Problem 9: We wish to compute the laziest way to dial given n-digit number on a standard pushbutton telephone using two fingers. We assume that the two fingers start out on the star and hash keys, and that the effort required to move a finger from one button to another is proportional to the Euclidean distance between them. Design and analyze an efficient algorithm that computes in time of dialing that involves moving your fingers the smallest amount of total distance.

PSEUDOCODE :

Def distance(p1, p2):

Return Euclidean distance between p1 and p2

Def dialing(input_number):

total_dist, finger1_position, finger2_position

For each digit i in input_number:

Calculate dist1, dist2

If dist1 <= dist2:

total_dist += dist1
Update finger1_position

Else:

total_dist += dist2
Update finger2_position

Return total_dist

Input input_number

Print "Total distance moved:" followed by dialing(input_number)

CODE :

```
import math
keypad = {
    '1': (0, 0), '2': (0, 1), '3': (0, 2),
    '4': (1, 0), '5': (1, 1), '6': (1, 2),
    '7': (2, 0), '8': (2, 1), '9': (2, 2),
    '0': (3, 1), '*': (3, 0), '#': (3, 2)
}

def distance(p1, p2):
    return math.sqrt((p1[0] - p2[0])**2 + (p1[1] - p2[1])**2)

def dialing(input_number):
    total_dist = 0
    finger1_position = keypad['*']
    finger2_position = keypad['#']
```

```
    for i in input_number:
        i_position = keypad[i]

        dist1 = distance(finger1_position, i_position)
        dist2 = distance(finger2_position, i_position)

        if dist1 <= dist2:
            total_dist += dist1
            finger1_position = i_position
        else:
            total_dist += dist2
            finger2_position = i_position

    return total_dist

input_number = input()
print("Total distance moved:", dialing(input_number))
```

ALGORITHM ANALYSIS :
Time Complexity: O(n)

Problem 7:

PSEUDOCODE :

CODE :

ALGORITHM ANALYSIS :

Time Complexity: O(n^2)

Problem 10: There are N floors and N persons each one is tagged with some random unique number between 1 to N (represents floor number). We have a lift which can accommodate one person at a time. Every person is in some random floor. Initially lift is at floor 1 and all floors have single person. Design an algorithm to move the persons to their corresponding floor with minimum number of lift movements. [Restriction: Lift can have at most one person at a time. While moving persons, at some point of time, we can keep more than one person at one floor.]

PSEUDOCODE :

Def min_movement(N, persons):

Sort persons based on their tagged floor numbers

```
    movements = 0
    current_floor = 1
```

For each person in persons:

```
        movements += abs(current_floor - person[1])
```

```
        movements += abs(person[0] - person[1])

        current_floor = person[1]
```

Return movements

CODE :

```
def min_movement(N, persons):
    persons.sort(key=lambda x: x[1])
    movements = 0
    current_floor = 1

    for person in persons:
        movements += abs(current_floor - person[1])
        movements += abs(person[0] - person[1])
        current_floor = person[1]
    return movements

N = int(input('number of floor: '))
persons = []
for i in range(N):
    person_floor = int(input('person floor: '))
    persons.append((i+1, person_floor))
print(min_movement(N, persons))
```

ALGORITHM ANALYSIS :

Time Complexity: O(nlogn)

Assignment - 5 (Dynamic Programming)

Problem 1: Given 3 strings of all having length ¡ 100, write a program to find the longest common sub-sequence in all three given sequences.

