CS5800: Algorithms — Spring '21 — Virgil Pavlu

Homework 5
Submit via Gradescope

Name: Ketaki Nitin Kolhatkar
Collaborators: Madhusudan Malhar Deshpande

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 15.2-1.*

Solution:
$< 5, 10, 3, 12, 5, 50, 6 >$
Dimensions : D0 = 5, D1 = 10, D2 = 3, D3 = 12, D4 = 5, D5 = 50, D6 = 6
The matrices would be as following:
A = 5 * 10,
B = 10 * 3,
C = 3 * 12,
D = 12 * 5,
E = 5 * 1,
F = 50 * 6
Using the a

m[1,2] = m[1,1] + m[2,2] + (D0 * D1 * D2)
m[1,2] = 0 + 0 + (5*10*3)
m[1,2] = 150

m[3,4] = m[3,3] + m[4,4] + (D2 * D3 * D4)
m[3,4] = 0 + 0 + (3 * 12 * 5)
m[3,4] = 180

m[4,5] = m[4,4] + m[5,5] + (D3 * D4 * D5)
m[4,5] = 0 + 0 + (12 * 5 * 50)
m[4,5] = 3000

m[5,6] = m[5,5] + m[6,6] + (D4 * D5 * D6)
m[5,6] = 0 + 0 + (5 * 50 * 6)
m[5,6] = 1500

m[1,3] = min[(m[1,1] + m[2,3] + (D0 *D1 * D3)), (m[1,2] + m[3,3] + (D0 * D2 * D3))]
m[1,3] = min[0 + (10 *3*12) + (5*10*12), 150 + 0 +(5*3*12)]
m[1,3] = min[960, 330]
m[1,3] = 330

m[2,4] = min[(m[2,2] + m[3,4] + (D1 *D2*D4)), (m[1,2] + m[3,3] + (D1 * D3 * D4)]
m[2, 4] = min[330, 960]
m[2, 4] = 330

m[3, 5] = min[(m[3, 3] + m[4, 5] + (D2*D3*D5)), (m[3,4] + m[5,5] + (D2 * D4 * D5)]
m[2, 4] = min[4800, 930]
m[2, 4] = 930

m[4,6] = min[(m[4,4] + m[5,6] + (D3*D4*D6)), (m[4,5] + m[6,6] + (D3 * D5 * D6)]
m[2, 4] = min[1860, 6600]

m[2, 4] = 1860

m[1,4] = min[(m[1,1] + m[2,4] + (D0*D1*D4)), (m[1,2] + m[3,4] + (D0*D2*D4), m[1,3]+m[4,4]+ (D0*D3*D4)]
m[2, 4] = min[580, 405, 630]
m[2, 4] = 405

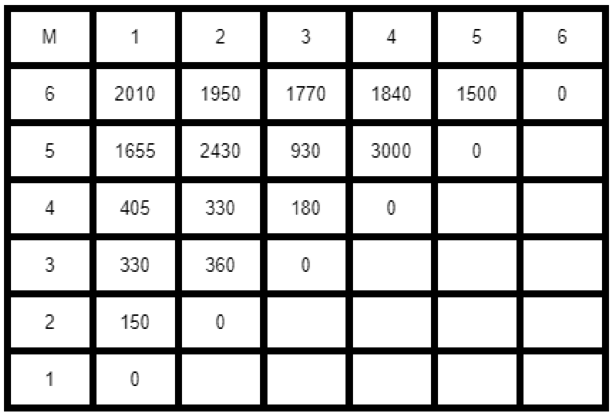
m[2,5] = min[(m[2,2] + m[3,5] + (D1*D2*D5)), (m[2,3] + m[4,5] + (D1*D3*D5), m[2,4]+m[5,5]+ (D1*D4*D5)]
m[1, 4] = min[2430, 9360, 2830]
m[1, 4] = 2430

m[3,6] = min[(m[3,3] + m[4,6] + (D2*D3*D6)), (m[3,4] + m[5,6] + (D2*D4*D6), m[3,5]+m[6,6]+ (D2*D5*D6)]
m[1, 4] = min[2076, 1770, 1830]
m[1, 4] = 1770

