DAA Assignment 3

1) Analysize algorithms for coin change problem.

a) (4 points) Give a recursive algorithm that generates a similar series of coins for changing *n* cents. Don't use dynamic programming for this problem.

Pseudocode:

```
// All the denominations we have.
denominations[5] = {50, 25, 10, 5, 1};

makeChange(amountToBeChanged)
{
      // Base case to handle invalid amount which cannot be
      // generated using the following denominations.
      if(amountToBeChanged < 0){
            print("invalid amount");
            return;
      }

      // Base case to handle when the amount is reduced to or is
      // zero.
      if(amountToBeChanged == 0){
            return;
      }

      // Main logic to print out the all the denominations.
      for(i = 0; i < 5; i++){
            if(amountToBeChanged >= denominations[i]){
                  print(denominations[i]);
                  makeChange(amountToBeChanged -  denominations[i]);
                  break;
            }
      }
}
```

Textual description: First, we examine our base cases which is if the input is negative or zero. Negative input is invalid, while 0 means no further work to do. For positive input, the algorithm tries to find the largest denomination which is less than or equal to the given amount. The algorithm outputs the coin value with that denomination. Now the amount for which we need to make change becomes the original amount subtracting that denomination. So the original problem converts to a new subproblem which is to make changes for the updated amount. Therefore, the algorithm recursively calls itself with the updated amount. Since the smallest denomination is one cent, this function will always be able to find the coin series for a positive amount greater than 0.

Example: In order to make the change amount for 67 cents, this algorithm first checks if a 50 cent coin could be used. Then, a recursive call is executed on an input of 67 - 50 = 17 cents. In this step of recursion 50 and 25 cents are tested but all are larger than 17. The algorithm keeps going on until a 10 cent appears which is the largest denomination less than or equal to 17. Now it takes the 10 cent coin into account. Now there remains 17 - 10 = 7 cents. A recursive call with an updated amount of 7 is made. This time the algorithm will find 5 cents because all other denominations are larger than 7 cents. Similar to the above steps, taking 5 cent coin, recursively calls with 7 - 5 = 2 cents. A recursive call is made with an updated amount which is 2. This time the algorithm will find 1 cent because all other denominations are larger than 2 cents. Now it takes 1 cent coin and again recursively calls with 2 - 1 = 1 cent. The same procedure as the last call with amount as 1, the algorithm finds 1 cent, then it recursively calls with 1 - 1 = 0 cent. Since the updated value is 0 which is in one of our base cases, it means that we have found the coin series which corresponds to 67 coins, thus the algorithm returns.

Proof of correctness:

- **Recursion Invariant:** In each recursive call, the algorithm deducts the given amount with possible denominations looking to fufill the given amount.

- **Base Case:** If the given amount is one among the bases cases (i.e) Negative or 0 then the algorithm returns in a single step. Negative input is invalid, while 0 means no further work to do.

- **Maintenance:** For positive input, the algorithm tries to find the largest denomination which is less than or equal to the given amount. The algorithm outputs the coin value with that denomination. Now the amount for which we need to make change becomes the original amount subtracting that denomination. So the original problem converts to a new subproblem which is to make changes for the updated amount. Therefore, the algorithm recursively calls itself with the updated amount until it finds the answer. Since the smallest denomination is one cent, this function will always be able to find the coin series for a positive amount greater than 0.

- **Termination:** The algorithm terminates when the given amount is reduced to 0 indicating that the denominations adding up to that amount are found.

Analysis of time & space complexity: Time complexity depends on the amount for which you want to make the change. Let us assume that the amount is n. In each and every recursive call, n is reduced by at least 1 cent or at most 50 cents. Therefore, the number of recursive calls is between n/50 to n. In every recursive call, the number of arithmetic operations is 1 which is a deduction. Therefore, the time complexity of the approach is O(n).

And for each recursive call the the algorithm uses stack space, which in worst case is O(n) in total.

b) (4 points) Write an O(1) (non-recursive!) algorithm to compute the number of returned coins.

Pseudocode:

```
// All the denominations we have in the decreasing order.
denominations[5] = {50, 25, 10, 5, 1};

makeChange(amountToBeChanged)
{
      // Base case to handle invalid amount which cannot be
      // generated using the following denominations.
      if(amountToBeChanged < 0){
            print("invalid amount");
            return;
      }
      noOfCoins = 0;
      // Main logic to count the number of coins.
      for (i=0; i<5; i++)
      {
            if(amountToBeChanged == 0){
                  break;
            }
            else{
                  noOfCoins += floor (amountToBeChanged / denominations[i]);
                  amountToBeChanged = (amountToBeChanged %
            denominations[i]);
            }
      }
      print(noOfCoins);
}
```

Textual description: First, we test for invalid input. Then, we compute in order of decreasing denominations on how many coins of a certain denomination value have to be returned. We adjust the amount by making deductions with the selected coins to make coin change correspondingly and iterate with the next denomination. Since the smallest denomination is one cent, this procedure will always be able to make change for a positive amount and it produces the same coin set as the previous recursive algorithm.

