Question 1

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

1. (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.

   Answer:

```
global variable series ← [ ]
def getCoins(num):
  if num = 0 then
    return series
  end if
  if num > 50 then
    series ← append 50
    return getCoins(num-50)
  else if num > 25 then
    series ← append 25
    return getCoins(num-25)
  else if num > 10 then
    series ← append 10
    return getCoins(num-10)
  else if num > 5 then
    series ← append 5
    return getCoins(num-5)
  else if num > 1 then
    series ← append 1
    return getCoins(num-1)
  else
    exit the function
  end if
```

   Let's consider an example to explain this. For a value of 70, it recursively checks it is greater than 50, if not, we check if it is greater than the next highest cent denomination. For 70, the highest cent value is 50, so the remaining value is 20. Now we check the highest denomination is, we see that the denomination is 10, thus we decrease 10 from the remaining value and recursively call the function with the new value. We see that the new value is 10. And after the recursive function for 10 is called, it can be seen that the function exits.

2. (4 points) Write an O(1) (non-recursive!) algorithm to compute the number of returned coins.
   Answer:

```
quotient ← 0
if num > 50 then
    remainder ← num % 50
    quotient += (num - remainder)/50
    num ← remainder
end if
if num > 25 then
    remainder ← num % 25
    quotient += (num - remainder)/25
    num ← remainder
end if
if num > 10 then
    remainder ← num % 10
    quotient += (num - remainder)/10
    num ← remainder
end if
if num > 5 then
    remainder ← num % 5
    quotient += (num - remainder)/5
    num ← remainder
else
    quotient += num
end if
return quotient
```

For an iterative algorithm, we firstly check if the given value is greater than the highest denomination value. If that is true, we check the number of coins in that denomination that add up to a value less than the given value. i.e for a value for 520, we have 10 - 50 cent coins that add up to 500 and the first loop in the iterative function saves a value of 10, i.e 10 coins used. Now we move to the next if loop. The quotient variable keeps addding the number of coins used in each loop. Finally, when the iterative function ends, it can be seen that the quotient variable returns the number of coins.

3. (1 point) 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.

   Answer:
   Since a greedy algorithm always takes the highest possible denominations and then checks the next highest possible denomination in the remainder, when the given denominations are not the factors of the highest given denomination, there may be faulty answers.
   i.e, for the case where the denominations are 1, 6 and 10 cents, let us consider the example where the denominations are calculated for a value of 12. According to greedy algorithm, since $12 > 10$, it considers 10 as one of the output denominations. Thus, when the greedy approach is followed, for a value of 12, coins 10, 1 and 1 cent are in the output representation (3 coins). However, it can be seen that two 6 cent coins are enough for 12. Thus, greedy algorithms does not always give the minimum number of coins.

4. (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 in increasing order.

   Answer:

```
def NumberofCoins(self, denominations, n)
  m ← len(denominaitons)
  Initialize table matrix of size m*n to 0
  for i in range(m)) do
    for j in range(n + 1) do
      if i > 0 then
        if j ≥ arr[i] then
          table[i][j] ← min(table[i - 1][j], table[i][j - arr[i]] + 1)
        else
          table[i][j] ← table[i - 1][j]
        end if
      else
        table[i][j] ← j
      end if
    end for
  end for
  return table[-1][-1]
```

Explanation:
Initially, a two-dimensional array is initialized for a size of n and number of denominations. Now we iterate through the matrix and assign values to the cells. For the denomination of 1 cent, number of coins required is the number itself. However, for higher denominations, we compare the number of coins previously taken along with the number for the current denomination and we take the minimum of the two numbers.

Proving the correctness of the algorithm:
Initialization: Table is initialized with 0 since no coin is chosen yet and the coin value is 0.
Maintainence: When the first coin denomination is considered, if the denomination is 1, the least number of possible coins would be the value itself. If the value is not 1, it takes the min of the number of coins needed for previous denomination and curr denomination. This in turn gives the minimum number of coins needed for a given n value.
Termination: At this step, the values for all the denominations are filled with the minimum number of coins needed and the optimal number of coins is shown at the final element of the matrix.

Example:
Let us consider the value to be 12. For a given set of denominations to be [1, 6, 10], we get the following matrix representation and the minimum number of coins required would be the last element in the matrix.

1	0	1	2	3	4	5	6	7	8	9	10	11	12
6	0	1	2	3	4	5	1	2	3	4	5	6	2
10	0	1	2	3	4	5	1	2	3	4	1	2	2

The last element here is 2. i.e, we can use a minimum of 2 coins to obtain the value from the given list of denominations.

