asatish2@ncsu.edu **asatish2** **Aakarsh Satish**

DAA HOMEWORK 3

1) **Purpose: Apply various algorithm design strategies to solve a problem, practice formulating and analyzing algorithms, and implement an algorithm. In the US, coins are minted in 50, 25,10, 5, and 1-cent denominations. 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 1cent.**

   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.

   Sol:

```
Function coins_change(n):
if n == 0:
        return []
Else if n >= 50:
        return [50] + coins_change(n - 50)
Else if n >= 25:
        return [25] + coins_change(n - 25)
Else if n >= 10:
        return [10] + coins_change(n - 10)
Else if n >= 5:
        return [5] + coins_change(n - 5)
else:
        return [1] + coins_change(n - 1)
```

Description of the algorithm:

1. The function coins_change takes an integer n as an input, which is the input given to calculate the minimum number of coin denominations for.
2. If n is equal to 0 then it is the base condition and it returns an empty list. This says that we have achieved the minimum number of coins and there are no coins left.

3. If n is greater than or equal to 50, the function returns a list containing [50]. It then recursively calls itself with the remaining amount by calling the function coins_change(n-50).
4. If n is less than 50 but greater than or equal to 25, the function returns an element [25] and recursively calls itself with the remaining amount by calling the function coins_change(n-25)..
5. If n is less than 25 but greater than or equal to 10, the function returns an element [10] and recursively calls itself with the remaining amount by calling the function coins_change(n-10).
6. If n is less than 10 but greater than or equal to 5, the function returns an element [5] and recursively calls itself with the remaining amount by calling the function coins_change(n-5).
7. If n is less than 5, it returns an element [1] and recursively calls itself with the remaining amount by calling the function coins_change(n-1).
8. The recursion continues until n becomes 0, at which point, the function returns an empty list, and thus ends the recursive call and starts backtracking.

Example Output:

1) Lets take the example of 17.
2) First the coins_change(n) will be called where n is the input. In our case n is17.
3) First it will check the base condition. In our case n is not equal to 0.
4) It checks the following 'IF' conditions. n is not greater than or equal to 50 and n is not greater than or equal to 25. It goes to the next IF condition.
5) It checks whether n is greater than or equal to 10 and it is a true condition.
6) Now it goes inside the if statement. The statement return an array element [10] + and then calls the function recursively again. The function coins_change(17-10) = coins_change(7) will be called in the next recursive call.
7) Now in the function coins_change(7), it repeats the same process.
8) Now the condition n is greater than or equal to 5. So it goes inside that if statement.
9) Now that function returns [5] + and calls the function again recursively calls coins_change(7-5) which is coins_change(2).
10) The [5] which is returned in the second call will ultuimately be inserted after [10] in the array when the program terminates.
11) Now coins_change(2) is executed and all the if statements turns out to be false, so now it executes the else statement.
12) Now it returns [1] and recursively calls coins_change(2-1) which is coins_change(1).

13) Again in the next recursive call coins_change(1), the if statements fails and the else condition is executed. Now it returns [1] and recursively calls coins_change(1-1) which is coins_change(0).
14) Now in the next call coins_change(0), the base condition satisfies and it just returns.
15) Now backtracking the whole recursive calls we get an array with the coin denominations which in our case will be [10,5,1,1].

Time Complexity:

The time complexity is dependent on the number of recursive calls, and it can be expressed as O(n), where 'n' is the initial value of n. In the worst case, the function will make O(n) recursive calls, and each call involves a constant amount of work, so the overall time complexity remains O(n).

Proof of Correctness:

Base case:
For base case when n is equal to 0, it returns an empty list showing that no changes i required.

Inductive hypothesis:
For the hypothesis, we can see that coins_change(k), (let k equal to remaining coins) generates a list of coin denominations for some k value which are less than n.

Inductive step:

- To prove that if the function performs correctly for k values, it must also be correct for n values.
- Considering when n is greater than 0, the function generates the largest available coin denomination and adds to the the list accordingly.
- The function then calls itself recursively with n - coin (coin that satisfies the condition).
- From the hypothesis we can say that the function returns the correct list of coin denominations for the available remaining number of coins.
- Hence it shows that the function performs correctly for k and it will be correct for n as it performs on smaller subproblems.

Termination:
The function terminates as and when the n keeps reducing after every call by either large coin denominations which is > 1 or by atleast 1. As it reduces by atleast 1 the function eventually reaches the base case and equals 0.