Example: For the 67 cents example, in the for loop, when i equals 0, we get noOfCoins as 1 and amount as 17, meaning 1 50-cent coin can be used, and another 17 cents need to be changed. When i equals 1, since the denomination of 25-cent is larger than 17 cents, noOfCoins keeps the same. When i equals 2, we get 1 10-cent coin and the remainder is 7 cents. When i equals to 3,

1 5-cent coin is used leaving 2 cents as remainder. And finally, when i equals 4, 2 1-cent coins are used. Therefore, the result is 5.

Proof of correctness:

- **Loop Invariant:** In each iteration, the algorithm keeps the count of number of coins which are required in order to fulfill the given amount.

- **Initialization:** Initially the number of coins is 0, indicating that for zero amount we will not require no coins.

- **Maintenance:** Here, we compute in order of decreasing denominations on how many coins of a certain denomination value have to be returned. We adjust the amount by making deductions with the selected coins to make coin change correspondingly and iterate with the next denomination. Since the smallest denomination is one cent, this procedure will always be able to make change for a positive amount and it produces the same coin set as the recursive version of this algorithm.

- **Termination:** The algorithm terminates when the given amount is reduced to 0 indicating that the denominations adding up to that amount are found.

Time & space complexity Analysis: For negative input, the code returns in a single step saying that it is an invalid input. For positive input, the number of operations performed is always the same which is 5 loop iterations, and all iterations have exactly the same number of basic operations. Therefore, the time complexity is O(1).

As this algorithm makes use of constant space excluding the given input, the space complexity is O(1).

c) (4 points) 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.

Counter-example: 13 cents need the changing of one 10-cent coin and three 1-cent coins using the above algorithm which results in a total of 4 coins. However, the minimum number of coins changing is with two 6-cent coins and one 1-cent coin which results in a total of 3 coins.

d) (6 points) 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.

Pseudocode:

```
// input Value and the denominations.
Input:V, {C[1], C[2], …, C[d]}
// minimum number of coins to make change.
Output: minimumCoin

// setting the first index to 0, indicating to generate a value
// of 0 we need 0 coins.
Table[0] = 0

// Main bottom-up approach logic to find minimum number of
coins.
for(i = 1; i <= V; i++){
      // initialization
      Table[i] = V + 1
      for(j = 1; j <= d; j++){
            if(C[j] > i){
                break
            }
            if(1 + Table[i - C[j]] < Table[i]) {
                Table[i] = 1 + Table[i - C[j]]
            }
       }
      // minimum number of coins for (V, C).
      minimumCoin = Table[V]
      print (minimumCoin)
}
```

Textual Description and Argument for Correctness:

The above algorithm follows bottom-up approach and accumulates results from the following recurrence relation:

minCoin(V, C) = 0, when V = 0
minCoin(V, C) = min(1 + minCoin(V - Ci, C), minCoin(V, C)), when V > 0 and Ci <= V over 1 <= i <= d

The base case is trivial as a value of 0 needs 0 coins to make the change. The recurrence relation follows from the fact that after using a coin Ci as part of the change, we still need to make a change for the value of (V-Ci) using the same set of denominations. We have to make a change for (V - Ci) using the minimum number of coins. So, in total, we need that number + 1 coin, which is Ci. Now, we can use any coin Ci from the set of coins C to make the first change so long as Ci <= V. Thus, overall Ci, which are <= V, we will use one for which 1 + minimumCoin(V - Ci, C) is the minimum to ensure that minimumCoin(V, C) is assigned the smallest possible value. Since 1 is part of the coin set, there will always be at least one Ci <= V available.

The problem has the optimal substructure property since we have constructed the solution for V using the optimal solution for smaller values such as V - Ci. We have initialized a DP table, Table, with the value 0 at the 0th index, and an infinity at the indices from 1 to V. We have computed the table bottom up starting at index 1, i.e., V=1, and ending at index V. For each value, we have iterated over the set of coins to find out which one, being used as the first change, gives the optimal answer. Since the optimal solution for the values smaller than the current value is already computed, we have simply looked up the table.

Example:

V = 11, C = {1, 2, 5}

For this example the memorization table gets filled as follows. The above textual description gives the explanation on the working of this part.

Table = {0, 1, 1, 2, 2, 1, 2, 2, 3, 3, 2, 3}

In this finally filled table the value in the last index gives the minimum number of coins required to satisfy the given amount .i.e. minimumCoin = Table[11] = 3.

