```
1)

def coin(coins,amount,output):

  if (amount==0 or amount<0):
      return output
  i=0
  x=0
  for i in coins:
      if(i<=amount):
          x=i
          break
  if(i==len(coins)+1):
      return -1
  output.append(x)
  return coin(coins, amount-x, output)
```

Textual Description: First, we check if the amount is 0 or <0. Negative is invalid and zero means we have all the required denominations. Then we search for the largest denomination which is less than or equal to the amount. That denomination is put in output and subtracted from amount and count is updated. Now a recursive call is done in which taken in which a new function is called where denomination is subtracted from amount. If no denomination is smaller than the amount then we return -1 which means we can't give the change for it.

Worked Out Example: Let's assume we have a coin= [10,5,1] and we have amount of 21. Then in the first recursive call. We can see that largest denomination less than amount 21 is 10. Hence, we subtract 10 from amount and add 10 to output. Then we put a second recursive call-in which amount will be 11 and then we again subtract 10 from it and add it to the output. In the last we only have 1 in amount remaining. In the third recursive call as we run the for loop, we find the largest denomination which is greater than equal to 1 is 1. Hence, we add 1 to output. And subtract 1 from amount. Now our amount is 0 hence we return the output.

```
def coin(coins,amount,output):

  if (amount==0 or amount<0):                    ……1
      return output
  i=0
  x=0
  for i in coins:……number of denominations
      if(i<=amount):
          x=i
          break
  if(i==len(coins)+1):……..1
      return -1
  output.append(x)……..1
  return coin(coins, amount-x,output)
```

T(n)= {0 if amount=0}
{T(n-1) +3+noofdenominations}

T(n) = T(n-1) + 3+2
T(n) = (T(n-2) + 3) + no of denominations
Let's assume no of denominations=2
T(n) = (T(n-2) + 5) +5
T(n) = T(n-k) + k*5
Let n-k=0
T(n) = T (0) + 5n
T(n) €O(n)

b)
Algorithm:

```
coin_count( amount, denomination):
        coin_amount=0
        for coin in denomination:
                if(coin<=value):
                        coin_count= coin_count + amount//coin
                        value=value%coin
        return coin_count
```

Code Explanation:

1) Value=17 and denomination=[10,5,1]. We loop over denominations. Hence iterate over its denominations and we take the largest denomination which is less than amount. Which is in this case 10. So, we divide amount the amount by largest denomination. Then add it too coin_count.
2) Then take modulus of amount by largest denomination. Hence 17//10 is 1. 17%10 is 7. In next iteration we find the largest denomination less than or equal to 7 which is 5 in this case. Coin_count will get increased by 1 as 7//5.
3) Amount gets updated to 2 as 7%5. It is the next iteration the largest denomination which is 1 in this case. 2//1 is 2. hence the coin_count gets updated by 2. Now the amount is 2%1 which is 0. Our final coin_count is 4.

c) Denominations=[1,6,10] and amount= 18 . In these if we apply the algorithm in b) we will get coin_count=4 or denominations that will make the change will be [10,6,1,1]. But it is not true as the minimum number of coins is [6,6,6] for these.

d)
```
def no_of_coins(coins, amount):
  table = [float('inf') for i in range(amount+1)]
  table[0] = 0
  for i in range(len(table)):
    for j in coins:
      if i-j >= 0:
        table[i] = min(table[i], table[i-j]+1)
  return table[-1]
```

An array of amount+1 is made. This array's indexes range from 0 to Amount. Since 0 cents require no coins, we fill the array's 0th index with 0. We put in the remaining cells. Since the coins in "C" are sorted, we begin by selecting the coin with the lowest denomination from index=1 because, as was mentioned previously, index=0 will always be 0. Either you use the coin for change or you don't use the coin for change; those are your two options. If you use the coin as change, you will receive one change (for this particular coin) plus however many changes are necessary for the new amount (amount-coin). You just put in the current value if you don't use the currency.

```
def no_of_coins(coins, amount):
  table = [float('inf') for i in range(amount+1)] ……n
  table[0] = 0
  for i in range(len(table)):……..n
    for j in coins:……n*d
      if i-j >= 0:
        table[i] = min(table[i], table[i-j]+1)……n*d
  return table[-1]
```

T(n)= O (2n*d+2n0 € O(n*d)
T(n) = O(n*d)

Where n=amount and d= number of denominations

Proof of Correctness:

Loop Invariant: At the beginning of each iteration array table contains minimum number of coins required for change at the first non-infinite index.