5. (6 points) Implement the algorithm described in d). The code framework are given in the zip file: framework.zip. To avoid loss of marks please make sure that all provided test cases pass on remote-linux server by using the test file. Instructions for setting up remote-linux server and testing are given in the document HW3 Programming Assignment Setup.pdf

   Answer:
   A python script implementing the above algorithm has been submitted along with this document.

Question 2

(10 points) In class we showed that multiplying two matrices

1. (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.
   Answer:

i\j	1	2	3	4
1	0	2500	3500	7000
2	x	0	1250	5250
3	x	x	0	1500
4	x	x	x	0

i\j	1	2	3	4
1	0	1	2	2
2	x	0	2	2
3	x	x	0	3
4	x	x	x	0

   m[1,2]
   (k = 1) = m[1,1] + m[2,2] + d0*d1*d2 = 2500
   m[2,3]
   (k = 2) = m[2,2] + m[3,3] + d1*d2*d3 = 1250
   m[1,3]
   (k = 1) = m[1,1] + m[2,3] + d0*d1*d3 = 6250
   (k = 2) = m[1,2] + m[3,3] + d0*d2*d3 = 3500
   m[3,4]
   (k = 3) = m[3,3] + m[4,4] + d2*d3*d4 = 1500
   m[2,4]
   (k = 2) = m[2,2] + m[3,4] + d1*d2*d4 = 5250
   (k = 3) = m[2,3] + m[4,4] + d1*d3*d4 = 8750
   m[1,4]
   (k = 1) = m[1,1] + m[2,4] + d0*d1*d4 = 20250
   (k = 2) = m[1,2] + m[3,4] + d0*d2*d4 = 7000
   (k = 3) = m[1,3] + m[4,4] + d0*d3*d4 = 9500

2. (1 point) What is the minimum number of scalar multiplications required to compute A1* A2 *A3 *A4?
   Answer:
   From the above explanation, we consider the minimum value obtained for m[1,4]. It can be seen that the minimum value is found when k = 2.
   m[1,4] (k = 2) = m[1,2] + m[3,4] + d0*d2*d4 = 7000
   Therefore, the minimum number of scalar multiplications is 7000.

3. (2 points) Give the optimal order of computing the matrix chain by fully parenthesizing the matrix chain below.

   Answer:
   Step 1:
   We check where the first parenthesis occurs for the given matrix chain from the k-table depicted above. It can be seen that m[1,4] splits at position 2.
   (A1 * A2) * (A3 * A4)
   Step 2:
   We check where the next parenthesis is present. For this, we consider m[1,2] and m[3,4]. It can be seen from the k-table that the parenthesis is done at positions 1 and 3 respectively making the representation of parenthesis as follows.
   ((A1) * A2) * ((A3) * A4)

4. (1 points) How many scalar multiplications are used to compute (((A1*A2) * A3) *A4)? Keep the order of matrix multiplications indicated by the brackets. Justify your solution.

   Answer:
   The number of scalar multiplications = 9,500
   From the given paranthesis, it is seen that the first split occurs at A3. This shows that for m[1,4] at k = 3, the split occurs.
   (k = 3) = m[1,3] + m[4,4] + d0*d3*d4 = 9500

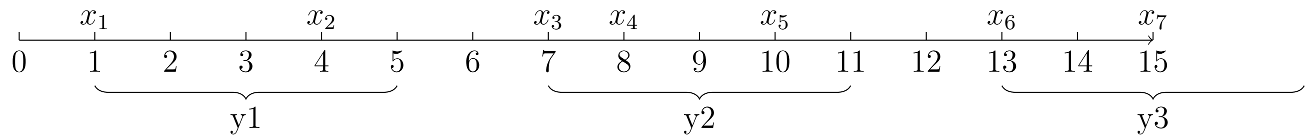
Question 3

Problem 3. Purpose: practice designing greedy algorithms. (10 points) Suppose you have a long straight country road with houses scattered at various points far away from each other. The residents all want cell phone service to reach their homes and we want to accomplish this by building as few cell phone towers as possible. More formally, think of points x1, . . . , xn, representing the houses, on the real line, and let d be the maximum distance from a cell phone tower that will still allow reasonable reception. The goal is to find a minimum number of points y1,. . . ,yk so that, for each i, there is at least one j with mod $yj - xi$ d. Describe a greedy algorithm for this problem. If the points are assumed to be sorted in increasing order your algorithm should run in time O(n). Be sure to describe the greedy choice and how it reduces your problem to a smaller instance. Prove that your algorithm is correct.

