# DAA HW 3

## Problem 1

Apply various algorithm design strategies, practice formulating and analyzing algorithms, and implement an algorithm. In the US, coins are minted with 50, 25, 10, 5, and 1-cent denominations. Making change using the smallest possible number of coins repeatedly returns the biggest coin smaller than the amount to be changed until zero is reached. For example, 17 cents will result in the series 10 cents, 5 cents, 1 cent, and 1 cent.

1. Give a recursive greedy algorithm that generates a similar series of coins for changing n cents. Don't use dynamic programming for this problem. You don't have to prove that the algorithm returns the minimum number of coins.

Ans:

Assume - Output is required in the form of a series of coins used for changing n cents and the number of coins required..
Number of coins required will be measured using the length of the list "change"

1) **Pseudocode -**

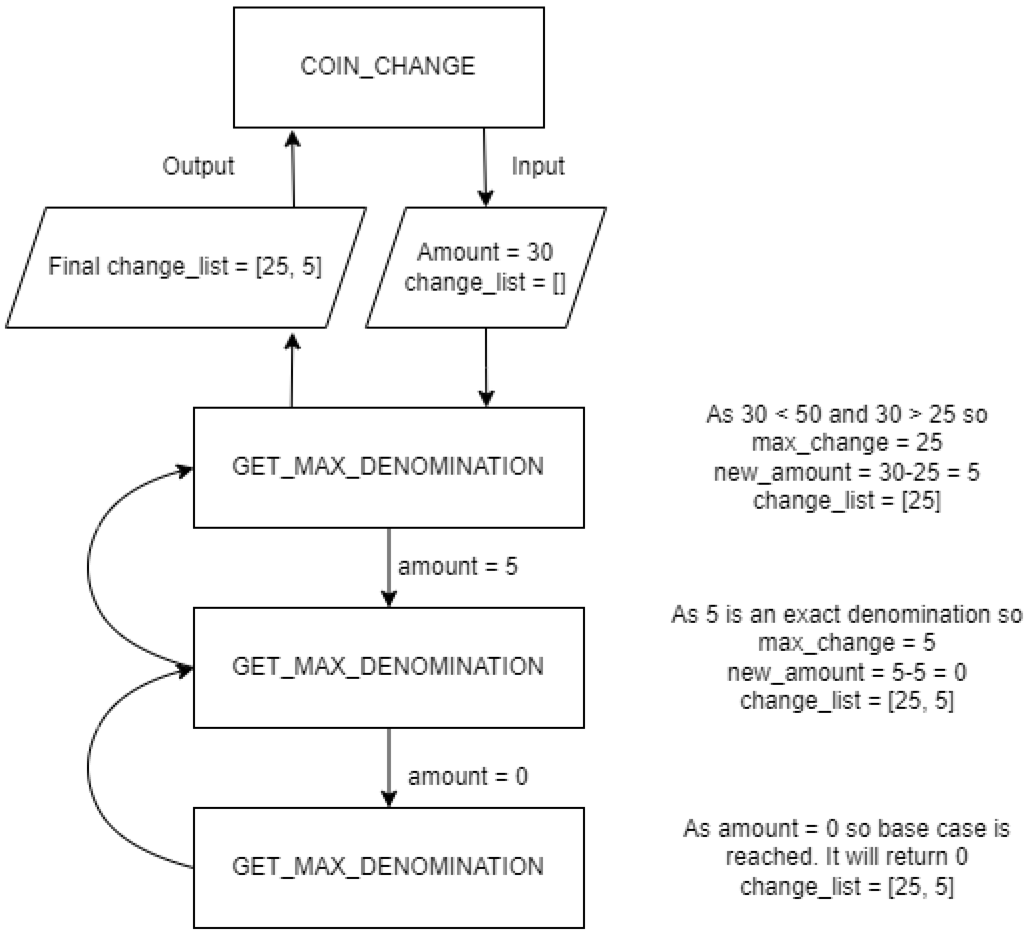
```
COIN_CHANGE(amount):
    if amount = 0:
        return 0
    change = []
    GET_MAX_DENOMINATION(amount, change)
    return change, len(change)

GET_MAX_DENOMINATION(amount, change):
    if amount = 0:
        return 0
    max_change = 0
    if amount >= 50:
        max_change = 50
    else if amount >= 25:
        max_change = 25
    else if amount >= 10:
        max_change = 10
    else if amount >= 5:
        max_change = 5
    else:
        max_change = 1
```

```

change.append(max_change)
amount = amount - max_change
GET_MAX_DENOMINATION(amount, change)
```

2) **Textual Description** -
    a) In this algorithm, the main function is the COIN_CHANGE function which takes an amount as an input and returns the list of coin change and number of coins required to change n cents.
    b) The COIN_CHANGE function calls the GET_MAX_DENOMINATION internally to calculate the sequence of the coin change recursively.
    c) Coin change list is passed by reference to GET_MAX_DENOMINATION function so that values can be written to it and returned by COIN_CHANGE function
    d) If-else statements are used to select the minimum number of coins and max_change is selected based on the amount. This max_change is then appended to the change list in every recursive call.
    e) After selecting max_change, a recursive call to GET_MAX_DENOMINATION function is placed by deducting max_change from the amount and the new amount with change list is given as an input to the next recursive call of GET_MAX_DENOMINATION.
    f) The base is reached when the recursive call is made with the amount as 0. In this case no change is made to the change list.
    g) Since the recursive call was the last statement in the GET_MAX_DENOMINATION method, all calls are subsequently removed from the stack as all the steps in GET_MAX_DENOMINATION method are executed.
    h) Finally, control reaches the last statement in the COIN_CHANGE method and it returns the change list which was updated by the GET_MAX_DENOMINATION method and number of coins required to change n cents.
3) **Example** -
    Let amount = 30
    As the amount is > 0 so COIN_CHANGE function will call the recursive function GET_MAX_DENOMINATION with amount = 30 and change list is passed as an input to the next recursive call.



So change_list[25, 5] and length=2 is returned.

