Exp.01 | BFS(Breadth First Search), DFS | Date: 1/8/2024

Write a program to implement BFS and DFS.

DFS(Depth First Search):

AIM: To write a program to implement DFS using python

ALGORITHM:

1. Start
2. Start by putting any 1 of the graphs vertices on top of a stack.
3. Take the top item of stack and add it to the visited list.
4. Create a list of that vertex adjacent nodes. Add the one or not in visited list to top of stack
5. Keep repeating steps 2 and 3 until the stack is empty.
6. Stop

PROGRAM:

```
graph = {
  '5' : ['3', '7'],
  '3' : ['2', '4'],
  '7' : ['8'],
  '2' : [],
  '4' : ['8'],
  '8' : []
```

OUTPUT:

Following is the DFS

5

3

2

4

8

7

```
}
visited = set()
def dfs(visited, graph, node):
        if node not in visited:
                print(node)
                visited.add(node)
                for neighbour in graph[node]:
                        dfs(visited, graph, neighbour)

print("Following is the DFS")
dfs(visited, graph, '5')
```

b) BFS(Breadth-First Search):

AIM: To write a program to implement BFS using python.

ALGORITHM:

1. Start
2. Start by putting any 1 of the graphs vertices on top of a stack.
3. Take the top item of stack and add it to the visited list.
4. Create a list of that vertex adjacent nodes. Add the one or not in visited list to top of stack.
5. Keep repeating steps 2 and 3 until the stack is empty.
6. Stop

PROGRAM:

```
graph = {
    '5' : ['3', '7'],
    '3' : ['2', '4'],
    '7' : ['8'],
    '2' : [ ],
    '4' : ['8'],
    '8' : [ ]
}
```

OUTPUT:

Following is the BFS

5 3 7 2 4 8

```
visited = [ ]
queue = [ ]
def bfs(visited, graph, node):
        visited.append(node)
        queue.append(node)
        while queue:
            m=queue.pop(0)
            print(m, end=" ")
            for neighbour in graph[m]:
                if neighbour not in visited:
                    visited.append(neighbour)
                    queue.append(neighbour)
print("Following is the BFS")
bfs(visited, graph, '5')
```

Result: Hence, program to implement BFS and DFS has been executed successfully.

8/8/24

Exp.No:02 | Travelling Salesman Problem | Date: 8/8/2024

Write a program for Travelling Salesman Problem.

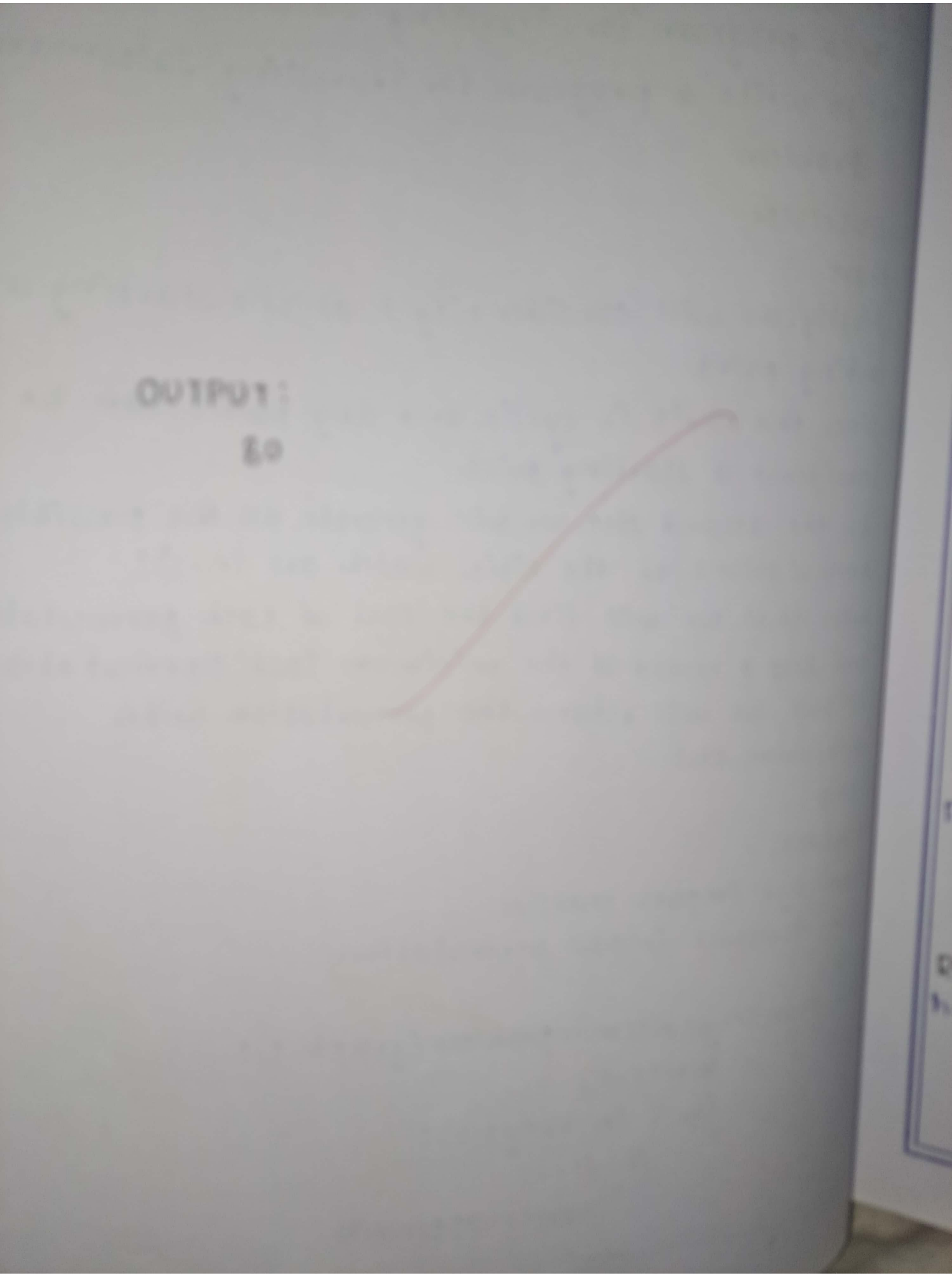
AIM: To write a program for Travelling Salesman Problem.