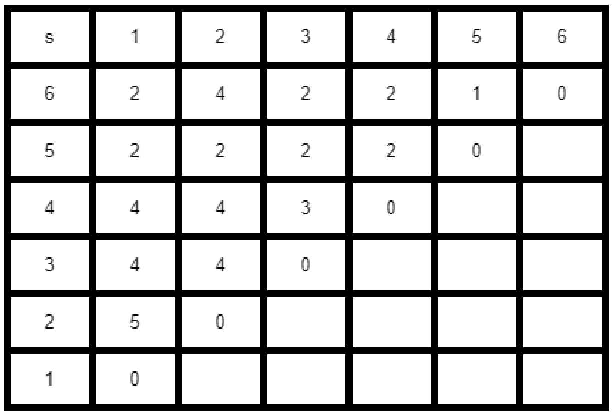
m[1,5] = min[m[1,1]+m[2,5]+ (D0*D1*D5),m[1,2]+m[3,5]+ (D0*D2*D5), m[1,3]+m[1,4]+ (D0*D3*D5),m[1,4]+m[1,5]+ (D0*D4*D5)]
m[1, 4] = min[4930, 1830, 6330, 1655]
m[1, 4] = 1665

m[1,6] = min[m[1,1]+m[2,6]+ (D0*D1*D6), m[1,2]+m[3,6]+ (D0*D2*D6), m[1,3]+m[4,6]+ (D0*D3*D6), m[1,4]+m[5,6]+ (D0*D4*D6), m[1,5]+m[6,6]+ (D0*D5*D6)] m[1,6] = min[2250, 2010, 2250, 2055, 3155]
m[1,6] = 2010

m[2,6] = min[m[2,2]+m[3,6]+ (D1*D2*D6), m[2,3]+m[4,6]+ (D1*D3*D6), m[2,4]+m[5,6]+ (D1*D4*D6), m[2,5]+m[6,6]+ (D1*D5*D6)]
m[2,6] = min[1950, 2940, 2130, 5430]
m[2,6] = 1950



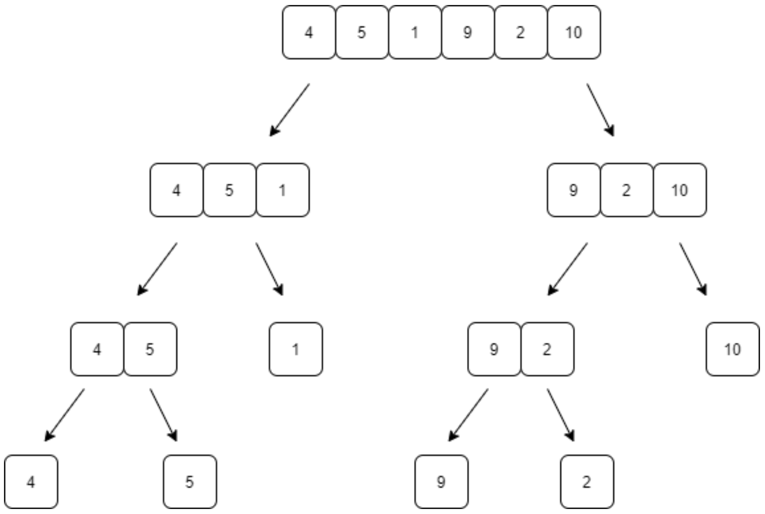
M	1	2	3	4	5	6
6	2010	1950	1770	1840	1500	0
5	1655	2430	930	3000	0	
4	405	330	180	0		
3	330	360	0			
2	150	0				
1	0					



s	1	2	3	4	5	6
6	2	4	2	2	1	0
5	2	2	2	2	0	
4	4	4	3	0		
3	4	4	0			
2	5	0				
1	0					

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

Solution:
Merge sort works on a recursive function, but the sub problems in merge sort are not the same and there is no overlap in them. This is because every time we call the recursive function we have different values to merge recursively. Hence there is no benefit of memoization - storing the sub problem solutions.
As we can see in the example below we call the function recursively for merging different numbers with different solutions.