2. Give an O(1) (non-recursive!) algorithm to compute the number of returned coins in a).

Ans:

1) **Pseudocode -**

```
NO_OF_RETURNED_COINS(amount):
    count = 0
    if amount = 0:
        return 0
    denominations = [50, 25, 10, 5, 1]
    d = 5
    for i ← 0 to d-1:
        if amount = 0
            return count
        else if amount >= denominations[i]
            count = count + floor(amount / denominations[i])
            amount = amount % denominations[i]

    return count
```

2) **Textual Description** -

Step 1 - check if the amount is 0
    If amount is 0 then return 0
    Else go to step 2
Step 2 - If amount is greater than or equal to denomination d = 50
    Then update count = count + floor(amount / d)
    Update amount = amount % d
Step 3 - Follow step 2 for all denominations - 25, 10, 5, 1 respectively
    If amount becomes zero in any iteration then return count
Step 4 - return the count

a) For any amount, we start comparing it with the highest denomination we have and the maximum number of coins of the highest denomination possible are returned so that we have to return the minimum number of coins.
b) If the if statement is satisfied then count is increased by floor(amount / d) where d is any denomination from (50, 25, 10, 5, 1)
And new_amount will be amount % d
Again same if checked against the next if statement.
Finally the count is returned.
c) For example:
If we have to return the change for 169, we will return the highest denomination coin of 50 for 3 times because after 50*3=150 and 169-150=19 is less than 50. Again for the remainder which is 19 of the amount, we try to match with the highest denomination of coins after 50 which are 25, 10, 5 and 1. So 19 will match with 1 coin of 10 then remainder = 19-10 = 9. Since 9 > 5 so remainder = 9-5 = 4. And 4 > 1 so 4 coins of 1 will get added.
Hence, count = (3 coins of 50) + (1 coin of 10) + (1 coin of 5) + (4 coins of 1) = 3 + 1 + 1 + 4 = 9
d) Since we start from the highest denomination and go in the decreasing order of denomination, it is guaranteed that it will always return with the least number of coins possible.

3) **Proof of Correctness** -
a) Loop Invariant -
   i) We iterate in decreasing order of denomination value.
   ii) At each iteration we deduct the maximum number of the current denomination.
   iii) This ensures that we will deduct the minimum number of coins of subsequent denominations.
   iv) The loop will run till the number of different denominations times or the amount becomes zero the loop will stop.
b) Initialization -

        i) The count before starting the iteration is zero as we have not considered any denomination.
    c) Maintenance -
        i) Before ith iteration, where 1<= i <no_of_denominations-1, the maximum number of coins of denomination[i-1] is taken. So for the ith iteration the amount is deducted as max as possible.
    d) Termination -
        i) When i=no_of_denominations, the loop terminates and the sum of count required for each denomination is given.
        ii) If the amount becomes zero then we return the count.

4) **Time Complexity** -

    If we consider the number of denominations as constant k
    Here, k = 5 for denomination - 50, 25, 10, 5, 1
    In a for loop, basic operations are performed only size(denominations) times.
    here, the size(denominations) = 5
    Time Complexity - O(1+1+1+1+1) = O(5) = O(k) = O(1)
    as k is constant.

5) **Example** -

    Let amount = 49
    Initialize count = 0

Step 1 - Compare the amount with 0. Amount is greater than 0 so go to step 2

Step 2 - Compare the amount with 50. Since if condition is false as 49 < 50 so go to step 3

Step 3 - Compare the amount with 25. Since if condition is true as 49 > 25 so we try to add as many coins of 25 as possible.
    Hence, count = 0 + floor(49 / 25) = 1
        amount = 49 % 25 = 24

Step 4 - Compare the amount with 10. Since if condition is true as new amount 24 > 10 so we
    try to add as many coins of 10 as possible.
    Hence, count = 1 + floor(24 / 10) = 1+2 = 3
        amount = 24 % 10 = 4

Step 5 - Compare the amount with 5. Since if condition is false as new amount 4 < 5 so we go to the next step 6

Step 6 - Compare the amount with 1. Since if condition is true as new amount 4 > 1 so we
    try to add as many coins of 1 as possible.
    Hence, count = 3 + floor(4 / 1) = 3 + 4 = 7

amount = 4 % 1 = 0

Step 7 - Final count = 7 is returned.

3. Show that the above greedy algorithm does not always give the minimum number of coins in a country whose denominations are 1, 6, and 10 cents.

Ans:

Let the amount = 18.

According to our greedy approach, it returns the maximum denomination coin which is less than or equal to the given amount.

This way it will return a series of coins as 10, 6, 1, 1 for an amount of 18 that is 4 coins in total.

But actually this amount can be returned just by using 3 coins of 6. Therefore our greedy method does not always return the minimum number of coins for denominations 1, 6 and 10 cents.

4. Given a set of arbitrary denominations C =(c1,..., cd), describe an algorithm that uses dynamic programming to compute the minimum number of coins required for making change. You may assume that C contains 1 cent, that all denominations are different, and that the denominations occur in increasing order.

Ans:

Assume - all denominations are in increasing order

1) Pseudocode -

```
MIN_COIN_CHANGE(amount, denominations):
      rows = size(denominations)
      columns = amount + 1
      dp_result[rows][columns] = {}       // Initialize dp table
      for i ← 0 to rows-1 :
            dp_result[i][0] = 0

      // Initialize the first row of dp array for number of 1 denomination
      // coins required
      for i ← 1 to columns-1 :
            dp_result[0][i] = i

      for i ← 1 to rows-1 :
            for j ← 1 to columns-1 :
                  if j < denominations[i] :
                        dp_result[i][j] = dp_result[i-1][j]
                  else :
                        dp_result[i][j] = min(dp_result[i-1][j],
                              dp_result[i][j-denominations[i]] + 1)
      return dp_result[rows-1][columns-1]
```

