle.

```
Function CURLY-CHEAPEST-SERVICES(i):
    If i ≤ N-1 :
        Return 0
    For j = N-1 to i
        For k = i+1 to j+1
            d = CURLY-CHEAPEST-SERVICES(k) + T[i, k]
            If d < dist[i,j] :
                m[i,j] = d,  s[i,j] = k
    Return m[i,j]
```

Step 4: Tracing the solution

Algorithm 4: Tracing the path for professor Curly

```
Function GET-COST-PATH(i,j,cost):
    path.append(s[i,j])
    cost += m[i,j]
    If s[i,j] < N-1 :
        GET-COST-PATH(s[i,j], j, cost)
    Return cost
```

Step 5:

Argument We make use of dynamic programming for this question. The optimal substructure can be obtained as,

$$m[i,j] = \begin{cases} 0 & \text{if i = j,} \\ min\{T[i,k] + m(k,j)\} & \text{if i < j and k = i+1 to j.} \end{cases}$$

The same argument that is explained in Step-1 can be used here to prove that the solution provided is optimal.

Time Complexity The loops are nested two deep and each take at most n-1 values. This yields a running time of $O(n^2)$.

Space Complexity It takes $\Theta(n^2)$ space for storing 'dist' and 's' arrays.