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

Sol:

```
Function constant_deno(amt)
        denom = [50,25,10,5,1]
        If amt < 0:
                Return 'Cant be solved'
        Result_coins =  0
        For i in denom:
                If amt >= i:
                        Count = amt // coin
                        Result_coins = Result_coins + Count
                        amt= amt - (Count*i)
        Return Result_coins
```

Time Complexity: This is a constant time algorithm as it is traversing through the already given array one time. So in our case it is BigO(5), which is a O(1) constant time.
In this algorithm the Result_coins will return the maximum number of coins that will be needed to match the amt given. If it is not possible it will return Invalid or in our case it "Cant be solved".

Simple Input and Output:
Assume the amount to be 12 and the denominations to be [50,25,10,5,1]
To calculate the minimum number of coins needed to make change for an amount of 12 with the denominations as given in the problem. We start with 12 and iterate.
First, we see if 12 is greater than 10. Since it is, we use a $10 coin and subtract 10 from the amount that leaves us with 2. Then, we check if 2 is greater than 1, which it is true, so we use two $1 coins and subtract 2 from the amount. We've used a total of three coins which are one $10 coin and two $1 coins to make change for 12, so the result returned is 3.

In case we do not find a value it returns 'Cant be solved'

Proof of Correctness:

Initisalisation: At the beginning of the loop, Result_coins is initialized to 0, which is the minimum number of coins needed to make change when no coins have been considered yet for the given amount.

Maintenance: During each iteration of the loop, the function checks if the current denomination i can be used to make change from the remaining amt. If amt is greater than or equal to i, it calculates the number of times the current denomination can be used and adds this to Result_coins. Later it subtracts Count * i from amt, updating amt to be changed. The loop maintains the loop invariant by correctly updating Result_coins and amt at each step or iteration.

Termination: At the end of the loop Result_coins returns the total number of coins that are needed to calculate the amount that is given.

c) (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.

Sol:

For this lets consider the denominations 1, 6 and 10 cents as given in the question. Consider an example:

Suppose we need to make change for 14 cents using the greedy algorithm approach. The greedy algorithm selects the largest denomination first in this case and then starts choosing 1's.

So if we perform the greedy approach we shall get the following denominations:

- First we select a 10 cent coin
- Second it selects 1 cent coin
- Third it selects 1 cent coin
- Fourthly it selects 1 cent coin and
- Again it selects 1 cent coin.

Clearly this is not the least number of coins which you can make for 14 cents.

The ideal answer would be two 6 cent coins and two 1 cent coins.

Using the greedy approach it gives us 5 coins as the least number of coins(10,1,1,1,1). But ideally it should be just 4 coins (6,6,1,1). This is more efficient solution than greedy approach which was proposed in this case.

Using two 6 cent coins and choosing two 1 cent coins is much better solution than choosing one 10 cent coin and four 1 cent coins.

This demonstrates that the greedy algorithm does not always give the minimum number of coins when the coin denominations are 1, 6, and 10 cents. In this case, the minimum number of coins is four, not five.

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.

Sol:

```
function dp_min_coins(coins, amt):
n = len(coins)
dp = [[float('inf')] * (amt + 1) for _ in range(n + 1)]
for i in range(n + 1):
        dp[i][0] = 0

for i in range(1, n + 1):
        for j in range(1, amt+ 1):
                if coins[i - 1] <= j:
                        include_current_coin = dp[i][j - coins[i - 1]] + 1
                        exclude_current_coin = dp[i - 1][j]
                        dp[i][j] = min(include_current_coin, exclude_current_coin)
                else:
                        dp[i][j] = dp[i - 1][j]
return dp[n][amt] if dp[n][amt] != float('inf') else 0
```

Algorithm description:

- First we shall initialise the variables where n is the length of coins which indicates the number of coin denominations and then we create an array dp which is ued to initialise our dynamic programming array with n + 1 and amt(= amount) + 1 rows and columns and we shall initialise with the maximum value in each.
- First we shall initialse the whole first column to be 0. This indicates that in order to make it to 0 amount no coins are needed to make it.
- Now it iterates through the 2 loops which we have mentioned. Here the i value contains coin denominations and j has the possible change in amounts.
- Now when it comes to the dp table, it checks the value of the current denominations coins[i-1] and compares it whether it is less than or equal to the current amount j. If the condition satisfies then we can make changes to the array and indicates the coin can be used to make change for the current amount.
- This requires us to calculate two conditions when the coin is included and when the coin is excluded.
- -> Including the current coin: To include the current coin we take the value present at j - coins[i - 1] and add 1 which indicates the coin has been included.