ALGORITHM:

1. Start
2. Firstly, we will consider city 1 as the starting and ending point.
3. Since the route is cyclic, and any point can be considered a starting point.
4. As the second step, we will generate all the possible permutations of the cities which are (n-1)!
5. After that, we will find the cost of each permutation and keep a record of the minimum last permutation.
6. At last, we will return the permutation with minimum cost.
7. Stop

PROGRAM:

```
from sys import maxsize
from itertools import permutations
V=4
def travellingSalesmanproblem(graph,s):
        vertex=[]
        for i in range(v):
            if i!=s:
                vertex.append(i)
```



```
        min_path = maxsize
        next_permutation = permutations(vertex)
        for i in next_permutation:
            current_pathweight = 0
            k = s
            for j in i:
                current_pathweight += graph[k][j]
                k = j
            current_pathweight += graph[k][s]
            min_path = min(min_path, current_path
                                        weight)
        return min_path
# Driver Code
if __name__ == "__main__":
        graph = [[0, 10, 15, 20], [10, 0, 35, 25],
                 [15, 35, 0, 30], [20, 25, 30, 0]]
        s = 0
print(travellingSalesmanproblem(graph, s))
```

Result: Hence, program for Travelling Salesman Problem has been executed successfully.

8/8/24

Exp.No:03 Stimulated Annealing Algorithm Date:8/8/2024

AIM: To write a program to implement stimulated annealing algorithm.

ALGORITHM:

Step.1: Start

Step.2: Initialize start with a random initial placement through a defined move.

Step.3: Move perturb the placement in very high temp.

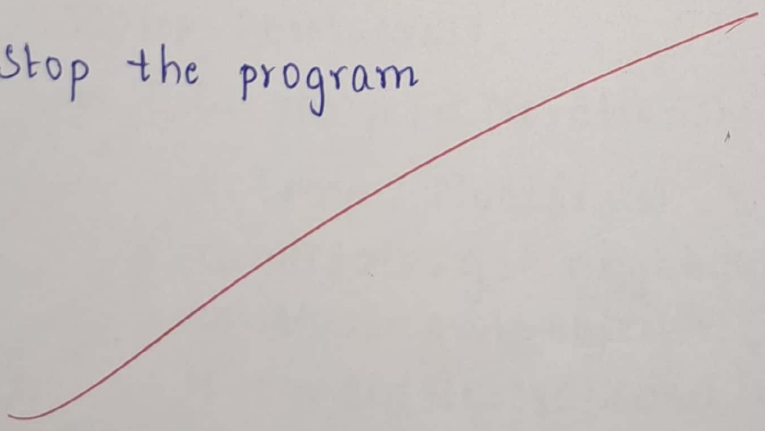
Step.4: calculate score_calculate the change in the score due to move made.