PSEUDOCODE :

Function Lcs(s1, s2, s3, i, j, k, dict):

If i = 0 or j = 0 or k = 0:

Return // Base case: One of the strings is empty

If (i, j, k) is already in dict:

Return dict[(i, j, k)]

If s1[i - 1] = s2[j - 1] = s3[k - 1]:

lcs = Lcs(s1, s2, s3, i - 1, j - 1, k - 1, dict) + s1[i - 1]

dict[(i, j, k)] = lcs

Return lcs

Else:

lcs1 = Lcs(s1, s2, s3, i - 1, j, k, dict)

lcs2 = Lcs(s1, s2, s3, i, j - 1, k, dict)

lcs3 = Lcs(s1, s2, s3, i, j, k - 1, dict)

lcs = Longest of lcs1, lcs2, lcs3 // Find the longest subsequence

dict[(i, j, k)] = lcs

Return lcs

Function lcs(s1, s2, s3):

dict = Empty dictionary

Return Lcs(s1, s2, s3, Length of s1, Length of s2, Length of s3, dict)

CODE :

```
def Lcs(s1, s2, s3, i, j, k, dict):
    if i == 0 or j == 0 or k == 0:
        return ""

    if (i, j, k) in dict:
        return dict[(i, j, k)]

    if s1[i - 1] == s2[j - 1] == s3[k - 1]:
        lcs = Lcs(s1, s2, s3, i - 1, j - 1, k - 1, dict) + s1[i - 1]
```

```
            dict[(i, j, k)] = lcs
            return lcs
        else:
            lcs1 = Lcs(s1, s2, s3, i - 1, j, k, dict)
            lcs2 = Lcs(s1, s2, s3, i, j - 1, k, dict)
            lcs3 = Lcs(s1, s2, s3, i, j, k - 1, dict)
            lcs = max(lcs1, lcs2, lcs3, key=len)
            dict[(i, j, k)] = lcs
            return lcs

def lcs(s1, s2, s3):
    dict = {}
    return Lcs(s1, s2, s3, len(s1), len(s2), len(s3), dict)

s1 = input("first string: ")
s2 = input("second string: ")
s3 = input("third string: ")
print("Longest common subsequence:", lcs(s1, s2, s3))
```

ALGORITHM ANALYSIS :
Time Complexity: O(n^3)

Problem 3: 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). Devise an algorithm that takes a sequence x[1 : : : n] and returns the (length of the) longest palindromic subsequence. Its running time should be O(n^2).

PSEUDOCODE :

```
Function lps(S):
   R = ReverseString(S)
   length = Length(S)
   dp = initialize 2*2 matrix with 0
     for i from 1 to length:
      for j from 1 to length:
        if S[i - 1] == R[j - 1]:    If the characters at positions i and j are equal
            dp[i][j] = 1 + dp[i - 1][j - 1]
        else:
           dp[i][j] = max(dp[i][j - 1], dp[i - 1][j])

   return dp[length][length]
```

CODE :

```
def lps(S):
    R = S[::-1]
    length = len(S)
    dp = [[0] * (length + 1) for i in range(length + 1)]

    for i in range(1, length + 1):
        for j in range(1, length + 1):
            if S[i - 1] == R[j - 1]:
                dp[i][j] = 1 + dp[i - 1][j - 1]
            else:
                dp[i][j] = max(dp[i][j - 1], dp[i - 1][j])

    return dp[length][length]

s = input()
print(lps(s))
```

ALGORITHM ANALYSIS :

Time Complexity: O(n^2)

Problem 7: Write a program that calculates the highest sum of numbers passed on a route that starts at the top and ends somewhere on the base. n = 0 7 n = 1 3 , 8 n = 2 8 ,1, 0 n = 3 2, 7, 4, 4 n = 4 4, 5, 2, 6, 5 For the above figure shows a number triangle and its output is 30(7,3,8,7,5). Each step can go either diagonally down to the left or diagonally down to the right.