Time and space complexity:

The initialization of the table takes θ(V) time. Then, for each value, we have to iterate from C1 until Ci > V. So, in the worst case, for a value we need O(d) time. Hence, for V values we need O(Vd) time. Therefore the time complexity of the algorithm is O(Vd).

The space complexity is θ(V+d) for the table with V items and the list of coins with d items.

2) Matrix chain multiplication.

a) (6 points) 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.

i/j	**1**	**2**	**3**	**4**
1	m[i,j]=0, k=0	m[i,j]=30, k=1	m[i,j]=42, k=1	m[i,j]=58, k=3
2	Undefined	m[i,j]=0, k=0	m[i,j]=30, k=2	m[i,j]=54, k=3
3	Undefined	Undefined	m[i,j]=0, k=0	m[i,j]=40, k=3

4	Undefined	Undefined	Undefined	m[i,j]=0, k=0

b) (1 point) What is the minimum number of scalar multiplications required to compute the matrix chain? - **58**.

c) (2 points) Give the optimal order of computing the matrix chain by fully parenthesizing the matrix chain below.

((A1 * (A2 * A3)) * A4).

d) (1 points) How many scalar multiplications are used to compute $(A_1 * A_2) * (A_3 * A_4)$? Keep the order of matrix multiplications indicated by the brackets. - **110.**

3) Maximum Palindromic Subsequence Problem.

a) (4 points) Describe the optimal substructure of the MPSP and give a recurrence equation for L(i,j).

A problem is said to have optimal substructure if an optimal solution can be constructed from optimal solutions of its subproblems.

Optimal substructure:

Let L(1,k) be length of longest palindromic subsequence. Let S(1,k) be the string under consideration of length k. If the 1st character and kth character of the string S(1,k) is the same, then L(1,k) will be 2 + L(2,k-1) else if they are not the same, then L(1,k) will be maximum of L(1,k-1) and L(2,k).

Recurrence equation:

L(i,j) = 1 if i == j
= 2 if i-j == 1 and S[i] == S[j]
= L(i+1,j-1) + 2 if S[i] == S[j]
= max(L(i,j-1),L(i+1,j)) otherwise

If i=j the substring is a single character and the length of the longest palindromic subsequence is 1. For i-j=1, if S[i] == S[j], the palindromic subsequence is of length, Therefore L(i,j)=2.

For the remaining cases, the recurrence holds because if the first and the last character are the same, then we can say that it length of that substring is 2 + L(i+1,j-1). Therefore, these

characters can be included in the optimal solution and we get L(i,j)=2 + L(i+1,j-1). If the characters are different at most one of them will be part of the optimal solution and we obtain the length of the optimal solution via max(L(i,j-1),L(i+1,j)).

b) (6 points) Describe an algorithm that uses dynamic programming to solve the MPSP. The running time of your algorithm should be $O(n^2)$.

Pseudocode:

```
MPSP(s){
      // s is the string

      n = len(s)
      if n == 0:
            return 0

      // L is n*n 2d matrix for memorization.

      L = int[n][n]

      // Setting the diagonals to 1.

      for i <- 1 to n:
            L[i][i] = 1

      // Main Logic to calculate longest palindromic subsequence length.

      for currLength <- 2 to n:
            for i <- 1 to n-currLength + 1:
                  j = i + currLength - 1
                  if i - j == 1 and s[i] == s[j]:
                        L[i][j] = 2
                  else:
                        if s[i] == s[j]:
                              L[i][j] = 2 + L[i+1][j-1]
                        else:
                              L[i][j] = max(L[i+1][j], L[i][j-1])


      return L[1][n]
}
```

Textual Description: The algorithm tries to find the longest palindromic subsequence in the given string s. It calculates the length of the longest palindrome for every substring of s. However, it calculates the solution for each substring only once. It remembers the result by storing it in a n*n 2d matrix. Every cell[i,j] in the matrix denotes the maximum palindromic length of the substring of s which starts at i^{th} character and ends at j^{th} character. Hence, the

given matrix becomes an upper diagonal matrix. The calculation of the given cell is performed using the optimal substructure as described above.

Example:
Let string be “LPASPAL”

So, we will calculate the matrix L for every value of i,j
For i =j, L[i][j] = 1

For i = 1 and j = 2,
s[i] = ‘L’, s[j] = ‘P’
s[i] != s[j], Hence, L[i][j] = max(L[i][j-1], L[i+1][j]) = max(L[1][1],L[2][2]) = 1
.
.
.
.
.
.
.

For i = 2 and j = 5,
s[i] = ‘P’, s[j] = ‘P’
s[i] == s[j], Hence, L[i][j] = L[i+1][j-1] + 2 = L[3][4] + 2 = 1 + 2 = 3

Similarly it computes for the rest of the combinations across the table.