2) Textual Description -
    a) The main idea in the dynamic programming approach is to first find the overlapping subproblems.
    b) For column 0, the count of denominations required will be zero because it is the base case where the amount is 0.
    c) It is given that denomination set C will always include 1 as one of the denominations. So we will fill the 1st row i.e. i=0 with j for each column because every column j >= 1 can be expressed in terms of 1.
    So for any column j >= 1, denomination in terms of 1 = 1 + 1 + 1 + … + j times = j
    d) Let size of denomination array = m,
    For each row i from 1 to m-1 and each column j from 1 to amount,
    There are two cases to compute the value for each cell -
        i) If column j < denomination[i]
        Then we won't consider the current denomination in the solution as it is greater than the amount and the result will come from the previous denomination row for column j
        The solution will be taken from the element -
        dp_result[ i-1 ][j]
        ii) else
        We will take the minimum count between - 1. only considering the denominations smaller than current. And 2. considering current denomination and count from dp_result[i][amount - d(current)]
        The solution will be -
        min(dp_result[i-1][j], dp_result[i][j-denominations[i]]+1)
    e) This will be repeated till the last cell dp_result[m-1][amount]
    f) Final count is returned from the cell dp_result[m-1][amount]
3) Time complexity -
    a) Let size(denominations) = d
    The max number of basic operations are performed in two nested for loops
    Basic operations in the first for loops executed d-1 times
    Basic operations in the second for loop executed amount times
    So basic operations are getting executed inside two for loops
    = (d-1)*amount
    Hence, Time complexity of above algorithm = O(d*amount)

4) Proof of Correctness -
    a) Loop Invariant -
        i) Given a set of denominations {1,c2,c3…cn}, for each row i and amount, we find a minimum number of coins required using set {1, c2, c3…ci}

ii) For each row i the output of the inner for loop always gives the minimum number of coins for an amount at position dp_result[i][amount] required for making change

b) Initialization -

The first loop iteration for row i = 1 finds the solution to give the minimum number of coins required if the denominations set is [denominations[0], denominations[1]] = [1, c2].
Before this first iteration only one denomination is considered which is [1]. As there is only one denomination present, the row [i-1] = row 0 gives the minimum number of coins required for making change.

c) Maintenance -

Before any iteration i = x , the minimum count is saved in the row i-1 and comparison is done between 2 cases- 1) considering the denomination denominations[i-1] in the count and 2) without considering the denomination denominations[i-1] in the count. So the minimum from both the cases is selected. Hence it will give the minimum count in at the position dp_result[i-1][amount]. This gives guarantee that for row i = x minimum count is selected.

d) Termination -

After the termination of the outer loop, we need the minimum count by considering all the denominations so the minimum count is returned from position dp_result[size(denominations)][amount]

5) Example -

Let's consider we have denominations of 1, 5, and 10 and we have to return the count for making a change of 16.
Following table shows the dynamic programming table for values computed:

| Amount→ Denominations ↓ | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 | 14 | 15 | 16 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 1 (row 0) | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 | 14 | 15 | 16 |
| 5 (row 1) | 0 | 1 | 2 | 3 | 4 | 1 | 2 | 3 | 4 | 5 | 2 | 3 | 4 | 5 | 6 | 3 | 4 |
| 10 (row 2) | 0 | 1 | 2 | 3 | 4 | 1 | 2 | 3 | 4 | 5 | 1 | 2 | 3 | 4 | 5 | 2 | 3 |

a) For amount = 0, count = 0 for all denominations
b) For row i = 0, the denomination set is considered as [1], hence count will be j, where j is column number.
c) For row i = 1, the denomination set is considered as [1, 5], count for amount 16 will be given by the cell dp_result[1][16].
It is calculated as follows-

Amount = 16,
As amount > 5 so it will go in else part -
Case 1 - without considering 5,
count = dp_result[1-1][amount] = 16
Case 2 - considering 5,
count = 1 + dp_result[1][amount - 5] = 1+dp_result[1][11]
count = 1 + 3 = 4
So minimum from case 1 and 2 is 4
Hence dp_result[1][amount] = 4

d) For row i = 2, the denomination set is considered as [1, 5, 10], count for amount 16 will be given by the cell dp_result[2][16].
It is calculated as follows-
Amount = 16,
As amount > 10 so it will go in else part -
Case 1 - without considering 10,
count = dp_result[2-1][amount] = 4
Case 2 - considering 10,
count = 1 + dp_result[2][amount - 10]= 1+dp_result[2][6]
count = 1 + 2 = 3
So minimum from case 1 and 2 is 3
Hence dp_result[2][amount] = 3 is returned.

## Problem 2

Consider a sequence of n integers S[1], …S[n]. The longest non-decreasing subsequence problem asks you to find a longest sequence (i1,…,ik) such that ij <ij+1 and S[ij] ≤S[ij+1] for j ϵ{1,…,k-1}.
For example,
Consider the sequence: 3, 45, 23, 9, 3, 99, 108, 76, 12, 77, 16, 18, 4. A longest nondecreasing subsequence is 3, 3, 12, 16, 18, having a length of 5.

1. Let S[1], …S[n] be a sequence of n integers. Denote $l_i$ the length of the longest non decreasing subsequence that ends in (and includes) S[i]. Use dynamic programming to compute $l_i$ and give an algorithm that generates the corresponding non-decreasing subsequence. Using $l_1$,…, $l_n$, how do you solve the longest non decreasing subsequence problem?

Ans:

1) Pseudocode -
   a) Method 1 LONGEST_NONDECREASING_SUBSEQUENCE - Finds the length of the longest non decreasing subsequence.

b) Method 2 FIND_LONGEST_NONDECREASING_SUBSEQUENCE - Finds the longest non decreasing subsequence.

```
LONGEST_NONDECREASING_SUBSEQUENCE(S, n):
      lns[0...n-1]
      lns[0] = 1
      for i ← 1 to n-1:
            lns[i] = 1
            for j ← 0 to i-1:
                  if S[i] >= S[j] and lns[i] < lns[j] + 1:
                        lns[i] = lns[j] + 1
      length_of_lns = MAX(lns, lns+n)
      ln_subsequence =
             FIND_LONGEST_NONDECREASING_SUBSEQUENCE(S, lns, length_of_lns)
      return length_of_lns, ln_subsequence

FIND_LONGEST_NONDECREASING_SUBSEQUENCE(S, lns, length_of_lns):
      index = FIND_INDEX(length_of_lns, lns)
      ln_subsequence[0…length_of_lns-1] = {}
      element = S[index]
      for i ← index to 0:
            if element >= S[i] and length_of_lns = lns[i]:
                  ln_subsequence[length_of_lns-1] = S[i]
                  length_of_lns = length_of_lns - 1
                  element = S[i]
      return ln_subsequence
```

2) Textual Description -

lns - List of length $l_i$ where $l_i$ is the length of the longest non decreasing subsequence that ends in (and includes) S[i]

ln_subsequence - It contains the longest non decreasing subsequence

length_of_lns - It contains the length of the longest non decreasing subsequence

a) For LONGEST_NONDECREASING_SUBSEQUENCE -
To find the length of the longest non decreasing subsequence length that ends in (and includes) S[i], we will find the maximum length $l_j$ -longest non decreasing subsequence length which ends in (and includes) S[j] where 0<=j<i and S[j] <= S[i] and add 1 to it for including S[i]. If such j does not exist then we get the length as 1. We will repeat this for each 1<=i<n-1. Then we choose max from $[l_0 ... l_{n-1}]$ which will give us the length of the longest non decreasing subsequent. And to find that subsequence we call a function FIND_LONGEST_NONDECREASING_SUBSEQUENCE.