- -> Excluding the current coin: This can be found out in the row above at the same column dp[i - 1][j]
- Now the dp[i][j] will be the minimum of these two scenarios and updated accordingly.
- Finally we shall return the last element in the dp table which dp[n][amount]. This will contain the least number of coins required. If it is infinity it will return 0 which indicates the answer is not possible else will return the minimum number of coins.

Example Input and Output with Dp table:

Let us take an example where amount is 11 and let the denominations be:[2,4,5,7].
Now calculating the whole dp table according to the algorithm we shall be able to see a dp table like this:

	Amount	0	1	2	3	4	5	6	7	8	9	10	11
Coins													
0		0	inf	inf	inf	inf	inf	inf	inf	inf	inf	inf	inf
2		0	inf	1	inf	2	inf	3	inf	4	inf	5	inf
4		0	inf	1	inf	1	inf	2	inf	2	inf	3	inf
5		0	inf	1	inf	1	1	2	2	2	2	2	3
7		0	inf	1	inf	1	1	2	1	2	2	2	2

Now it return the last element in row and column which is 2.
Practically we can see that with denomination [2,4,5,7] to make an amount 11, we can use one 7 coin denomination and one 4 coin denomination.

Time Complexity Analysis:

In this algorithm inside the loops, we just perform the mathematical comparison and assigning operation which takes constant time. But the iteration over the rows and columns of coins and amount determines the time complexity of the problem.
We iterate through the whole outerloop of rows which is coins and inner loop of columns which is amount in our case. This takes Big O(number of coins * the amount) which is nothing but Big O(m*n) time complexity.

Proof of correctness:
Loop Invariant: For each i (coin) and j (amount), dp[i][j] represents the minimum number of coins required to make change for the amount j using the first i coins.

1) Initialization: At the start of the loopall the values of the array dp for 0 <= j <= amt are initialised to infinity except for the dp[0][0] element which is initialised to 0. The initialization step establishes the loop invariant for i =0.
2) Maintenance: When we iterate through the loop from i = 1 to n and j from 1 to amount(amt) we calculate dp[i][j] based on few conditions. We get to see two options: One is to include the current coin and another to exclude it.

- include_current_coin: We include the i-th coin and reduce the amount by coins[i-1]. In this case, the number of coins required is dp[i][j - coins[i-1]] + 1.
- exclude_current_coin: We exclude the i-th coin and keep the amount unchanged. In this case, the number of coins required is dp[i-1][j].

We choose the minimum of these two options and update dp[i][j] accordingly.

3) Termination: After the loop has completed the dp[n][amt] will contain the minimum number of coins required to make change for the given amount amt using all n coins.

Hence the loop invariant holds and is correct.

e) (6 points) Implement the algorithm described in d). The code framework is given in the zip file: CoinChange_framework.zip. To avoid loss of marks, please ensure that all provided test cases pass on the remote-linux server using the test file. Instructions for setting up remote-linux server and testing are given in the document HW3_ProgrammingAssignmentSetupInstructions.pdf.

```
CoinChange.py  CoinChangeTest.py  __pycache__
[asatish2@engr-ras-204 python]$ python3 CoinChangeTest.py -v
testcase1 (__main__.TestCases) ... ok
testcase2 (__main__.TestCases) ... ok
testcase3 (__main__.TestCases) ... ok
testcase4 (__main__.TestCases) ... ok
testcase5 (__main__.TestCases) ... ok

----------------------------------------------------------------------
Ran 5 tests in 0.139s

OK
```

In the programming file.

2) Problem 2. Purpose: practice algorithm design using dynamic programming. (10 points) A subsequence is palindromic if it is the same whether read left to right or right to left. For instance, the sequence A,C,G,T,G,T,C,A,A,A,A,T,C,G has many palindromic subsequences, including A,C,G,C,A and A,A,A,A (on the other hand, the subsequence A,C,T is not palindromic). Assume you are given a sequence of characters x[1…n]. Denote L(i,j) the length of the longest palindrome in x[i,...,j]. The goal of the Maximum Palindromic Subsequence Problem (MPSP) is to take a sequence x[1,...,n] and return the length of the longest palindromic subsequence L(1,n).