We will calculate for each values and get a table like this:

i↓ j→	L	P	A	S	P	A	L
L	1	1	1	1	3	3	5
P		1	1	1	3	3	3
A			1	1	1	3	3
S				1	1	1	1
P					1	1	1
A						1	1
L							1

Hence, the answer is 5.

Proof of correctness:

The base cases are strings of length 0, 1 and 2. If length is 0, algorithm returns 0. For length 1, it is 1. For length 2, if both the characters are same, the length is 2, else it will be max of (1,1) and (2,2) which both are of length 1, so it would be 1. Hence, it is correct.

The problem follows optimal substructure property. We calculated the answer of string of length n from the values of substrings which are of length n-1(unequal) or n-2(if the starting and ending characters of string of length n are equal) and so on. This is done by looking up the answer in the matrix L and getting the values of required substrings.

Time and space complexity:

Initially there is one for loop for initializing the diagonal elements to 1, which runs in O(n). Additionally, there are two nested for loops for filling the matrix. These loops run in the order of n each. Hence, runtime for these loops is O(n^2). Hence the **time complexity of the algorithm is O(n^2)**.

The algorithm requires additional space for matrix L which is n*n matrix. Hence, the **space complexity is O(n^2)**.

4) Practice Greedy Algorithms.

Assume that the points are sorted so that $x_1 < x_2 < \ldots < x_n$.

Greedy choice: The choice is to put a tower that serves x_1 , i.e. a tower at a point y in the interval [x_1 - d, x_1 + d] will reduce the problem to a subproblem that find towers to cover all the points x_1, ..., x_n, where x_1 , ..., $x_{i-1} \in [y - d, y + d]$.

Pseudocode:

```
Input maxDistanceAllowed: d, distances[] = {d1, d2, d3, d4, d5}(distances
in sorted order);

countTowers(distances)
{
      // Base case to handle invalid input which contains no
      // houses.
      if(distances.length < 0){
            print("Invalid input. No houses present in the street");
```

```
            return;
        }

        towerCount = 0;
        towerRange = {0, 0};

        // Main logic to count minimal number of towers.
        for(i = 0; i < distances.length; i++){
            if(distances[i] > towerRange[0] and distances[i] <
towerRange[1]){
                continue;
            }
            else{
                towerRange = {distances[i], distances[i] + 2d};
            }
            towerCount += 1;
        }
        print(towerCount);
}
```

Algorithm and analysis: The greedy choice is to put the first tower at position $x_1 - d$ Suppose x_1, ..., x_{i-1} are covered by the first tower, then finding towers to cover the remaining points x_i, ..., x_n becomes a subproblem. Iterate the same procedure for the remaining points x_i, ..., x_n until all houses are covered.

Example: For example, there are 6 houses x_1, x_2, x_3, x_4, x_5, x_6 based on the algorithm, the first tower is $y_1 = x_1 + d$, and y_1 can cover all the houses in the range of $[x_1, x_1 + 2d]$. Let us assume that x_1, x_2, x_3 fall in that range. For the remaining houses, we pick the second tower $y_2 = x_4 + d$, and y_2 will cover the range, $[x_4, x_4 + 2d]$ let's say that $x_4 <= x_5 <= x_6 <= x_4 + 2d$ so x_4, x_5 *and* x_6 can be covered by the second tower. So y_1 and y_2 are our solution.

Proof and the correctness of the algorithm:
The greedy choice will result in an optimal solution based on the following argument. Assume greedy does not always result in an optimal solution. Choose a counter example with minimum number of towns. Let y^* be the position of the first tower in an optimal solution. If $y^* = x_1 + d$, we are done, since for the reduced subproblem, due to our assumption of a minimum counter example, greedy will result in an optimal solution; if not, it cannot be the case that $y^* > x_1 + d$, otherwise the house at x_1 would not be covered by any tower. So if $y^* < x_1 + d$, the reduced subproblem for the optimal choice would be one with houses at position $x_1,.....,x_n$ where $x_i < y^* + d$ for $1 \leq i \leq j - 1$. This subproblem has at least as many houses as the one resulting from the greedy choice and therefore requires at least many towers, again using the assumption that greedy will produce an optimal solution for the reduced subproblem due to our assumption that we start from a counter example of minimum size. So the optimum choice leads to a solution with at least as many towers as the greedy one.

The time complexity of the algorithm:
During the execution of the algorithm we scan the sorted list of houses from left to right to

identify the first house x_j that is not covered by the tower that was selected last. In particular, we compare the position of the current house x_j with the position t_{last} of the tower that was selected last. If $x_j - t_{last} > d$ the house is the anchor point for the next tower at position x_j+d, and the scan proceeds until no further uncovered houses exist. Since there are n houses, and during the scan, each house is compared only once to a tower location, and might trigger at most one new tower, the algorithm runs in O(n) worst case time.