This is because

i) Define lns[0…n-1] to store length of the longest non decreasing subsequence that ends in (and includes) S[i]
ii) Put lns[0] = 1 as s[0] will be only longest non decreasing subsequence of length 1 that ends in (and include) S[0]
iii) For each 0 < i <= n-1, put length as 1 in lns[i] and then iterate j from 0 to i-1 for the condition -
if S[j] <= S[i] and lns[i] < lns[j] + 1.
If the condition is true then update the value of lns[i] to lns[j] + 1
iv) Find the max length from lns array.
v) Give S, max_length, lns as an input to the function FIND_LONGEST_NONDECREASING_SUBSEQUENCE to find the actual subsequence.
vi) return length and the actual longest non decreasing subsequence.

b) For FIND_LONGEST_NONDECREASING_SUBSEQUENCE -
The main idea to find the longest non decreasing subsequence is to Backtrack form the index of the longest non decreasing subsequence length in array lns.
The intuition is that if current element is greater than or equal to the Element S[i] then S[i] can be the part of the longest non decreasing Subsequence. So to confirm it check if the $l_i$ = length_of_lns. If it is true then S[i] is added to the ln_subsequence at position length_of_lns-1 and for next iteration length_of_lns is decreased by one and same procedure is done for
element = s[i]

i) Find index of the length of longest non decreasing subsequence from an array lns.
ii) Assign element = S[index]
iii) For each i in index to 0, check if element >= S[i] and length_of_lns = lns[i]
If it is true then add S[i] to ln_subsequence[length_of_lns-1], Decrease length_of_lns by 1 and
element = S[i]
iv) Return ln_subsequence.

3) Time Complexity -
Max size of lns array possible can be n because in worst case the length_of_lns can be equal to size of the original sequence S

a) For an algorithm LONGEST_NONDECREASING_SUBSEQUENCE -
The worst case time taken to perform basic operations inside both the for loops - $O(n^2)$
For i = 1, basic operations are performed 1 time
For i = 2, basic operations are performed 2 times
For i = 3, basic operations are performed 3 times
.

.

.

For i = n-1, basic operations are performed n-1 times

Hence time taken =O(n*(n-1)/2) = O($n^2$)

Time taken to find max length from lns = O(n)

Time taken to find the longest non decreasing subsequence = O(n) (Explanation given in (b) part)

Hence, Total time complexity of the algorithm = O($n^2 + n + n$) = O($n^2$)

b) For an algorithm FIND_LONGEST_NONDECREASING_SUBSEQUENCE worst case time complexity -
The FIND_INDEX algorithm will take time O(n) to find the index of the longest length as we need to search for length_of_lns sequentially in the lns array.
Basic operations inside the for loop can be performed O(n) times because index value can be the last index in array lns in the worst case.
Hence, worst case time complexity = O(n + n ) = O(n)

4) Space Complexity -
   a) For an algorithm LONGEST_NONDECREASING_SUBSEQUENCE -
   Extra auxiliary space required = O(2n) = O(n)
   O(n) space is required to store the sequence of length in lns array
   And O(n) space is required to store the longest non decreasing subsequence in the ln_subsequence array. Because in the worst case the longest non decreasing subsequence can be of size n.
   b) For an algorithm FIND_LONGEST_NONDECREASING_SUBSEQUENCE -
   Extra auxiliary space required = O(n)
   O(n) space is required to store the longest non decreasing subsequence in the ln_subsequence array. Because in the worst case the longest non decreasing subsequence can be of size n.
5) Proof of Correctness -
   a) Algorithm LONGEST_NONDECREASING_SUBSEQUENCE -
      i) Loop Invariant - For each i of the outer for loop, the output is always the length of the longest non decreasing subsequence that ends in (and includes) S[i], where 1<=i<=n-1
      ii) Initialization - Before the beginning of the outer for loop, the longest non decreasing subsequence for S[0] is stored in lns[0] which is 1 because there is only 1 element present for lns[0]. Hence the length of the longest non decreasing subsequence is given correctly for S[0].
      iii) Maintenance - Let 1<=k<i. Before ith iteration the correct length $l_k$ for the longest non decreasing subsequence that ends in (and

includes) S[k] is already stored in lns[k]. So for iteration i, if such k exists then the correct length $l_i$ = $l_k$ + 1 is stored in lns[i]. If such k does not exist then $l_i = 1$ is stored in lns[i]. Hence giving the correct length for ith iteration.

   iv) Termination - The outer loop terminates when i >= n. Hence by the maintenance step for all i, where 0 <= i < n-1, the correct length of the longest non decreasing subsequence that ends in (and includes) S[i] is stored in array lns.

b) Algorithm FIND_LONGEST_NONDECREASING_SUBSEQUENCE -
   i) Loop Invariant - After each iteration of the for loop, we will get the ith element of the longest non decreasing subsequence where 0<=i<=length_of_lns-1
   ii) Initialization - Before 1st iteration of the for loop, there is no element in the ln_subsequence.
   iii) Maintenance - Let i > k >= 0. Before ith iteration the correct element S[k] is stored in ln_subsequence. For ith iteration, S[i] can be placed in ln_subsequence based on the condition using the last element S[k] which should be element S[k] >= S[i] and length_of_lns = lns[i]. So it makes sure to correctly choose the element S[i] to place in ln_subsequence.
   iv) Termination - After termination of the for loop at i = -1, we get a ln_subsequence so the correct longest non decreasing subsequence is returned.