Step.5: Choose depending on the change in score accept or reject the move, the probability of acceptance depending upon the current temperature.

Step.6: Of update and repeat_update the temperature value by lowering the temperature. Go back to step.2

Step.7: The process is done until "Freezing point" is reached

Step.8: Stop the program



PROGRAM:

```
import math
import random
def objective_function(x):
    return 10*len(x)+sum([xi**2-10*math.cos(2*
                          math.pi*xi)) for xi in x])

def get_neighbour(x,step_size=0.1):
        neighbour=x[:]
        index=random.randint(0,len(x)-1)
        neighbour[index]+=random.uniform(-step_size,
      return neighbour                    step_size)
def simulated_annealing(objective,bounds,n_iterations,
                                    step_size,temp):
        best=[random.uniform(bound[0],bound[1]) for
                                    bound in bounds]
        best_eval=objective(best)
        current,current_eval=best,best_eval
        scores=[best_eval]
        for i in range(n_iterations):
            t=temp/float(i+1)
            candidate=get_neighbour(current,step_size)
            candidate_eval=objective(candidate)
            if candidate_eval<best_eval or random.
random()<math.exp((current_eval-candidate_eval)/t):
                current,current_eval=candidate,candidate_
                                                    eval
```

OUTPUT:

Iteration 0, Temperature 10.000, Best Evaluation 27.087328

Iteration 100, Temperature 0.099, Best Evaluation 12.94984

Iteration 200, Temperature 0.050, Best Evaluation 12.93719

Iteration 300, Temperature 0.033, Best Evaluation 12.93500

Iteration 400, Temperature 0.025, Best Evaluation 12.93500

Iteration 500, Temperature 0.020, Best Evaluation 12.93500

Iteration 600, Temperature 0.017, Best Evaluation 12.93500

Iteration 700, Temperature 0.014, Best Evaluation 12.93495

Iteration 800, Temperature 0.012, Best Evaluation 12.93495

Iteration 900, Temperature 0.011, Best Evaluation 12.93495

Best Solution : [-2.9851690070596724, -1.9914945515505844]

Best Score : 12.934947647329672

```
            if candidate_eval < best_eval:
                best, best_eval = candidate, candidate_eval
                scores.append(best_eval)
        if i % 100 == 0:
            print(f"iteration {i}, Temperature {t:.3f},
            Best Evaluation {best_eval:.5f}")
return best, best_eval, scores
bounds = [(-5.0, 5.0) for _ in range(2)]
n_iterations = 1000
step_size = 0.1
temp = 10
best, score, scores = simulated_annealing(objective_function,
bounds, n_iterations, step_size, temp)
print(f'Best Solution : {best}')
print(f' Best Score : {score}')
```

Result: Hence, the program to implement stimulated annealing algorithm has been executed successfully.

5/9/24

Exp.No:05 | 8-Puzzle Problem | Date: 5/9/2024

Aim: To write a program to implement 8 puzzle problem.

Algorithm:

1. Start from the root node.
2. Explore the leftnode childnode recursively until you reach a leaf node.
3. If a goal state is reached, return the solution.
4. If leaf node is reached without finding a solution, back track to explain other branches.

Program:

```
class Solution:
    def solve(self, board):
        dict={}
        flatten=[]
        for i in range(len(board)):
            flatten += board[i]
        flatten = tuple(flatten)
        dict[flatten]=0
        if flatten == (0,1,2,3,4,5,6,7,8):
            return 0
        return self.get_path(dict)
    def get_paths(self,dict):
        cnt=0
```

```
while True:
    current_nodes=[x for x in dict if dict[x]=cnt]
    if len(current_nodes)==0:
        return -1
    for node in current_nodes:
        next_moves = self.find_next(node)
        for move in next_moves:
            if move not in dict:
                dict[move]=cnt+1
            if move==[0,1,2,3,4,5,6,7,8]:
                return cnt+1
                cnt+=1
def find_next(self,node):
    moves={
    0:[1,3],
    1:[0,2,4],
    2:[1,5],
    3:[0,4,6],
    4:[1,3,5,7],
    5:[2,4,8],
    6:[3,7],
    7:[4,6,8],
    8:[5,7],
    }
```