Initialization**:** At start, the table [0] contains 0 as there are no coins needed to change an amount 0. This is trivial and correct

Maintenance: Each time, we fill the table array at index "i" with a coin from the coins array, which are arranged in ascending order. There are always two options: either we utilize the coin to create change (in which case the number of coins would be 1 plus the number of coins

required for the amount minus the coin), or we do not (in which case the current value of the table index would be the minimal number of coins).

Termination: The loop terminates when all the coins have been seen. As we have extracted all the possible change starting from highest value to lowest value, we have the minimum number of changes required.

Solved Example:

Coins= [1,2,4] amount=7

Index 0	Index 1	Index 2	Index 3	Index 4	Index 5	Index 6	Index 7
0	1	1	2	1	2	2	3

As, we can on the 0^{th} index we can make we need o coins. On the 1^{st} index we need 1 coin of 1 cent, On the index 2 we need one coin of 2 cents. On the index three the min number of coins needed to make 3 are 2 which are (2,1). In the index 4 we need one 4 cent coin and index 5 and index 6 we need coins which are [4,1] and [4,2] respectively. On the index 7 we can see that it has updated to 3. table[7] = min(table[7], table[7-2]+1).

2)

a) Let L(i,j) be the longest palindrome subsequence and x[1,…..n] be the sequence. Let i=0 and j=len(x)-1 or the last index of x

If the i^{th} and j^{th} character of the sequence are same then L[i,j]= 2+ L[i+1,j-1]

Else if there are not same then L[i,j] will be maximum of L[i+1,j] and L[i,j-1]

Recurrence Equation:

Then L(i,j)= 1 ……..if i=j
2 ……. if i+1=j and x[i]= y[j]
2+ L[i+1, j-1] ……..if x[i]==y[j]
max (L[i+1, j] , L[i,j-1]) ……else

First Case: if there is a single element in palindrome sequence or i=j returns 1
Second Case: if there are two elements in palindrome sequence or i+1=j then returns 2
Third Case: If first and last elements of palindrome are equal then return 2+ L[i+1, j-1]
Fourth Case: If the two elements are not equal then return the maximum of max (L[i+1, j] , L[i,j-1])

b)

```
def longestPalindromeSub(x):

      if len(x) <= 1:
          return len(x)

      #Table Initialization
      memo= [[0] * len(x) for _ in range(len(x))]
      #Single characters are palindrome of length 1
      for i in range(len(x)):
          memo[i][i] = 1

      for i in range (2, len(x)+1):
          for k in range (0, len(x)-i+1):
              j=k+i-1

              if x[k] == x[j]:
                  # Both two elements are same
                  memo[k][j] = memo [k + 1] [j - 1] + 2
```

```
                else:
                    memo[k][j] = max (memo [k + 1] [j], memo[k] [j - 1])
        return memo [0] [-1]
```

Time Complexity:

```
def longestPalindromeSub(x):

        if len(x) <= 1:
            return len(x)                                   ……..1

        #Table Initialization
        memo= [[0] * len(x) for _ in range(len(x))] ……n^2
        #Single characters are palindrome of length 1
        for i in range(len(x)):                          ……. n
            memo[i][i] = 1

        for i in range (2, len(x)+1):……..n
            for k in range (0, len(x)-i+1):……..n^2
                j=k+i-1

                if x[k] == x[j]:
                    # Both two elements are same
                    memo[k][j] = memo [k + 1] [j - 1] + 2
                else:
                    memo[k][j] = max (memo [k + 1] [j], memo[k] [j - 1])
        return memo [0] [-1]
```

Time Complexity = $O(2n^2+n+1)$ € $O(n^2)$
Space Complexity = O(n2) {we are storing a 2D matrix of size n}

The correctness of the recurrence is the base for the proof for correctness of the algorithm.