6) Example -
   a) LONGEST_NONDECREASING_SUBSEQUENCE -
   Input - S[] = {4, 11, 3, 12}
   lns[] = {1, 1, 1, 1} (initially)

   Iteration-wise simulation -
   For i = 1, j = 0
      S[1] > S[0] and lns[1] < lns[0] + 1 {lns[1] = 2}
      lns[] = {1, 2, 1, 1}

   For i = 2, j = 0
      S[2] < S[0] {No change}

   For i = 2, j = 1
      S[2] < S[1] {No change}

   For i = 3, j = 0
      S[3] > S[0] and lns[3] < lns[0] + 1 {lns[3] = 2}
      lns[] = {1, 2, 1, 2}

```
For i = 3, j = 1
        S[3] > S[1] and Ins[3] < Ins[1] + 1 {Ins[3] = 3}
        Ins[] = {1, 2, 1, 3}

For i = 3, j = 2
        S[3] > S[2] and Ins[3] > Ins[2] + 1 {No change}

final Ins[] = {1, 2, 1, 3}
length_of_Ins = MAX({1, 2, 1, 3}) = 3
In_subsequence = {4, 11, 12}
// returned from method
//FIND_LONGEST_NONDECREASING_SUBSEQUENCE
return length_of_Ins, In_subsequence
```

b) FIND_LONGEST_NONDECREASING_SUBSEQUENCE -

```
Input - S[] = {4, 11, 3, 12}
        Ins[] = {1, 2, 1, 3}
        length_of_Ins = 3
index = FIND_INDEX(3, Ins) = 3
In_subsequence[] = {}(initially)
element = S[3] = 12

Iteration-wise simulation -
For i = 3,
        element = S[3] and length_of_Ins = Ins[3]  {Condition true}
                In_subsequence[2] =12
                In_subsequence[] = {12}
                length_of_Ins = 2
                element = S[3] = 12

For i = 2,
        element > S[2] and length_of_Ins != Ins[2] {No change}

For i = 1,
        element > S[1] and length_of_Ins = Ins[1] {Condition true}
                In_subsequence[1] =11
                In_subsequence[] = {11,12}
                length_of_Ins = 1
                element = S[1] = 11

For i = 0,
        element > S[0] and length_of_Ins = Ins[0]  {Condition true}
                In_subsequence[0] =4
                In_subsequence[] = {4,11,12}
```

length_of_lns = 0
element = S[0] = 4

ln_subsequence[] - {4, 11, 12}
return ln_subsequence

2. Describe an alternative algorithm that uses the Longest Common Subsequence (LCS) Problem that we have discussed in class to solve the longest non-decreasing subsequence problem.

Ans:

1) Pseudocode -
   a) Method 1 LONGEST_NONDECREASING_SUBSEQUENCE - Finds the length of the longest non decreasing subsequence.
   b) Method 2 FIND_LONGEST_COMMON_SUBSEQUENCE - Finds the longest non decreasing subsequence.

```
LONGEST_NONDECREASING_SUBSEQUENCE(S, n):
      sorted_S[] = sort(S)
      dp[0...n][0...n]
      for i ← 0 to n:
            dp[0][i] = 0
            dp[i][0] = 0
      for i ← 1 to n:
            for j ← 1 to n:
                  if S[i-1] = sorted_S[j-1]:
                        dp[i][j] = dp[i-1][j-1] + 1
                  else:
                        dp[i][j] = MAX(dp[i][j-1], dp[i-1][j])
      length_of_lns = dp[n][n]
      ln_subsequence =
        FIND_LONGEST_COMMON_SUBSEQUENCE(S, sorted_S, dp,
                                                length_of_lns)
      return length_of_lns, ln_subsequence

FIND_LONGEST_COMMON_SUBSEQUENCE(S, sorted_S, dp, length_of_lns):
      lc_subsequence[0...length_of_lns-1] = {}
      i = n, j = n
      while i > 0 and j > 0:
            if S[i-1] = sorted_S[j-1]:
                  lc_subsequence[length_of_lns-1] = S[i-1]
                  i = i-1
                  j = j-1
                  length_of_lns = legth_of_lns -1
        else if dp[i][j-1] > dp[i-1][j]:
              j = j-1
        else:
              i = i-1
  return lc_subsequence
```

2) Textual Description -
   a) For algorithm LONGEST_NONDECREASING_SUBSEQUENCE -
      If we sort the sequence S, it will give us the sequence sorted_S and finding the longest common subsequence between S and sorted_S will give us the longest non decreasing subsequence.
      The length of the longest common subsequence is obtained from the cell dp[n][n]
      Algorithm-
      i) Get a new sequence sorted_S by sorting the original sequence S.
      ii) Fill the 0th row and 0th column of the dp matrix with 0.
      iii) For each cell dp[i][j], where 1<=i<=n and 1<=j<=n,
           if S[i-1] = sorted_S[j-1] that means the current element is included in the longest common subsequence. And hence the length of longest common subsequence till the current element will be dp[i-1][j-1] + 1
           else - S[i-1] is not equal to sorted_S[j-1], so longest common subsequence will come from MAX(dp[i-1][j], dp[i][j-1])
      iv) length_of_lns = dp[n][n]
      v) Find the longest common subsequence using method FIND_LONGEST_COMMON_SUBSEQUENCE(explained in b))
      vi) Return length_of_lns and ln_subsequence
   b) For algorithm FIND_LONGEST_COMMON_SUBSEQUENCE -
      The main idea to find the longest common subsequence is start from the Cell dp[n][n], see if the S[i] is equal to sorted_S[j] where 1<=i<=n and 1<=j<=n, if they are equal then add S[n-1] in the lc_subsequence and go to the cell dp[i-1][j-1]. Otherwise, go to the cell from where the dp[i][j] value came from i.e. MAX(dp[i-1][j], dp[i][j-1]). By Backtracking this way, find the longest common subsequence.
      i) Initialize i=n, j=n
      ii) If S[i-1] = sorted_S[j-1], then add S[i-1] to lc_subsequence[length_of_lns-1],
          decrease i, j and length_of_lns by 1
      iii) Else if dp[i][j-1] > dp[i-1][j] then decrease j by 1
      iv) Else decrease i by 1
      v) Repeat step 2,3 and 4 until i or j becomes 0.
      vi) Return lc_subsequence.
3) Proof of Correctness -

a) Algorithm LONGEST_NONDECREASING_SUBSEQUENCE -
   i) Loop Invariant - For each row i the output of the inner for loop always gives the length of the longest common subsequence between S[0…i-1] and sorted_S[0…n-1]
   ii) Initialization -
   We have considered that i is 1<= i <= n
   Before this first iteration, i=0 which means the two sequences are S(empty set) and sorted_S[0…n-1]. For that dp[0][n] value is 0 which means there is no common subsequence between these two sequences, which is consistent because S is an empty set for i=0.
   iii) Maintenance - Before any iteration i, where 1<i<=n the length of the longest common subsequence between S[0…i-2] and sorted_S[0…n-1] is given at dp[i-1][n]. So our algorithm is performing correctly before any iteration i.
   iv) Termination - After the termination of the outer loop(when i>n), the length of the longest common subsequence between S[0…n-1] and sorted_S[0…n-1] is present at dp[n][n], which is our desired output. Hence our algorithm works correctly.