Output:

4

```
        results=[]
        pos_0=node.index(0)
        for move in moves[pos_0]:
            new_node=list(node)
            new_node[move],new_node[pos_0]=
                                    new_node
            [pos_0],new_node[move]
            results.append(tuple(new_node))
        return results()
ob=solution()
matrix=[
    [3,1,2],
    [4,7,5],
    [4,8,0]
    ]
print(ob.solve(matrix))
```

**Result:** Hence, the above program has been executed successfully.

Exp.No:06 | Tower of Hanoi | Date: 5/9/2024

Aim: To implement Tower of Hanoi problem.

Algorithm:

* Create a tower of Hanoi recursive function and pass two arguments: The no.of disks n and the name of rods such as source, aux and target.
* We can define the base when the no.of disks is 1. In this case simply move the one-disk from the source to target and return.
* Now, move remaining n-1 disks from source to auxilary using the target as auxilary.
* Then, the remaining 1 disk move on the source to target.
* Move the n-1 disks on the auxilary to the target using the source as the auxilary.

Program:

```
def TowerOfHanoi(n, source, destination, auxilary):
        if n==1:
            print("Move disk1 from source", source, "to
                    destination", destination)

            return
        TowerofHanoi(n-1, source, auxilary, destination)
        print("Move disk", n, "from source", source, "to destination",
                                                    destination)
        TowerOfHanoi(n-1, auxilary, destination, source)
n=4
TowerOfHanoi(n, 'A', 'B', 'C')
```

Output:

Move disk1 from source A to destination C
Move disk2 from Source A to destination B
Move disk1 from source C to destination B
Move disk3 from source A to destination C
Move disk1 from source B to destination A
Move disk2 from source B to destination C
Move disk1 from source A to destination C
Move disk4 from source A to destination B
Move disk1 from source C to destination B
Move disk2 from source C to destination A
Move disk1 from source B to destination A
Move disk3 from source C to destination B
Move disk1 from source A to destination C
Move disk2 from source A to destination B
Move disk1 from source C to destination B

Exp.No:7 | A* Algorithm | Date:19/9/2024

Aim: To write a program for implementing A* Algorithm.

Algorithm:

1. Add the beginning node to openList. In openlist find the square with lowest fcost, which denotes current square.
2. Consider 8 squares adjacent to current square and ignore if it is an closed list or if not workable.
3. If it is openlist use Gcost to measure the better plan.
4. Now recalculate the scores of G, F scores of the square.
5. Now, you can save path and work backward starting from target square.

Program:

```
def aStarAlgo(start_node, stop_node):
        open_set = set(start_node)
        closed_set = set()
        g = { }
        parents = { }
        g[start_node] = 0
        parents[start_node] = start_node
        while len(open_set) > 0:
            n = None
            for v in open_set:
                if n == None or g[v] + heuristic(v) < g[n] +
                heuristic(n):
```

```
            n=v
        if n==stop_node or Graph_nodes[n]==None:
            pass
        else:
            for(m,weight) in get_neighbors(n):
                if m not in open_set and m not in
                                        closed_set:
                    open_set.add(m)
                    parents[m]=n
                    g[m]=g[n]+weight
els             else:
                    if g[m]>g[n]+weight:
                        g[m]=g[n]+weight
                        parents[m]=n
                        if m in closed_set:
                            closed_set.remove(m)
                            open_set.add(m)
    if n==None:
        print('Path does not exist')
        return None
    if n==stop_node:
        path=[]
        while parents[n]!=n:
            path.append(n)
            n=parents[n]
```

```
                path.append(start_node)
                path.reverse()
                print('Path found:{}'.format(path))
                return path
                open_set.remove(n)
                closed_set.add(n)
        print('path doesn't exist')
        return None
def get_neighbors(v):
        if v in Graph_nodes:
            return Graph_nodes[v]
        else:
            return None
def heuristic(n):
        H_dist = {
            'A':11,
            'B':6,
            'C':99,
            'D':1,
            'E':7,
            'G':0,
          }
        return H_dist[n]
```

Output:

Path Found : ['A', 'E', 'D', 'G']

```
Graph_nodes = {
        'A' : [('B', 2), ('E', 3)],
        'B' : [('C', 1), ('G', 9)],
        'C' : None,
        'E' : [('D', 6)],
        'D' : [('G', 1)]
      }
aStarAlgo('A', 'G')
```

**Result:** Hence, the above program has been executed successfully.

10

26/9/24

Exp.No:08 | Hill Climbing -Algorithm | Date: 26/9/24

Aim: To write a program to implement hill climbing problem.