PSEUDOCODE :

high_sum(triangle):

n = length of triangle

dp = Last row of triangle

for row in range(n - 2, -1, -1):

// Iterate over elements in the current row

for i in range(length of triangle[row])):

dp[i] = triangle[row][i] + maximum of (dp[i], dp[i + 1])

print(dp)

// Return the maximum sum from the top of the triangle

return dp[0]

CODE :

```
def high_sum(triangle):

    n = len(triangle)

    dp = triangle[-1][:]

    for row in range(n - 2, -1, -1):

        for i in range(len(triangle[row])):

            dp[i] = triangle[row][i] + max(dp[i], dp[i + 1])

    print(dp)

    return dp[0]

triangle = []
n = int(input("Enter the number of rows: "))
print("Enter the elements:")
for i in range(n):
    row = list(map(int, input().split()))
    triangle.append(row)

result = high_sum(triangle)
print(" maximum sum :", result)
```

ALGORITHM ANALYSIS :

Time Complexity: O(n^2)

Problem 10: **In the art gallery guarding problem, a line L represents a long art gallery hallway. We are given a set of location points on it X=x0, x1, ..., xn−1 of real numbers that specify the positions of the paintings along the hallway. A single guard can guard paintings standing at most 1 meter from each painting on either side. Design an algorithm that finds the optimal number of guards to guard all the paintings along the hallway**.

PSEUDOCODE :

min_guards(locations):

- Sort the locations list
- Initialize a list dp with n elements, each set to positive infinity
- Set dp[0] to 1 (as the base case)
- For each location from the second one to the last:
- For each previous location:
- If the distance between the current location and the previous one is less than or equal to 1:
    - Update dp[i] to be the minimum of its current value and dp[j] + 1

- Return the last element of dp, which represents the optimal number of guards needed

CODE :

```
def min_guards(locations):
    locations.sort()
    n = len(locations)
    dp = [float('inf')] * n
    dp[0] = 1

    for i in range(1, n):
        for j in range(i):
            if locations[i] - locations[j] <= 1:
                dp[i] = min(dp[i], dp[j] + 1)

    return dp[-1]

n = int(input("Enter the number of location points: "))
locations = []
print("Enter the location points:")
for i in  range(n):
    location = float(input())
    locations.append(location)
print("Optimal number of guards:", min_guards(locations))
```

ALGORITHM ANALYSIS :

Time Complexity: O(n^2)

Problem 4 :A list of n positive integers a1; a2; :::; an; and a positive integer t is given. Write a program to find subset of the ai 's add up to t? (You can use each ai at most once.)

PSEUDOCODE :

Function subset_sum(nums, target):

n = length of nums

Initialize a 2D array dp with dimensions (n+1) x (target+1)

Set dp[i][0] = true for all i from 0 to n // Base case: subset with sum 0 exists for all subsets

Set dp[0][j] = false for all j from 1 to target // Base case: subset with non-zero sum doesn't exist for an empty set

for i from 1 to n:

for j from 1 to target:

if nums[i-1] <= j:

dp[i][j] = dp[i-1][j] or dp[i-1][j-nums[i-1]]

else:

dp[i][j] = dp[i-1][j]

Initialize an empty list subset

i = n, j = target

while i > 0 and j > 0:

if dp[i][j] and not dp[i-1][j]:

Append nums[i-1] to subset

j = j - nums[i-1]

Decrement i

Return subset

CODE :

```
def subset_sum(nums, target):
    n = len(nums)
    dp = [[False] * (target + 1) for i in range(n + 1)]

    for i in range(n + 1):
        dp[i][0] = True

    for i in range(1, n + 1):
        for j in range(1, target + 1):
            if nums[i - 1] <= j:
                dp[i][j] = dp[i - 1][j - nums[i - 1]] or dp[i - 1][j]
            else:
                dp[i][j] = dp[i - 1][j]

    subset = []
    i, j = n, target
    while i > 0 and j > 0:
        if dp[i][j] and not dp[i - 1][j]:
            subset.append(nums[i - 1])
            j -= nums[i - 1]
        i -= 1

    return subset[::-1]

nums = list(map(int, input('enter the list').split()))
target = int(input('enter the target'))
print("Subset with sum equal to", target, ":", subset_sum(nums, target))
```

ALGORITHM ANALYSIS :

Time Complexity: O(n^2)