b) Algorithm FIND_LONGEST_COMMON_SUBSEQUENCE -
   i) Let X ={ x1, x2, . . . , xm } and Y ={ y1, y2, . . . , yn } be sequences, and let Z ={ z1, z2, . . . , zk } be any LCS of X and Y .
   Case 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 $Y_{n-1}$.
   Case 2: If $x_m$ != $y_n$, then $z_k$ != $x_m$ implies that Z is an LCS of $X_{m-1}$ and Y .
   Case 3: If $x_m$ !=$y_n$ , then $z_k$ != $y_n$ implies that Z is an LCS of X and $Y_{n-1}$.
   ii) For case 1:
      (1) If $z_k$ != $x_m$, then we can append $x_m$ = $y_n$ to Z to obtain a common subsequence of X and Y of length k + 1, contradicting the supposition that Z is a longest common subsequence of X and Y . Thus, we must have $z_k$ = $x_m$ = $y_n$ . Now, $Z_{k-1}$ is of length-(k−1) common subsequence of $X_{m-1}$ and $Y_{n-1}$. We want to show that it is an LCS. Suppose to contradict that there is a common subsequence S of $X_{m-1}$ and $Y_{n-1}$ with length greater than k− 1. Then, appending $x_m$ = $y_n$ to S produces a common subsequence of X and Y whose length is greater than k, which is a contradiction.

iii) For case 2:
   (1) If $z_k$ != $x_m$, then Z is a common subsequence of $X_{m-1}$ and Y . If there were a common subsequence S of $X_{m-1}$ and Y with length greater than k, then S would also be a common subsequence of $X_m$ and Y , contradicting the assumption that Z is an LCS of X and Y .

iv) For case 3:
   (1) If $z_k$ != $y_n$, then Z is a common subsequence of $X$ and $Y_{n-1}$ . If there were a common subsequence S of $Y_{n-1}$ and X with length greater than k, then S would also be a common subsequence of $X$ and $Y_n$ , contradicting the assumption that Z is an LCS of X and Y .

4) Time Complexity -
   a) For algorithm LONGEST_NONDECREASING_SUBSEQUENCE -
      i) Time taken to sort the sequence S is O(nlg(n))
      ii) Time taken to fill the 0th row and column is O(n) as there are n elements in the 0th row and 0th column.
      iii) Time taken to perform basic operations inside two nested for loops which goes from 1 to n each is O($n^2$)
      iv) Time taken to give the longest common subsequence is O(n) which is calculated in part b)
      v) Hence the total time complexity of this algorithm is

         O(nlg(n) + n + $n^2$ + n) = O($n^2$)
   b) For algorithm FIND_LONGEST_COMMON_SUBSEQUENCE -
      i) The number of basic operations are performed in the while loop which runs the length_of_lns times. This is because the purpose of this while loop is to find the elements in the longest common subsequence. In each iteration one element from both the sequences are compared and the length of both the sequences is n. Hence the basic operations can be performed n times. The loop will terminate when one of the i and j becomes 0.
      ii) Hence, the time complexity of this algorithm = O(n)
5) Space Complexity -
   a) For algorithm LONGEST_NONDECREASING_SUBSEQUENCE -
      Extra auxiliary space required is for sorted_S sequence which is of size O(n), storing dp elements which is of size O($n^2$), and storing the longest non decreasing subsequence which is of size O(n)
      Hence total space complexity = O(n + $n^2$ + n) = O($n^2$)
   b) For algorithm FIND_LONGEST_COMMON_SUBSEQUENCE -
      Extra auxiliary space required apart from the input parameters is O(n) to

store the longest common subsequence.

6) Example -

   a) For algorithm LONGEST_NONDECREASING_SUBSEQUENCE -
   Input - S[] = {4, 11, 3}, n=3
   sorted_S[] = {3, 4, 11}
   dp[0…3][0…3]        //initialize dp table

   Filling up the 0th row and 0th column by 0

| sorted_S → S ↓ | 0 | 1 [3] | 2 [4] | 3 [11] |
|---|---|---|---|---|
| 0 | 0 | 0 | 0 | 0 |
| 1 [4] | 0 | | | |
| 2 [11] | 0 | | | |
| 3 [3] | 0 | | | |

For i = 1 and j = 1,
S[0] != sorted_S[0] hence dp[1][1] = MAX(dp[0][1], dp[1][0]) = 0

For i = 1 and j = 2,
S[0] = sorted_S[1] hence dp[1][2] = dp[0][1] + 1 = 1

For i = 1 and j = 3,
S[0] != sorted_S[2] hence dp[1][3] = MAX(dp[0][3], dp[1][2]) = 1

For i = 2 and j = 1,
S[1] != sorted_S[0] hence dp[2][1] = MAX(dp[1][1], dp[2][0]) = 0

For i = 2 and j = 2,
S[1] != sorted_S[1] hence dp[2][2] = MAX(dp[1][2], dp[2][1]) = 1

For i = 2 and j = 3,
S[1] = sorted_S[2] hence dp[2][3] = dp[1][2] + 1 = 2

For i = 3 and j = 1,
S[2] = sorted_S[0] hence dp[3][1] = dp[2][0] + 1 = 1

For i = 3 and j = 2,
S[2] != sorted_S[1] hence dp[3][2] = MAX(dp[2][2], dp[3][1]) = 1