Algorithm:

1. Evaluate the initial state. If it is good state quit, otherwise make current state as initial state.
2. Select a new operator that could be applied to this state and generate a new state.
3. Evaluate the new state. If this new state is closer to the goal state.
4. If the current state is good state or not quit otherwise repeat from 2.

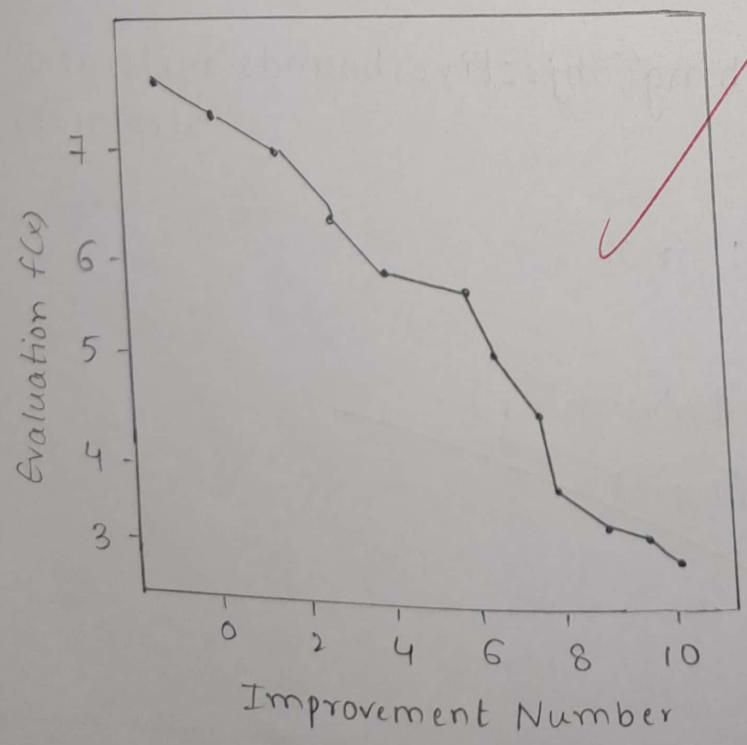
Program:

```
from numpy import asarray
from numpy.random import randn
from numpy.random import rand
from numpy.random import seed
from numpy. matplotlib import pyplot
def objective(x):
    return x[0]**2.0
def hillclimbing(objective, bounds, n_iterations, step_size):
    solution=bounds[:,0]+rand(len(bounds))*(bounds[:,1]
                                            -bounds[:,0])
    solution_eval = objective(solution)
```

```
    scores = list()
    scores.append(solution_eval)
    for i in range(n_iterations):
        candidate = solution + randn(len(bounds)) * step_size
        candidate_eval = objective(candidate)
        if candidate_eval <= solution_eval:
            solution, solution_eval = candidate, candidate_e
            scores.append(solution_eval)
            print('>%d f(%s) = %.5f' % (i, solution,
                                        solution_eval))
    return [solution, solution_eval, scores]
seed(5)
bounds = asarray([[-5.0, 5.0]])
n_iterations = 25
step_size = 0.1
best, score, scores = hillclimbing(objective, bounds, n_iterations,
print('Done!')                                    step_size)
print('f(%s) = %f' % (best, score))
pyplot.plot(scores, '.-')
pyplot.xlabel('Improvement Number')
pyplot.ylabel('Evaluation f(x)')
pyplot.show()
```

Output:

```
>1  f([-2.74290923]) = 7.523551
>3  f([-2.65873147]) = 7.068853
>4  f([-2.52197291]) = 6.360347
>5  f([-2.46450214]) = 6.073771
>7  f([-2.44740961]) = 5.989814
>9  f([-2.28364676]) = 5.215043
>12 f([-2.19245939]) = 4.806878
>14 f([-2.01001538]) = 4.040162
>15 f([-1.86425287]) = 3.475439
>22 f([-1.79913002]) = 3.236869
>24 f([-1.57525573]) = 2.481431
Done!
f([-1.57525573]) = 2.481431
```



Exp.No:12 Date:17/10/24

Write a program using Function that counts the no.of times a string occurs in another string.