Answer:
Let $x_1, ...x_n$ be the houses on a long straight country road and d be the distance of coverage of each tower.
we can say that for a given range d, the greedy algorithm starts from the first house and places the tower as far as possible from the house such that it falls in the range of the tower. and the algorithm moves to the next house to see if it is in a given towers range or not. Finally, when it again encounters a house that is not in range, it again places the tower as far as possible from the house such that it falls in range of the tower.
Sample code

```
houses ← list of houses
towers ← initialize to 0 all house locations
towerRange ← tower range
count = 1
for house in houses do
  if towers[house] == 0 then
    for x in range(house, house + 2*towerRange) do
      towers[x] = count
    end for
  end if
  count += 1
end for
print(towers)
```

For example let d = 2units and
the list of houses that need signal be at positions [1, 4, 7, 8, 10, 13, 15] in a straight road.
We get the tower output as [0, 1, 1, 1, 1, 0, 0, 2, 2, 2, 2, 0, 0, 3, 3, 3, 3, 0] for house locations 0 to n. The tower range is depicted below.



Running time:
Since the algorithm iterates through the houses only once, we can say that the towers are placed in O(n) time.

We can also say that this greedy algorithm reduces the number of houses it is considering before setting up a tower to just one house and it places the tower independent of any other house in the straight road.
Thus reducing the problem size to one house instead of 'n' houses.

Correctness of the algorithm:
From the above problem statement, it is understood that each houses needs to get the tower service. From the greedy algorithm, we can say that each house has indeed got a tower service due to the strategic placement of the towers. Furthermore, we can also justify that there exists no more than one tower for each house and thereby making it an optimal solution. This is true because all houses in the straight line exists within a towers range and each house does not get signal from more than one tower. If there exists m towers placed for a given set of houses, we can argue that the number of towers placed by the greedy algorithm would be lesser than or equal to m as each house has at most only signal from one tower
Initialization: Initially, no house is considered thus zero towers are placed.
Maintenance: When each house is considered, it can be seen that there exists a tower placed that provides signal to each house placed till that instance thus proving that each house placed has at least one tower in range
Termination: At this point, all towers are placed and every house in consideration has a signal from at least one tower.

Question 4

Purpose: reinforce your understanding of data structures for disjoint sets. For background on binomial trees and binomial heaps please read 19-2 on page 527. (6 points) Please solve Problem 21.3-3 on page 572 of our textbook.

Answer:
Firstly, we need to perform 'n' MAKE-SET operations to create n singleton sets for the values x_1,..x_n. These singleton sets are depicted as $\{x_1\}$, $\{x_2\}$,...$\{x_n\}$

We know that to perform the union of values x_1 and x_2, we require one UNION operation. Thus, for the union of n values, we can say that we perform n-1 UNION operations and these operations are performed in $\log(n)$ time.

i.e n/2 of $UNION(x_1, x_2)$, $UNION(x_3, x_4)$,... $UNION(x_{n-1}, x_n)$ is performed, n/4 of $UNION(x_2, x_4)$, $UNION(x_6, x_8)$,... $UNION(x_{n-2}, x_n)$ is performed etc. until the final operation to perform the $UNION(x_{n/2}, x_n)$.

We have a total of n $MAKE-SET$ operations performed and a total of n-1 $UNION$ operations performed. Thus, for it to take a time of $\Omega(m \log n)$, we can say that m - (n - 1) - n = m - (2n - 1) $FIND-SET$ operations would be performed on the element present in the bottom of the tree.

Now, in order to get a cost of $\Omega(m \log n)$, we need to maximize the value of m and minimize (2n - 1). We can say this is true only when m > 2n - 1. Thus, when m takes the value of 3n or when m $\geq$ 3n, we can say that the total cost would be $\Omega(m \log n)$

Let us consider an example to justify the same
For a set of values x_1, x_2, x_3 and x_4
We first perform four MAKE-SET operations given by
$MAKE-SET(x_1), MAKE-SET(x_2), MAKE-SET(x_3), MAKE-SET(x_4)$
Now, 4-1 UNION operations are performed given by
$UNION(x_1, x_2), UNION(x_3, x_4), UNION(x_1, x_3)$
Finally, we perform the FIND-SET operation on the element at the bottom of the tree and it is given by $FIND-SET(x_4)$. Thus taking $\log(n)$ time where n = 4.