AI LAB MANUAL

(R20)

AI -LAB MANUAL

1. Write a program to implement DFS and BFS

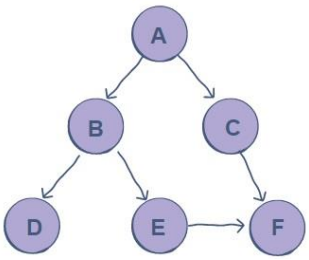
Breadth-first search (BFS) is an algorithm used for tree traversal on graphs or tree data structures. BFS can be easily implemented using recursion and data structures like dictionaries and lists.

The Algorithm

1. Pick any node, visit the adjacent unvisited vertex, mark it as visited, display it, and insert it in a queue.
2. If there are no remaining adjacent vertices left, remove the first vertex from the queue.
3. Repeat step 1 and step 2 until the queue is empty or the desired node is found.

Implementation

Consider the below graph, which is to be implemented:



SOURCE CODE:

```
graph = {
  'A' : ['B','C'],
  'B' : ['D', 'E'],
  'C' : ['F'],
  'D' : [],
  'E' : ['F'],
  'F' : []
}
visited = [] # List to keep track of visited nodes.
queue = []     #Initialize a queue

def bfs(visited, graph, node):
```

```
    visited.append(node)
    queue.append(node)

    while queue:
      s = queue.pop(0)
      print (s, end = " ")

      for neighbour in graph[s]:
        if neighbour not in visited:
          visited.append(neighbour)
          queue.append(neighbour)
bfs(visited, graph, 'A')
```

OUTPUT:

A B C D E F

Explanation

- Lines 3-10: The illustrated graph is represented using an adjacency list. An easy way to do this in Python is to use a dictionary data structure, where each vertex has a stored list of its adjacent nodes.
- Line 12: visited is a list that is used to keep track of visited nodes.
- Line 13: queue is a list that is used to keep track of nodes currently in the queue.
- Line 29: The arguments of the bfs function are the visited list, the graph in the form of a dictionary, and the starting node A.
- Lines 15-26: bfs follows the algorithm described above:

1. It checks and appends the starting node to the visited list and the queue.
2. Then, while the queue contains elements, it keeps taking out nodes from the queue, appends the neighbors of that node to the queue if they are unvisited, and marks them as visited.

3. This continues until the queue is empty.

Time Complexity

Since all of the nodes and vertices are visited, the time complexity for BFS on a graph is O(V + E)$O(V+E)$; where VV is the number of vertices and EE is the number of edges.

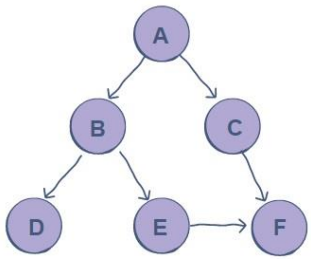
Depth-first search (DFS), is an algorithm for tree traversal on graph or tree data structures. It can be implemented easily using recursion and data structures like dictionaries and sets.

The Algorithm

1. Pick any node. If it is unvisited, mark it as visited and recur on all its adjacent nodes.
2. Repeat until all the nodes are visited, or the node to be searched is found.

Implementation

Consider the below graph, which is to be implemented:



SOURCE CODE:

 # Using a Python dictionary to act as an adjacency list

```
graph = {
  'A' : ['B','C'],
  'B' : ['D', 'E'],
  'C' : ['F'],
  'D' : [],
  'E' : ['F'],
  'F' : []
}
```

```
visited = set() # Set to keep track of visited nodes.

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

OUTPUT:

A B C D E F

Explanation

- **Lines 2-9**: The illustrated graph is represented using an **adjacency list** - an easy way to do it in Python is to use a dictionary data structure. Each vertex has a list of its adjacent nodes stored.
- **Line 11**: `visited` is a set that is used to keep track of visited nodes.
- **Line 21**: The `dfs` function is called and is passed the `visited` set, the `graph` in the form of a dictionary, and `A`, which is the starting node.
- **Lines 13-18**: `dfs` follows the algorithm described above:
    1. It first checks if the current node is unvisited - if yes, it is appended in the `visited` set.
    2. Then for each neighbor of the current node, the `dfs` function is invoked again.
    3. The base case is invoked when all the nodes are visited. The function then returns.

Time Complexity

- Since all the nodes and vertices are visited, the average time complexity for DFS on a graph is O(V + E)$O(V+E)$, where VV is the number of vertices and EE is the number of edges. In case of DFS on a tree, the time complexity is O(V)$O(V)$, where VV is the number of nodes.

Note: We say average time complexity because a set's in operation has an average time complexity of O(1)O(1). If we used a list, the complexity would be higher.

2. Write a Program to find the solution for travelling salesman Problem.

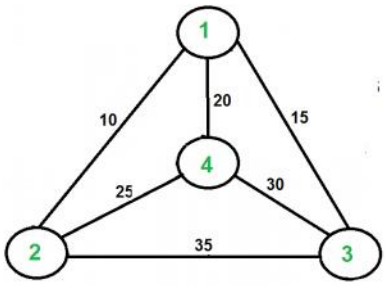
What is a Travelling Salesperson Problem?

- The travelling salesperson problem (TSP) is a classic optimization problem where the goal is to determine the shortest tour of a collection of n "cities" (i.e. nodes), starting and ending in the same city and visiting all of the other cities exactly once.
- In such a situation, a solution can be represented by a vector of n integers, each in the range 0 to n-1, specifying the order in which the cities should be visited.
- TSP is an NP-hard problem, meaning that, for larger values of n, it is not feasible to evaluate every possible problem solution within a reasonable period of time. Consequently, TSPs are well suited to solving using randomized optimization algorithms.

Traveling Salesman Problem (TSP) Implementation

Travelling Salesman Problem (TSP) : Given a set of cities and distances between every pair of cities, the problem is to find the shortest possible route that visits every city exactly once and returns to the starting point.

Note the difference between Hamiltonian Cycle and TSP. The Hamiltoninan cycle problem is to find if there exist a tour that visits every city exactly once. Here we know that Hamiltonian Tour exists (because the graph is complete) and in fact many such tours exist, the problem is to find a minimum weight Hamiltonian Cycle.



Explanation

1. Consider city 1 as the starting and ending point. Since the route is cyclic, we can consider any point as a starting point.
2. Generate all (n-1)! permutations of cities.

3. Calculate the cost of every permutation and keep track of the minimum cost permutation.
4. Return the permutation with minimum cost.

SOURCE CODE

```
# Python3 program to implement traveling salesman problem using naive approach.
from sys import maxsize
from itertools import permutations
V = 4
# implementation of traveling Salesman Problem
def travellingSalesmanProblem(graph, s):
    # store all vertex apart from source vertex
    vertex = []
    for i in range(V):
        if i != s:
            vertex.append(i)
    # store minimum weight Hamiltonian Cycle
    min_path = maxsize
    next_permutation=permutations(vertex)
    for i in next_permutation:
        # store current Path weight(cost)
        current_pathweight = 0
        # compute current path weight
        k = s
        for j in i:
            current_pathweight += graph[k][j]
            k = j
        current_pathweight += graph[k][s]
        # update minimum
        min_path = min(min_path, current_pathweight)
    return min_path
if __name__ == "__main__":
```

```
# matrix representation of graph
graph = [[0, 10, 15, 20], [10, 0, 35, 25],
        [15, 35, 0, 30], [20, 25, 30, 0]]
s = 0
print(travellingSalesmanProblem(graph, s))
```

OUTPUT

80

3. Write a program to implement Simulated Annealing Algorithm

SOURCE CODE

```
# convex unimodal optimization function
from numpy import arange
from matplotlib import pyplot

# objective function
def objective(x):
        return x[0]**2.0
# define range for input
r_min, r_max = -5.0, 5.0
# sample input range uniformly at 0.1 increments
inputs = arange(r_min, r_max, 0.1)
# compute targets
results = [objective([x]) for x in inputs]
# create a line plot of input vs result
pyplot.plot(inputs, results)
# define optimal input value
x_optima = 0.0
# draw a vertical line at the optimal input
pyplot.axvline(x=x_optima, ls='--', color='red')
# show the plot
```

```
pyplot.show()
```

SOURCE CODE

```
# explore temperature vs algorithm iteration for simulated annealing
from matplotlib import pyplot
# total iterations of algorithm
iterations = 100
# initial temperature
initial_temp = 10
# array of iterations from 0 to iterations - 1
iterations = [i for i in range(iterations)]
# temperatures for each iterations
temperatures = [initial_temp/float(i + 1) for i in iterations]
# plot iterations vs temperatures
pyplot.plot(iterations, temperatures)
pyplot.xlabel('Iteration')
pyplot.ylabel('Temperature')
pyplot.show()
```

SOURCE CODE