AIM: To write a program using function that counts the no.of times a string occurs in another string.

PROCEDURE:

m:length of str1 (first string)

n:length of str2 (second string)

If last character, consider and get count for remaining strings. So we recur for length m-1 and n-1

. We can ignore last character of first string and recurse for length m-1 and n.

else

if last characters are not same

We ignore last character of first string and recurse for length m-1 and n.

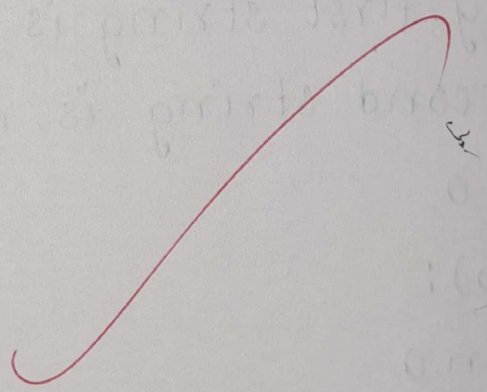
PROGRAM:

```
#A Naive recursive python program
#to find the number of times the
#second string occurs in the first
#string, where continuous or
#discontinuous
# Recursive function to find the
#no.of times the second string
# occurs in the first string
```

```
#whether continuous or discontinuous
def count(a,b,m,n):
    #if both first and second string
    #is empty,or if second string
    #is empty,return 1
    if((m==0 and n==0) or n==0):
      return 1
    #if only first string is empty
    #and second string is not empty
    #return 0
    if(m==0):
       return 0
    #if last characters are same
    #Recur for remaining strings by
    #1.considering last characters
    #  of both strings
    #2.ignoring last character
    # of first string
    if(a[m-1]==b[n-1]):
        return(count(a,b,m-1,n-1)+count(a,b,m-1,n))
    else:
      #if last characters are different
      #ignore last char of first string
```

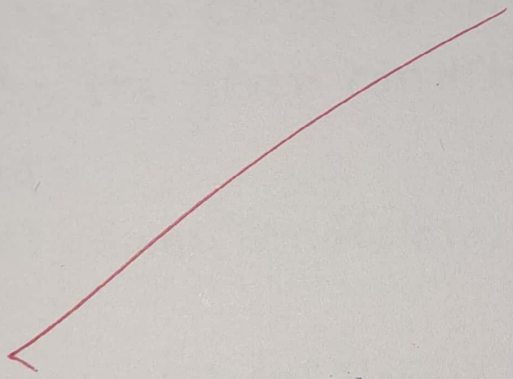
Output:

4

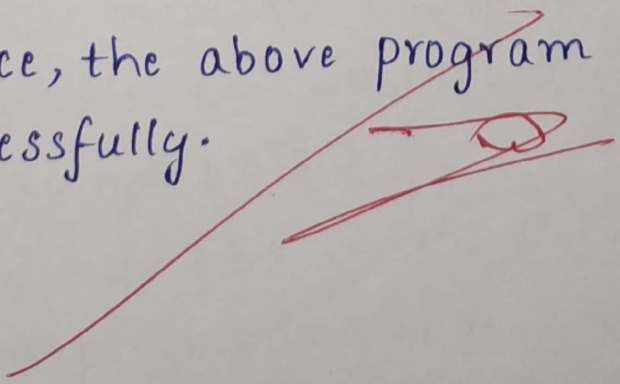


```
        # and recur for remaining string
        return count(a, b, m-1, n)

# Driver code
a = "GeeksforGeeks"
b = "Gks"
print(count(a, b, len(a), len(b)))
```



Result: Hence, the above program has been executed successfully.



Exp.No:13 | Higher Order Function | Date:24/10/24

AIM: To write a higher order function counts that no.of elements in a list that satisfy a given test in python.

