e) You will be given a list coin denomination that you can use to tender change to your customers, find the most optimum way to tender the exact change to your customers, the optimum is when you use the least number of coins

Example:

Input : [1,2,5,8,1] (Available coins)

Input : 7 (Change to be given)

Ans : [2,5]

Explain all the scenarios in better words and simpler to understand format compared to explanation available in the link below:

https://www.geeksforgeeks.org/coin-change-dp-7/

Solution:

```
def coinChange(coins, amount):
    # Initialize the dynamic programming table with a default value of infinity
    dp = [float('inf')] * (amount + 1)
    dp[0] = 0

    # Iterate through each coin denomination
    for coin in coins:
        # Update the dp table for each value from the coin denomination to the desired amount
        for i in range(coin, amount + 1):
            # Choose the minimum between using the coin and not using the coin
            dp[i] = min(dp[i], dp[i - coin] + 1)

    # Return the minimum number of coins required for the desired amount
    return dp[amount]
```

Explanation:

The bottom-up dynamic programming methodology is used in the code.

1. Create a default value of infinity in the dynamic programming table dp for each amount up to the appropriate quantity (amount + 1).
2. The minimal number of coins needed to equal each value will be kept in this table.
3. Set dp[0] to 0 to demonstrate that 0 coins are required to equal a certain amount.
4. Go through each coin denomination in the list of coins one more time.
5. Repeat the range from coin to amount + 1 for each coin denomination.
6. By selecting the minimum between the current value of dp[i] and dp[i - coin] + 1, the dp table is updated. This determines the bare minimum amount of coins required by contrasting using the current coin with not using it.

7. The least number of coins necessary for the requested amount is recorded in dp[amount] after iterating over all the coin denominations and updating the dp database.
8. Return the minimal number of coins required to equal the specified amount, dp[amount].
9. In conclusion, this solution successfully solves the coin-change problem by identifying the minimal number of coins required for a given amount using dynamic programming.

Question:

Explain what is a greedy algorithm and how dynamic programming helps in this case.

A greedy algorithm solves problems by making the locally optimal decision at each stage in the hopes that it would result in a globally optimal solution. In other words, it makes the best decision possible at the current time without taking into account probable repercussions at a later time.

On the other hand, dynamic programming is a method for resolving complex issues through the independent solution of overlapping subproblems. It keeps the answers to smaller problems in a table or array and reuses those answers to tackle bigger issues quickly.

A greedy algorithm would choose the greatest coin denomination available at each step until the target amount is attained in the context of the coin change problem in order to determine the most optimal way to tender change. A greedy strategy does not, however, necessarily result in the best outcome.

The concept of memoization is used in dynamic programming to offer an ideal solution in this situation. It divides the issue into more manageable subproblems, each of which it solves independently and records in a database. The algorithm determines the smallest number of coins necessary for each value up to the target amount by reusing these answers in order to avoid making duplicate calculations.

The method may identify the globally optimal solution, that is, the smallest number of coins needed to tender the precise change, thanks to the dynamic programming approach, which makes sure that all potential coin combinations are taken into account. It avoids the possibility of making decisions that are locally optimal but may not produce the best overall outcome.

In conclusion, dynamic programming complements the greedy algorithm by offering a methodical and effective approach to handle the coin-change problem optimally, ensuring that the precise change is presented with the fewest possible coins.

Bonus question:

given a number N, remove one digit and print the largest possible number.

E.g.

5-

1234

2945

9273

3954

19374

Answers:

234

945

973

954

9374

Why is the above solution part of a greedy algorithm?

Solution:

```
def remove_one_digit_greedy(num):
    # Convert the number to a string for easier manipulation
    num_str = str(num)

    # Initialize the maximum number with the input number
    max_num = num

    # Iterate through each digit of the number
    for i in range(len(num_str)):
        # Remove the current digit from the number
        new_num_str = num_str[:i] + num_str[i+1:]

        # Convert the modified string back to an integer
        new_num = int(new_num_str)

        # Update the maximum number if the new number is greater
        if new_num > max_num:
            max_num = new_num

    # Return the largest number after removing one digit
    return max_num

# Get input from the user
num = int(input("Enter a number: "))

# Call the function to find the largest number
largest_num = remove_one_digit_greedy(num)

# Print the result
print("Largest number after removing one digit from", num, "is:", largest_num)
```

Explanation:

1. The function remove_one_digit_greedy takes an input number num as an argument.
2. The number is converted to a string num_str to allow individual digit manipulation.
3. The variable max_num is initialized with the input number, assuming it to be the largest number initially.
4. A loop is used to iterate through each digit of the number. The loop variable i represents the index of the current digit.
5. Inside the loop, the current digit is removed from the number by slicing the string num_str and concatenating the parts before and after the digit.
6. The modified string new_num_str is converted back to an integer new_num.
7. The condition if new_num > max_num checks if the new number obtained by removing the current digit is greater than the current maximum number. If it is, the maximum number is updated with the new number.
8. After iterating through all the digits, the largest number obtained after removing one digit is stored in max_num.
9. The result is printed with a message indicating the largest number after removing one digit from the input number.

By repeatedly eliminating each digit and selecting the greatest resultant number at each step, this code implements a **greedy algorithm**. It seeks to discover the greatest number after eliminating one digit by choosing the digit that contributes the most to the overall value.