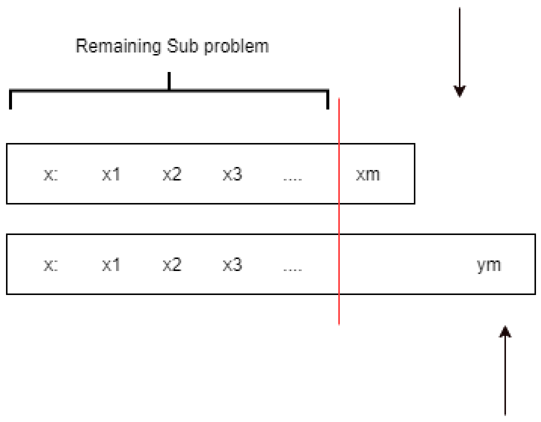
3. ***(20 points)*** *Exercise 15.3-5.*

Solution:
The number of pieces we cut on one side would decide the number of pieces we would cut on the other. That is the sub problems are not independent to solve, the next sub problem solution depends on the solution to the previous sub problem. Thus the optimal substructure property is not held true in this case.
Example : Consider a rod of length 10 cm.

p is the price and L is the max allowed counts of a particular length. Let p1 = 1, p2 = 3, p3 = 4, p4 = 7. Let L1 = 10, L2 = 2, L3 = 6, L4 = 1. The optimal solution would have been X = (4,4,2). However, this is not allowed as there can be only 1'4 cm' piece.

4. ***(20 points)*** *Problem 15-5.*

Solution:
(a) The problem is solved using bottom up approach that is comparing the strings from right to left.
Step 1 : Characterising Optimal Solution

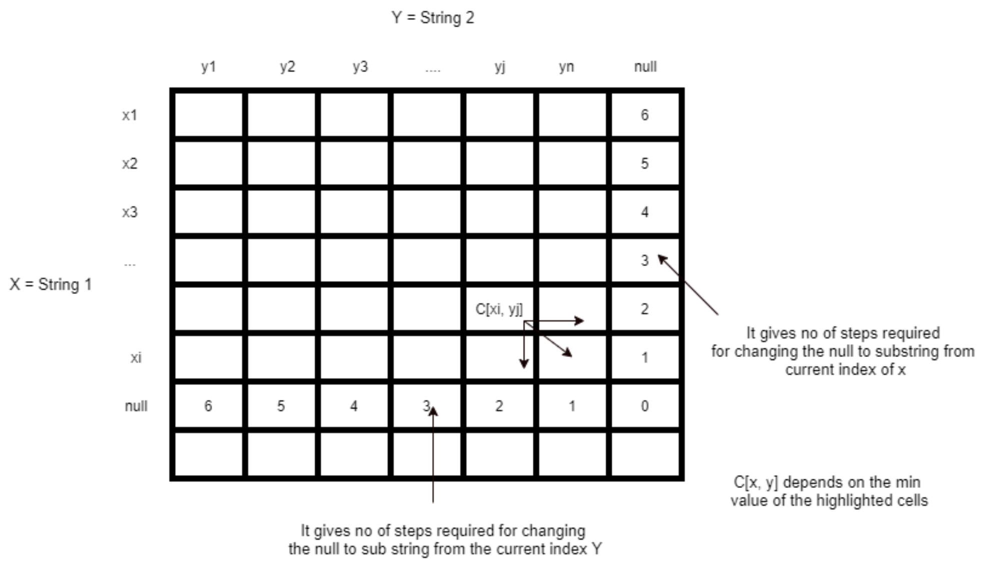


Step 2A. : Objective Function - choose the operation to perform

C[n] = 1 + min(C[i+1], C[1, j+1], C[i+1, j+1])

S[n] = Index of min(C[i+1,j], C[i, j+1], C[i+1, j+1])

Step 2B. : Sub problem Dependencies:


Y = String 2
y1
y2
y3
....
yj
yn
null
X = String 1
x1
x2
x3
...
xi
null
6
5
4
3
2
1
0
6
5
4
3
2
1
C[xi, yj]
It gives no of steps required for changing the null to substring from current index of x
C[x, y] depends on the min value of the highlighted cells
It gives no of steps required for changing the null to sub string from the current index Y

Step 3 : Solving the Dynamic Programming using bottom up method

Order of operation : Right to Left and Bottom to up

```
Let the costs of various operations be as follows:
Copy = -1
Delete = 2
Insert = 2
Replace = 1
Twiddle = Infinity
Kill = Infinity

Also,
Copy is the same as incrementing both i and j by 1
Replace is the same as incrementing both i and j by 1
Insert is the same as incrementing only j by 1
Delete is the same as incrementing only i by 1

X = String 1
Y = String 2

EditDistance(X, Y)
    m = X.length
    n = Y.length
    Let C[] be a 2D array of dimensions m+1 and n+1
    Let s[] be an 1D array which has operations to perform at each one
    C[m+1, n+1] = 0
    for j from 1 to n:
        C[m+1, j] = n-1 + 1
    for i from 1 to m:
        C[i, n+1] = m-1 + 1
    for i from m to 1 :
        for j from n to 1:
        if X[i] == Y[j]:
            C[i,j] == C[i+1, j+1]
        else:
            C[i, j] = 1 + min(C[i+1, j], C[i, j+1], C[i+1, j+1])
    if C[i+1, j] < C[i, j+1] and C[i+1, j+1]:
        S[i] = "Replace"
return C[], S[]
```

The time complexity of this algorithm would be $O(n^2)$. Results need to be stored of the order $m * n$, which gives us a space complexity of $O(n^2)$.

(b) The problem could be solved in a similar fashion as the previous problem. If we move the diagonal the cost of the cell increases by 1 if 2 strings match. Or we increase the cost of the cell by -1 if the two strings don't match. If we move horizontally, we add a in string 1 while moving vertically adds a in string 2. This reduces the value of the cell by 2. We want to move along the path which will maximize the overall value which indicates maximum overlap between two strings.

```
Let the costs of various operations be as follows:
Copy = -1
Delete = 2
Insert = 2
Replace = 1
Twiddle = Infinity
Kill = Infinity

Also,
Copy is the same as incrementing both i and j by 1
Replace is the same as incrementing both i and j by 1
Insert is the same as incrementing only j by 1
Delete is the same as incrementing only i by 1

X = String 1
Y = String 2

EditDistance(X, Y)
    m = X.length
    n = Y.length
    Let C[] be a 2D array of dimensions m+1 and n+1
    Let s[] be an 1D array which has operations to perform at each one
    C[m+1, n+1] = 0
    for j from 1 to n:
        C[m+1, j] = n-1 + 1
    for i from 1 to m:
        C[i, n+1] = m-1 + 1
    for i from m to 1 :
        for j from n to 1:
        if X[i] == Y[j]:
            C[i,j] == C[i+1, j+1]
        else:
            C[i, j] = 1 + min(C[i+1, j], C[i, j+1], C[i+1, j+1])
    if C[i+1, j] < C[i, j+1] and C[i+1, j+1]:
        S[i] = "Replace"
return C[], S[]
```

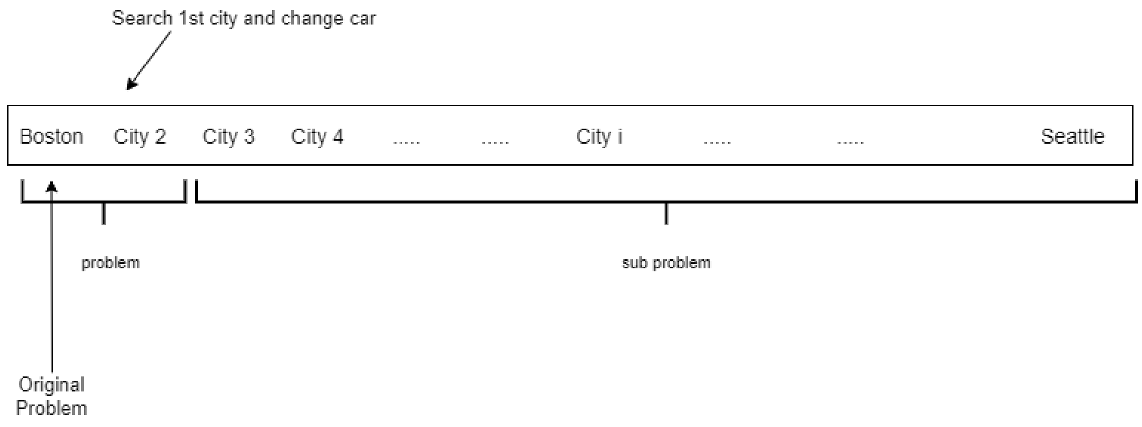
The time complexity of this algorithm would be $O(n^2)$. Results need to be stored of the order $m * n$, which gives us a space complexity of $O(n^2)$.