```
# explore metropolis acceptance criterion for simulated annealing
from math import exp
from matplotlib import pyplot
# total iterations of algorithm
iterations = 100
# initial temperature
initial_temp = 10
# array of iterations from 0 to iterations - 1
iterations = [i for i in range(iterations)]
# temperatures for each iterations
```

```
temperatures = [initial_temp/float(i + 1) for i in iterations]
# metropolis acceptance criterion
differences = [0.01, 0.1, 1.0]
for d in differences:
        metropolis = [exp(-d/t) for t in temperatures]
        # plot iterations vs metropolis
        label = 'diff=%.2f' % d
        pyplot.plot(iterations, metropolis, label=label)
# inalize plot
pyplot.xlabel('Iteration')
pyplot.ylabel('Metropolis Criterion')
pyplot.legend()
pyplot.show()
```

4. Write a program to find the solution for wampus world problem

SOURCE CODE

```
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
plt.rcParams['font.sans-serif'] = ['SimHei']
# Data generation
train_num = 200
test_num = 100
config = {
   'Corn': [[150, 190], [40, 700], [20,4]],
   'Potato': [[300, 600], [70, 10], [10, 20]],
   'grass': [[100, 40], [10, 40], [505, 1]]
}
plants = list(config.keys())
dataset = pd.DataFrame(columns=['height(cm)', 'Leaf length(cm)', 'Stem diameter(cm)', 'type'])
index = 0
```

```
# Natural
for p in config:
    for i in range(int(train_num/3-3)):
        row = []
        for  j, [min_val, max_val] in enumerate(config[p]):
            v = round(np.random.rand()*(max_val-min_val)+min_val, 2)
            while v in dataset[dataset.columns[j]]:
                v = round(np.random.rand()*(max_val-min_val)+min_val, 2)
            row.append(v)
        row.append(p)
        dataset.loc[index] = row
        index +=  1
# Wrong data
for i in range(train_num - index):
    k = np.random.randint(3)
    p = plants[k]
    row = []
    for  j, [min_val, max_val] in enumerate(config[p]):
        v = round(np.random.rand()*(max_val-min_val)+min_val, 2)
        while v in dataset[dataset.columns[j]]:
            v = round(np.random.rand()*(max_val-min_val)+min_val, 2)
        row.append(v)
    row.append(plants[(k+1)%3])
    dataset.loc[index] = row
    index+=1
# dataset = dataset.infer_objects()
dataset = dataset.reindex(np.random.permutation(len(dataset)))
dataset.reset_index(drop=True, inplace=True)
dataset.iloc[:int(train_num), :-1].to_csv('potato_train_data.csv', index=False)
dataset.iloc[:int(train_num):, [-1]].to_csv('potato_train_label.csv', index=False)
"""Here, only the training data set is generated, and the test data is similar to this
```

Data visualization

We can see the distribution of data points by drawing a scatter diagram of the data of two dimensions."""

```
def visualize(dataset, labels, features, classes, fig_size=(10, 10), layout=None):
    plt.figure(figsize=fig_size)
    index = 1
    if layout == None:
        layout = [len(features), 1]
    for i in range(len(features)):
        for j in range(i+1, len(features)):
            p = plt.subplot(layout[0], layout[1], index)
            plt.subplots_adjust(hspace=0.4)
            p.set_title(features[i]+'&'+features[j])
            p.set_xlabel(features[i])
            p.set_ylabel(features[j])
            for k in range(len(classes)):
                p.scatter(dataset[labels==k, i], dataset[labels==k, j], label=classes[k])
            p.legend()
            index += 1
    plt.show()

dataset = pd.read_csv('potato_train_data.csv')
labels = pd.read_csv('potato_train_label.csv')
features = list(dataset.keys())
classes = np.array(['Corn', 'Potato', 'grass'])
for i in range(3):
    labels.loc[labels['type']==classes[i], 'type'] = i
dataset = dataset.values
labels = labels['type'].values
visualize(dataset, labels, features, classes)
```

5. Write a program to implement 8 puzzle problem

SOURCE CODE