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

Sol:

Function longest_palindromic_subsequence(x, i, j):
    if i equals to j:
        return 1
    if i > j:
        return 0
    if x[i] is equals to x[j]:
        return longest_palindromic_subsequence(x, i + 1, j - 1) + 2
    return max(longest_palindromic_subsequence(x, i + 1, j),
longest_palindromic_subsequence(x, i, j - 1))

Description:

1) The function paliandromic_subsequence takes in x as the input string, and i and j as the index positions. (Starting and ending)

2) If the starting position equals to ending position, then that means that is the only character left and is a paliandrome with length 1.
3) If i > j, that means the starting index is now greater than the ending index, in this case it means that it is invalid and returns 0.
4) If x of ith position is equal to x of jth position that means that it is a paliandromic character and then it calls the function recursively again by increasing the ith index by 1 and decreasing the jth position by 1 and adds 2 with it. 2 is added because there are two characters matching and hence it is a paliandromic character.
5) If the characters do not match then recursively call the function by increasing i in one call by keeping the jth index the same and decrease the jth value by keeping the ith value same in another function call. The max value of these two call needs to be taken and returned.
6) The final value returned shall be the longest paliandromic subsequence.

Optimal Substructure:
To find the length of the longest palindromic subsequence in a sequence x[1...n], we can break it smaller subproblems.Consider a subsequence starting at position i and ending at position j, denoted as x[i...j]. If index i equals index j then that means it paliandrome with length 1. If ith position is greater than jth position that means it is invalid.If x[i] and x[j] are equal, then the length of the longest palindromic subsequence L(i, j) will be equal to the length of the longest palindromic subsequence in x[i+1...j-1] plus 2. But, if x[i] and x[j] are not equal, the length of the longest palindromic subsequence is the maximum of L(i+1, j) and L(i, j-1).

Recurrence Relationship:

Based on the optimal substructure, we can define the recurrence relationship for L(i, j) as follows:

L(i, j) = { 1, if i == j

2, if x[i] == x[j]

L(i+1, j-1) + 2, if x[i] == x[j] and i < j

max(L(i+1, j), L(i, j-1)), if x[i] != x[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).

Sol:
Function Pali_subseq(s):

```
n = take the length of s
for i from 0 to n-1:
dp[i][i] = 1
for i from n-2 down to 0:
        for j from i+1 to n-1:
                if s[i] == s[j]:
                        dp[i][j] = dp[i+1][j-1] + 2
                else:
                        dp[i][j] = max(dp[i][j-1], dp[i+1][j])

return dp[0][n-1]
```

Algorithm Description:

- First lets initialise the n value as the length of String s.
- Now lets initialse dp which is our array to 1 with rows and columns being i which is the diagonal elements. This means the character is paliandromic with itself.
- Now we shall iterate through the loop in the reverse from n-2 to 0 and the inner loop running from i+1 to n-1.
- Then we shall check if character at s[ith index] is the same as s[jth index]. If they are the same then it means the 2 characters will be part of the paliandromic subsequenece. So now we initialise the dp[i][j] it as 2 + the dp[i+1][j-1].
- If its not true then we initialise the maximum value of the dp[i][j-1] and dp[i+1][j].
- This loop continues and the same procedure goes on.
- After the loop ends we return the value dp[0][n-1] which will have the length of the longest common subsequence.

Time Complexity:

1. The algorithm uses a nested loop structure: an outer loop iterating from n-2 down to 0 (for i) and an inner loop iterating from i+1 to n-1 (for j).
2. For each pair of characters at indices (i, j), the algorithm computes the value of dp[i][j] using constant time operations which include comparisons, additions etc.
3. The nested loops iterates through all the possible pairs of characters in the string, so that every possible substring is considered and evaluated.
4. As the algorithm runs through two loops its time complexity is Big O(n*n) which is O(n^2).

Example Input and Output:

Lets take the example input string s as yyyxy

For this example the output should be 4 as the paliandromic subsequence would be yyyy and it should return the length of it as 4. By performing the algorithm we get the dynamic program table like this:

1	2	3	3	4
0	1	2	2	3
0	0	1	1	3
0	0	0	1	1
0	0	0	0	1

It returns 4 as expected. In our case it is in dp[0][n-1]

Proof of correctness:

Loop Invariant: For each iteration of the inner loop, the dynamic programming table calculates and stores the length of the longest palindromic subsequence in the subproblem of the substring which is generated.

Initialisation: At the beginning dp table is updated based on one of 2 conditions:
If s[i] == s[j] then dp[i][j] will be dp[i+1][j-1] + 2 as it indicates that those 2 characters can be part of the paliandromic subsequence.
If they are not equal then it is intialised to the maximum value of dp[i][j-1] and dp[i+1][j]
The diagonal elements in the table are initialised to 1 as the character is a paliandrome with itself.