5. ***(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 contraints. (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:
Greedy method will not work for this problem since choosing the cheapest option from car might not give us the most optimal solution, which means that choosing the next destination to change the car depends on the solution to the remaining sub problem.
Example: Consider 4 cities
T(1, 2) = 5
T(2, 3) = 5
T(3, 4) = 6
T(1, 3) = 7
T(2, 4) = 25
T(1, 4) = 45
Choosing the cheapest car from each city we get a cost of = 5+5+6 = 16. But the optimal solution would be city 1 to 3 and then city 3 to 4 = 7 +6 = 13. Thus we need to analyse the sub problems before making a decision. This is why we choose Dynamic Programming for this problem.

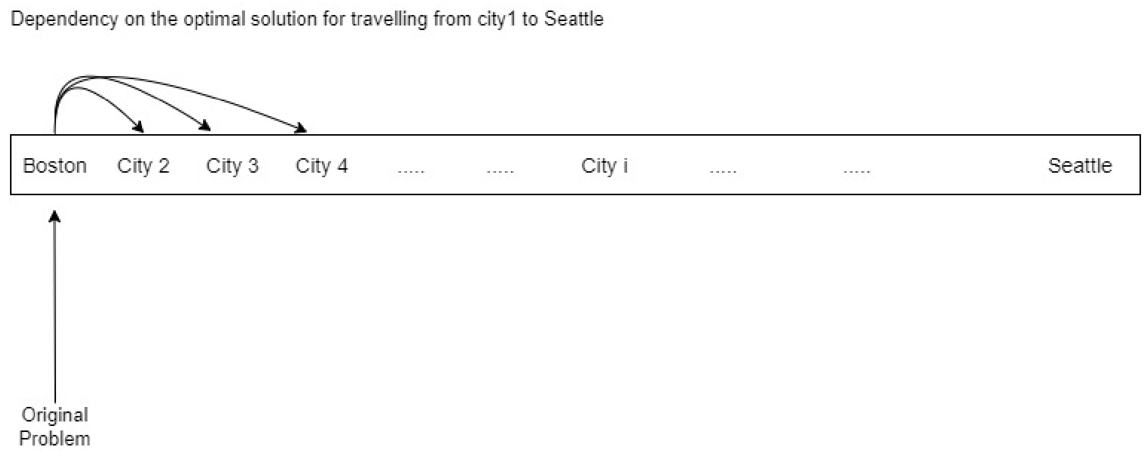
Step 1 : Characterizing the optimal solution



Step 2 A. : Objective Recurrence : Search for the next city to stop and change the car.

C[n] = min(cost of Boston to cityi + c[i+1]) $2 < i < n$

Step 2 B. : Subproblem dependencies



Step 3 : Solve all sub problems by the bottom up approach
Order from right to left - As the decision to travel upto any city i from Boston before changing car depends on cost to travel from city i+1 to Seattle.

```
cheapest_path(a):
    C = [0] * n
    S = [0] * n
    for i from n to 1:
        q = infinity
        if i == n:
            C[i] = 0
            S[i] = i
        else:
             for j from i+1 to n
             if q > T(i,j) + C(j+1, n):
                 q = T(i, j) + c(j+1, n)
                 C[i] = q
                 S[i] = j
return C, S
```

Time complexity would be $O(n^2)$ since there is a for loop inside another one.