For i = 3 and j = 3,

S[2] != sorted_S[2] hence dp[3][3] = MAX(dp[2][3], dp[3][2]) = 2

| sorted_S → S ↓ | 0 | 1 [3] | 2 [4] | 3 [11] |
|---|---|---|---|---|
| 0 | 0 | 0 | 0 | 0 |
| 1 [4] | 0 | 0 | 1 | 1 |
| 2 [11] | 0 | 0 | 1 | 2 |
| 3 [3] | 0 | 1 | 1 | 2 |

length_of_lns = dp[3][3] = 2
ln_subsequence = [4, 11]      //returned by the algorithm
//FIND_LONGEST_COMMON_SUBSEQUENCE
Hence, the length of the longest non decreasing subsequence is 2 and
Corresponding longest non decreasing subsequence is [4, 11]

b) For algorithm FIND_LONGEST_COMMON_SUBSEQUENCE -

Input - S[] = {4, 11, 3},
sorted_S[] = {3, 4, 11}
dp[0…3][0…3]
dp -

| sorted_S → S ↓ | 0 | 1 [3] | 2 [4] | 3 [11] |
|---|---|---|---|---|
| 0 | 0 | 0 | 0 | 0 |
| 1 [4] | 0 | 0 | 1 | 1 |
| 2 [11] | 0 | 0 | 1 | 2 |
| 3 [3] | 0 | 1 | 1 | 2 |

length_of_lns = 2
Initialize i = 3, j = 3

While loop -
For i = 3 and j = 3,
S[2] != sorted_S[2] and dp[2][3] > dp[3][2]
Hence i = i-1

        i = 2
For i = 2 and j = 3,
        S[1] = sorted_S[2]
        Hence,
        i = 1
        j = 2
        lc_subsequence[1] = 11
        length_of_lns = 1
For i = 1 and j = 2,
        S[0] = sorted_S[1]
        Hence,
        i = 0
        j = 1
        lc_subsequence[0] = 4
        length_of_lns = 0
Since i = 0 so while loop breaks and
lc_subsequence[] = {4, 11} is returned.

## Problem 3

We showed that multiplying two matrices

$$\begin{array}{ccccc} C & \leftarrow & A & * & B \\ mxp & & mxn & & nxp \end{array}$$

requires mnp scalar multiplications. You are given the following matrix chain:

$$\begin{array}{ccccccc} A1 & * & A2 & * & A3 & * & A4 \\ 20x25 & & 25x5 & & 5x10 & & 10x30 \\ doxd1 & & d1xd2 & & d2xd3 & & d3xd4 \end{array}$$

Denote m[i,j] the minimum number of scalar multiplications to compute Ai * … *Aj. We showed the following recurrence:

$$m[i,j] \quad := \quad \begin{array}{l} 0\,; \quad if\ i\ =\ j \\ min_{i-1<k<j}\{m(i,k)\ +\ m(k+1,j)\ +\ d_{i-1}d_kd_j\}\ ;\ otherwise \end{array}$$

1. Fill the Table below with the missing values for m[i,j]. Also, for each m[i,j], put the corresponding value k, where the recurrence obtains its minimum value, next to it.

Ans:

| i \ j | 1 | 2 | 3 | 4 |
|---|---|---|---|---|
| 1 | 0 | 2500, k=1 | 3500, k=2 | 7000, k=2 |

| | | | | |
|---|---|---|---|---|
| 2 | Undefined | 0 | 1250, k=2 | 5250, k=2 |
| 3 | Undefined | Undefined | 0 | 1500, k=3 |
| 4 | Undefined | Undefined | Undefined | 0 |

2. What is the minimum number of scalar multiplications required to compute A1* A2 *A3 *A4? How did you get the result?

Ans: The minimum number of scalar multiplications required to compute A1* A2 *A3 *A4 is 7000.

$$m[i,j] \quad := \quad 0\,; \quad if\ i\ =\ j$$
$$min_{i-1<k<j}\{m(i,k)\ +\ m(k\ +\ 1,j)\ +\ d_{i-1}d_k d_j\}\ ;\ otherwise$$

For m[1,4] ,

i = 1, j=4

So k = [1,2,3]

For k=1,

A = m(1,1) + m(2,4) + 20*25*30 = 0 + 5250 + 15000 = 20250

For k=2,

B = m(1,2) + m(3,4) + 20*5*30 = 2500 + 1500 + 3000 = 7000

For k=3,

C = m(1,3) + m(4,4) + 20*10*30 = 3500 + 0 + 6000 = 9500

So Minimum value came was min(A, B, C) = 7000.

As  m[i,j] denotes the minimum number of scalar multiplications to compute Ai * … *Aj

So for A1 * A2 * A3 * A4 result can be found at m[1, 4] which is 7000.

Minimum number of scalar multiplications can be found at the top right corner of the above table.

3. Give the optimal order of computing the matrix chain by fully parenthesizing the matrix chain below.

A1 * A2 * A3 * A4

Ans:

Step 1:

For multiplication from A1,.. A4 we look at the k value at position (1, 4) in the table.

k = 2

So we parenthesize from position 2

We get, (A1 * A2) * (A3 * A4)

Step 2:

For the first part (A1 * A2)

Look up k value at position (1, 2)

k = 1

We get ( (A1) * (A2) ) * (A3 * A4)

Step 3:

To compute (A3 * A4)

Look up k value at position (3, 4)
k = 3
Finally, ( ( (A1) * (A2) ) * ( (A3) * (A4) ) )

Hence,  the optimal order of computing the matrix chain is
( ( A1 * A2 ) * ( A3 * A4 ) )

4. How many scalar multiplications are used to compute ((A1*(A2*A3))*A4)? Keep the order of matrix multiplications indicated by the brackets. Justify your answer.

Ans:

1) Number of scalar multiplications done in a matrix multiplication of dimensions (m x n) and (n x p) are mnp.
2) Let's start debunking the chain with the innermost parentheses.
3) Step 1: ((A1***(A2*A3)**)*A4)
   Scalar multiplications in A2*A3 = 25*5*10 = 1250
   Resultant matrix A23 has dimension (25 x 10)
4) Step 2: (**(A1*(A2*A3))***A4)
   Scalar Multiplications in A1*A23 = 20 * 25 * 10 = 5000
   Resultant Matrix A123 has dimension (20 * 10)
5) Step 3: **( ((A1*A2) * A3 ) *A4)**
   Scalar Multiplications = 20 * 10 * 30 = 6000
   Resultant Matrix A1234 has dimension = (20 x 30)
6) Total Number of scalar multiplications =1250 + 5000 + 6000 = 12250
7) It is clearly greater than the sequence we follow for multiplication as obtained from the DP algorithm.