PROGRAM:

```
from funtools import reduce
def getCount(listOfElems, cond=None):
    'Returns the count of elements in list that
    satisfies the given condition'
    if cond:
      count = sum(cond(elem)for elem in listOfElems)
    else:
      count = len(listOfElems)
    return count
def main():
#list of numbers
listOfElems = [11, 22, 33,45, 66, 77, 88,99,101]
print('***Use map()&sum() to count elements in a
list that satisfies certain conditions***')
print('** Example 1**')
#Count odd numbers in the list
count = sum(map(lambda x : x%2==1, listOfElems))
print ('Count of odd numbers in a list : ', count)
print('*** Example 1: Explanation **)
```

```
#Get a map objective by applying given lambda to
 each element in list
mapObj = map(lambda x : x%.2 == 1, listOfElems)
print('Contents of map, object : ', list(mapObj))
print('** Example 2 **)
#Count even numbers in the list
count = sum(map(lambda x : x%.2 == 0, listOfElems))
print('Count of even numbers in a list : ', count)
print ('** Example 3 **')
#Elements Greater than 5
count = sum(map(lambda x : x>5, listOfElems))
print('Count of number in a list which are greater than
                                        5: ', count)

print ('*** using sum() & Generator expression to
 count elements in list based on conditions ***')
count = getCount(listOfElems, lambda · x : x>5)
print('Count of numbers in a list which are greater
                                    than 5: ', count)

#count numbers which are >5 and <20
print('Count of numbers in a list which are greater
    than 5 and but less than 20 : ', count)
count = getCount(listOfElems)
Print('Total number of elements in list : ', count)
```

Output:

*** Example1: Use map() & sum() to count elements in list that satisfy certain conditions ***

Count of odd numbers in a list: 6

Count of even numbers in a list: 3

Count of numbers in a list greater than 5: 9

*** Using sum() & Generator expression to count elements in list based on conditions ***

Count of numbers in a list which are greater than 5: 542

Count of numbers in a list which are greater than 5 but less than 20: 11

Total number of elements in a list: 9

*** Use list comprehension to count elements in list based on conditions ***

Count of numbers in a list which are greater than 5:

*** Use reduce() function to count elements in list based on conditions ***

Count of numbers in a list which are greater than 5:

```
print('*** Use list comprehension to count elements in
      list based on conditions ***')
count = len([elem for elem in listOfElems if elem>5])
print('Count of numbers in a list which are greater
                                    than 5:', count)

print('*** use reduce() function to count elements
      in list based on conditions ***')
count = reduce(lambda default, elem: default + (elem>5),
                                    listOfElem, 0)

print('Count of Numbers in a list which are greater than
                                    5:', count)

if __name__=='__main__':
    main()
```

Result: Hence, the above program has been executed successfully.

Exp.No:14 | Knapsack Problem | Date: 24/10/24

AIM: To write a function that allows you to generate instance of knapsack problem.

ALGORITHM:

The maximum value obtained from 'N' items is the max of the following two values

case 1 (include the $N^{th}$ item): value of the $N^{th}$ item plus maximum value obtained by remaining N-1 items and remaining weight i.e (W - weight of the $N^{th}$ item)

case 2 (exclude the $N^{th}$ item): Maximum value obtained by N-1 items and W weight

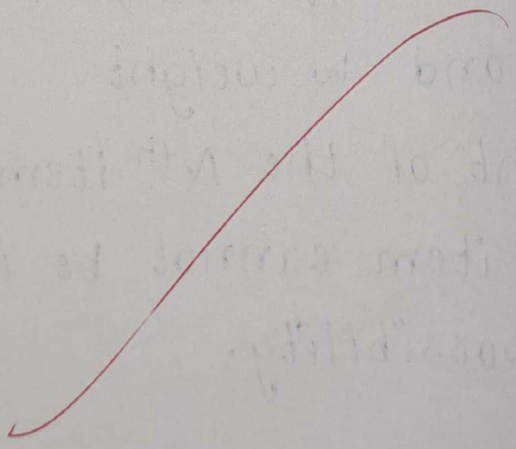
- If the weight of the $N^{th}$ item is greater than 'W' then the $N^{th}$ item cannot be included and case 2 is the only possibility.

PROGRAM:

```
def knapSack(W, wt, val, n):
    # Base Case
    if n==0 or W==0:
        return 0
    #if weight of the Nth element is
    if (wt[n-1] > W):
        return knapSack(W, wt, val, n-1)
    else:
        return max(
```

output:

220



```
            val[n-1] + knapSack(
              W - wt[n-1], wt, val, n-1),
            knapSack(W, wt, val, n-1))

if __name__ == '__main__':
    profit = [60, 100, 120]
    weight = [10, 20, 30]
    W = 50
    n = len(profit)
    print(knapSack(W, weight, profit, n))
```

Result: Hence, the above program has been executed successfully.