Step 4 : Trace Solution

```
Best_Travel
path = []
for k from 1 to n:
    path.append(S[k])
    k = S[k]
return path
```

6. ***(15 points)*** *Exercise 15.4-5.*

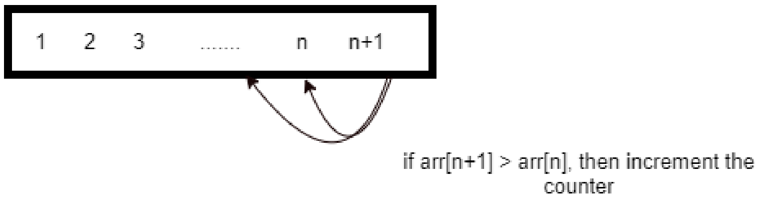
Solution:

Step 1 : Characterize the structure of an optimal solution

We want the sequence of the longest increasing numbers from a list of numbers. This does not necessarily have to be consecutive numbers. In a list [1, 3, 2, 6, 7, 5, 8, 9] the longest increasing sub sequence is of the length 6. This is the sequence [1, 2, 6, 7, 8, 9].

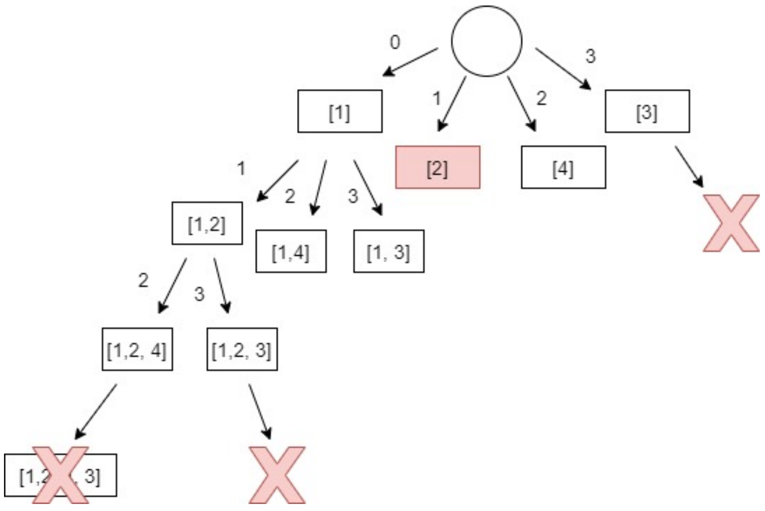
Step 2 : Recursively define the value of the optimal solution

We compute the solution by checking the array from the back. We start checking from the last element and check if the sequence is in a decreasing order. Once we get a number less than the last element we checked, we keep adding it to the counter. If a number is less than the number last checked then we do not increment the counter. This way we get a list of increasing numbers from the forward side. Counter gets incremented recursively if the $n[i] < n[i+1]$.



Step 3 : Digression - Why the Recursive approach is bad

One sub problem exists multiple times and hence we have to calculate it those many number of times. This gives us a time complexity of $O(2^n)$. This is solved by calculating the sub problem just once.



Step 4 : Compute the value the optimal solution Bottom-up

As discussed in the 2nd step we start by comparing two numbers from the end of the list. As we get $n[i] > n[i+1]$, we increase the counter. Hence we get an optimal solution bottom up.

```
array function(arr):
    Calculate the length of the array
    initialize lis[i] as [1] * n
    for i from 1, n:
        for j in range 0, i:
            if arr[i] > arr[j] and lis[i] < lis[j]+1:
                lis[i] = lis[j]+1

    for i from 1, n:
        maximum = max(maximum, lis[i])
return maximum
```

Time complexity of the code would be $O(n^2)$ since we have 2 for loops one under the other.

Step 5: Trace Solution

```
Building Longest Sub sequence
subseq = []
j = C.index(max(C))
for k from 0 to max(C):
    subseq.append(A[j])
    j = a[j]
return subseq
```