Maintenance: In every loop iteration the dp table is updated accordingly based on those conditions and maintains the table.

Termination: When the loop finishes dp[0][n-1] will hold the number of the longest paliandromic subsequence in the entire string.

Hence with this prood, we can say that the algorithm is correct.

3) Problem 3. Purpose: practice designing greedy algorithms. (10 points) Suppose you want dinner at a restaurant on the road from Raleigh to Chapel Hill. You start in Raleigh. Your car has a full gas tank with 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 therestaurant is next to gas station n. Your goal is to make as few as possible stops along theWay.
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.

Sol:

```
GasStops(d, gas_stations):
        stops = []
        current_position = 0
        remaining_gas = d
        for gas_station in gas_stations:
                distance_to_station = gas_station - current_position
                if distance_to_station <= remaining_gas
                        remaining_gas - = distance_to_station
                else
                        stops.append(current_position)
                        remaining_gas = d - distance_to_station
                current_position = gas_station
        return stops
```

Description:

1) Let us assume that the current_position is 0 (Raleigh) as mentioned in the question
2) At first the remaining gas will be full tank with d in it.
3) In the for loop first we ll calculate the distance_to_station from the current position.
4) If the distance to the next station is less than or equal to remaining gas left, we subtract the distance travelled from the remaining gas and update the remaining gas.
5) Else if the remaining gas is not sufficient to travel to the next stop, we need to append the stop into an array and update the remaining gas variable.
6) Finally we update the current position.

7) Once the loop ends we shall have our final answer.

Input and Output Example:

Let us take an example where d = 80 and gas_stations be an array with distances = [50,80,120,160]

1) First the current position is initialised to be 0 and the remaining_gas = d which is eual to 80 in our case
2) Now let us iterate through the gas_stations array and let each element be named gas_station
3) First each distance_to_station is calculated which is gas_station - current_position, in our case first it will be distance_to_station = 50 - 0 = 50
4) We can see that the first if condition satisfies (50<=80), So now we do not need to make a stop and the remaining_gas will be remaining_gas - distance_to_station which is 80 - 50 = 30
5) Now we iniitialise the current position to be 50 which is the distance travelled thus far or which we are currently at.
6) Now iterating again the gas_station will be 80 and distance_to_station will be gas_station - current_psoition which is 80-50 = 30
7) Again the if condition satisfies as remaing_gas is greater than or equal to distance_to_station. (30 <=30)
8) Now calculating remaining gas it will be 0 and the current_position is updated to 80.
9) Iterating again now gas_station is 120.
10) Now calculating distance_to_station, it will be 120 - 80 which is equal to 40.
11) Now checking the ‘if’ condition, the remaining gas which is 0 will treat this condition as false.
12) Now consider the else statement, we append 80 to the stops array and calculate the remaining_gas which will be 80 - 40 = 40 and change the current_position to 120.
13) Iterating for the last time, the distance_to_station will be 160 - 120 = 40.
14) Now checking the If condition we see that remaining_gas is greater or equal to distance_to_station (40<=40), so we calculate remaining_gas = 40-40 = 0.
15) Finally the loop ends by return the stops as [80].

There is only one stop required to complete the whole journey at stop 80.

Time Complexity Analysis:

Inside each loop, the program does only a constant amount of work by initialising and updating the values of distance_tostation, stops etc. These are only updates and comparisons inside each iteration of the loop.
The whole loop runs till the size of the gas_stations array and iterates through each element only once.So the time complexity of this algorithm in worst case will be big O(n) where n is the number of gas_stations.

Proof of correctness:

Loop Invariant: At the beginning of the the loop, the stops array contains no stops or the current stops ensuring that the maximum distance that can be travelled between each gas station by checking the remaining gas variable.

Initialisation: Before the loop starts the stops array is an empty array which is true in this case because no other gas station has been travelled yet. The remaining gas is also maximum which is ‘d’ in our case before the start of the loop.

Maintenance: For every loop it checks the if condition. Whether the next statio could be travelled with the remaining gass left in the tank. If it is true then the next station could be travelled without stopping at any point and the remaining gas is calculated accordingly thus maximising the distance. But if the next station is further than the remaining gas then the else condition is satisfied and it updates the stops array and calculates the remaining gas accordingly.

Termination: The loop terminates when all the gas stations are visited and the stops array correctly contains the gas stations where you need to stop to reach the next destination in the least amount of stops.

With this proof the proof of correctness we can say that the algorithm is correct.