## Problem 4

Suppose you want dinner at a restaurant on the road from Raleigh to Chapel Hill. You start in Raleigh. Your car holds enough gas to travel d miles. Let d1 < d2 <... < dn be the locations of all the gas stations along the road where di is the distance from Raleigh to the gas station i. Assume that d1<=d, that neighboring gas stations are at most d miles apart and that the restaurant is right next to gas station n. Your goal is to make as few as possible stops along the way.
Describe in text and pseudocode a greedy algorithm for this problem. Justify that your algorithm runs in time O(n). Describe the greedy choice and how it reduces your problem to a smaller instance. Prove that your algorithm is correct.

Ans:

1) Textual Description -
   a) The greedy approach always looks for the best possible solution at the moment. That means the locally optimal solution it gives.
   b) Here, consider the start_point as 0. We have given distance {d1, d2, d3,...dn} from the starting point.

c) Suppose, for any point di, where 1<= i <=n, if it is reachable from the starting point i.e. if di - start_point < d then we won't add di to the set of stops where we need to stop to fill the gas.

d) If for a stop di, if distance from the starting point is exactly d,
   i.e. di - start_point = d,
   Then , we need to add di in the set of stops where we need to stop to fill the gas. And the new starting point will be the gas station di for further calculation.
   start_point = di

e) If for a stop di, if distance from the starting point is greater than d,
   i.e. di - start_point > d,
   Then , we need to add the stop before di, which is di-1 in the set of stops where we need to stop to fill the gas, otherwise di will not be reachable from the start_point. Hence the new starting point will be the gas station di-1 for further calculation.
   start_point = di-1

f) d1 is always reachable from the first start_point. And for all consecutive stops, the distance is always <= d.

g) Algorithm -
   i) Step 1- Let start_point =0, n = number of gas stations
      Define result set as an empty array
   ii) Step 2 - For i, 0<=i<=n-1
      (1) If the distance of gas station i from the start_point is greater than d then add gas station di-1 to the result_set
          And make di-1 as new start_point
      (2) Else if the distance of gas station i from the start_point is equal to d  and i is not the last gas station, then add gas station di to the result_set
          And make di as new start_point
   iii) Step 3 - Repeat step 2 for all i, 0<=i<=n-1
   iv) Step 4 - return the result_set and its length

2) Pseudocode -

```
MINIMUM_STOPS_POSSIBLE(dist[] = {d1, d2, d3,...,dn}, d, n):
      start_point = 0
      result_set[] = {}
      for i ← 0 to n-1:
            if dist[i] - start_point > d:
                  result_set.append(dist[i-1])
                  start_point = dist[i-1]
            else if (dist[i] - start_point = d and i != n-1):
                  result_set.append(dist[i])
```

```
                start_point = dist[i]
    return size(result_set), result_set
```

3) Time complexity -
    a) We will check the reachability for every gas station to decide whether to fill the gas at that station or not. So in worst case we will add n-1 gas stations because the restaurant is next to nth gas station so we won't visit nth gas station, basic operations are performed n-1 times in the for loop, where n is the number of gas stations.
    b) Hence, the time complexity of the greedy algorithm is O(n-1) = O(n)
4) Space complexity -
    a) We have used the extra auxiliary space result_set array to store the list of gas stations which we need to visit.
    b) In the worst case we might need to visit n-1 gas stations because the restaurant is next to nth gas station so we won't visit nth gas station.
    c) The worst case will arrive when each gas station is at a distance d from the consecutive gas station so we need to add n-1 gas stations to the result_set.
    d) Hence the space complexity of above algorithm(considering extra auxiliary space) = O(n-1) = O(n)
5) Describe the greedy choice and how it reduces your problem to a smaller instance -
    a) The problem is to find the gas stations which we need to visit in order to reach the destination. So if we consider a smaller problem which would be whether to visit a gas station i or not will reduce our problem to a smaller problem.
    b) So the greedy way to solve this problem(our smaller problem which is whether to visit a gas station or not) is to find the locally optimum solution. I.e. choose the gas station to visit only if it is required.
    c) If the previous gas station which we visited is exactly d miles away from the current gas station then we need to visit the current gas station.
    d) If the current gas station is not reachable from the previous gas station which we visited then we need to visit the gas station which is just before our current gas station.
    e) Hence, in this way we make our greedy choices to the smaller problem of whether to visit a gas station or not.
6) Example -

    Input - dist[] = {1, 2, 5, 8, 10},
                d = 3
                n = 5
    start_point = 0
    result_set[] = {}

```
For i = 0,
        dist[0] - 0 < 3 {No change}

For i = 1,
        dist[1] - 0 < 3 {No change}

For i = 2,
        dist[2] - 0 > 3
        result_set{2}
        start_point = dist[1] = 2

For i = 3,
        dist[3] - 2 = 8 - 2 = 6 > 3
        result_set{2, 5}
        start_point = dist[2] = 5

For i = 4,
        dist[4] - 5 = 10 - 5 = 5 > 3
        result_set{2, 5, 8}
        start_point = dist[3] = 8

return result_set{2, 5, 8}, 3
```

7) Proof of correctness -
   a) Loop Invariant -
      i) The output of the loop is always the result_set that contains the minimum gas stations which need to be visited to reach the gas station which is one stop before the current gas station.
   b) Initialization -
      i) Before the first loop iteration, there are no elements in the result_set which is consistent with the loop invariant because we have not considered any gas station before starting the loop.
      ii) And there are no gas stations before the d1. So for a starting point, we don't need to consider any gas station.
   c) Maintenance -
      i) Before ith iteration, where 0<=i<=n, the starting point can be anything between 0, d1, d2…di-1. So we can reach till di-1.
      ii) Hence, in our result_set, we have all the minimum gas stations which need to be visited in order to reach di-1 gas station.
   d) Termination -
      i) After the loop terminates when i = n+1, we have result_set where all the minimum gas stations are stored which need to be visited to reach dn gas stations. And our destination is next to dn gas station.

ii) We have all the gas stations which need to be visited to reach the destination.
iii) Hence our algorithm is correct.