Proof by Induction

Base Case: When the input string is of length 1, the longest palindrome subsequence is simply the string itself. This is true because any single character is a palindrome.

Induction Hypothesis: Assume that the algorithm correctly computes the longest palindrome subsequence for all input strings of length up to k

Inductive step: Consider an input string of length k + 1. We need to show that the algorithm correctly computes the longest palindrome subsequence of this string.

We can divide the problem into two cases:

Case 1: The first and last characters of the input string are the same. In this case, the longest palindrome subsequence can be obtained by removing the first and last characters of the input string and computing the longest palindrome subsequence of the remaining substring. This is because the longest palindrome subsequence of the substring is also a palindrome subsequence of the original string. The algorithm correctly computes the longest palindrome subsequence of the substring because it uses the inductive hypothesis. Therefore, the algorithm also correctly computes the longest palindrome subsequence of the original string.

Case 2: The first and last characters of the input string are different. In this case, the longest palindrome subsequence can be obtained by either computing the longest palindrome subsequence of the substring without the first character, or computing the longest palindrome subsequence of the substring without the last character, This is because the longest palindrome subsequence of any of these three substrings is also a palindrome subsequence of the original string.

Let string be “acbdcb”

So, we will calculate the matrix L for every value of i,j

For i =j, L[i][j] = 1

For k=0 and j = 1, s[k] = ‘a’, s[j] = ‘c’

So, s[k] != s[j], Hence, memo[k][j] = max (memo [k + 1] [j], memo[k] [j - 1]) = 1

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

i ↓ j→	a	c	b	d	c	b
a	1	1	1	1	3	3
c		1	1	1	3	3
b			1	1	1	3
d				1	1	1
c					1	1
b						1

3.

```
Minimum_Stops(gas_stops,d)
    start=0
    stops_required=0
    for i in range (1, len(gas_stops)):
        if station[i]-start > d:
          min_stops=min_stops+1
          start=gas_stops[i-1]
    return stops_required
```

We iterate through the loop through the station array, starting at 0. We verify that the car can travel to gas_stops[i] from its starting position without refuelling, and if not, we must stop at the gas_stops[i-1] to refuel and increase the number of stops required. Additionally, we update the start to gas_stop[i-1], as our jourey now resumes from gas_stops[i-1].

```
Minimum_Stops(gas_stops,d)
    start=0
    stops_required=0
    for i in range (1, len(gas_stops) ):……..n
        if station[i]-start > d:
          min_stops=min_stops+1
          start=gas_stops[i-1]
    return stops_required
```

Time Complexity Running complexity is O(n)
n is the number of gas stations

Greedy Choice: The greedy choice is to always stop at the farthest gas station that is within distance d of the current gas station. This is because this choice reduces the problem to a smaller instance, where the problem is to find the minimum number of stops needed to reach the restaurant from the farthest gas station that is within distance d of the current gas station.

Correctness of algorithm:

We are going to prove this by proof of induction:

Base Step:
If the gas_stops array is empty that means we are actually standing at the destination and no of stops required is 0 and function return 0 which is correct and our induction holds true

Induction step:
In order to prove that our algorithm correctly calculates the number of stops to reach n station we have to assume that the algorithm returns the correct value for stops required to reach n-1 station shall be Gn-1

Now when our car arrives at gas station g_n, we check the if condition
If g_n-start>d is false that means car can reach the destination g_n without stopping at the station and we reach to the end of loop and our function return gn-1 which is correct and our induction holds true
If Above condition is False that means the car cannot reach g_n and has to stop at g_{n-1} for refuel. Thus, we increment the gn-1 value by 1. As per our assumption gn-1 was correct that means 1+gn-1=gn, stops required to reach gn shall be correct and our induction holds true.
Hence using the method of induction, we thoroughly proof that our code
correctly calculates the minimum of number of gas stops required.