```
import copy
from heapq import heappush, heappop
n = 3
row = [ 1, 0, -1, 0 ]
col = [ 0, -1, 0, 1 ]
class priorityQueue:
    def __init__(self):
        self.heap = []
    def push(self, k):
        heappush(self.heap, k)
    def pop(self):
        return heappop(self.heap)
    def empty(self):
        if not self.heap:
            return True
        else:
            return False
class node:
    def __init__(self, parent, mat, empty_tile_pos,
                 cost, level):
        self.parent = parent
        self.mat = mat
        self.empty_tile_pos = empty_tile_pos
        self.cost = cost
        self.level = level
    def __lt__(self, nxt):
        return self.cost < nxt.cost
def calculateCost(mat, final) -> int:
    count = 0
```

```
    for i in range(n):
        for j in range(n):
            if ((mat[i][j]) and
                (mat[i][j] != final[i][j])):
                count += 1
    return count
def newNode(mat, empty_tile_pos, new_empty_tile_pos,
                level, parent, final) -> node:
    new_mat = copy.deepcopy(mat)
    x1 = empty_tile_pos[0]
    y1 = empty_tile_pos[1]
    x2 = new_empty_tile_pos[0]
    y2 = new_empty_tile_pos[1]
    new_mat[x1][y1], new_mat[x2][y2] = new_mat[x2][y2], new_mat[x1][y1]
    cost = calculateCost(new_mat, final)
    new_node = node(parent, new_mat, new_empty_tile_pos,
                                    cost, level)
    return new_node
def printMatrix(mat):
    for i in range(n):
        for j in range(n):
            print("%d " % (mat[i][j]), end = " ")
        print()
def isSafe(x, y):
    return x >= 0 and x < n and y >= 0 and y < n
def printPath(root):
    if root == None:
        return
    printPath(root.parent)
    printMatrix(root.mat)
    print()
```

```
def solve(initial, empty_tile_pos, final):
        pq = priorityQueue()
        cost = calculateCost(initial, final)
        root = node(None, initial,
                                empty_tile_pos, cost, 0)
        pq.push(root)
        while not pq.empty():
                minimum = pq.pop()
                if minimum.cost == 0:
                        printPath(minimum)
                        return
                for i in range(n):
                        new_tile_pos = [
                                minimum.empty_tile_pos[0] + row[i],
                                minimum.empty_tile_pos[1] + col[i], ]
                        if isSafe(new_tile_pos[0], new_tile_pos[1]):
                                child = newNode(minimum.mat,
                                                        minimum.empty_tile_pos,
                                                        new_tile_pos,
                                                        minimum.level + 1,
                                                        minimum, final,)
                                pq.push(child)
initial = [ [ 1, 2, 3 ],
                        [ 5, 6, 0 ],
                        [ 7, 8, 4 ] ]
final = [ [ 1, 2, 3 ],
                [ 5, 8, 6 ],
                [ 0, 7, 4 ] ]
empty_tile_pos = [ 1, 2 ]
solve(initial, empty_tile_pos, final)
```

OUTPUT:

```
1  2  3
5  6  0
7  8  4

1  2  3
5  0  6
7  8  4

1  2  3
5  8  6
7  0  4

1  2  3
5  8  6
0  7  4
```

6. Write a program to implement Towers of Hanoi problem

SOURCE CODE

```
class Tower:
  def __init__(self):
    self.terminate = 1
  def printMove(self, source, destination):
    print("{} -> {}".format(source, destination))
  def move(self, disc, source, destination, auxiliary):
    if disc == self.terminate:
      self.printMove(source, destination)
    else:
      self.move(disc - 1, source, auxiliary, destination)
      self.move(1, source, destination, auxiliary)
      self.move(disc - 1, auxiliary, destination, source)
t = Tower();
t.move(3, 'A', 'B', 'C')
```

OUTPUT

```
A -> B
A -> C
B -> C
```

A -> B

C -> A

C -> B

A -> B

7. Write a program to implement A* Algorithm

SOURCE CODE

```
from queue import PriorityQueue

#Creating Base Class
class State(object):
    def __init__(self, value, parent, start = 0, goal = 0):
        self.children = []
        self.parent = parent
        self.value = value
        self.dist = 0
        if parent:
            self.start = parent.start
            self.goal = parent.goal
            self.path = parent.path[:]
            self.path.append(value)

        else:
            self.path = [value]
            self.start = start
            self.goal = goal

    def GetDistance(self):
        pass
    def CreateChildren(self):
        pass
 # Creating subclass
```

```
class State_String(State):
    def __init__(self, value, parent, start = 0, goal = 0 ):
        super(State_String, self).__init__(value, parent, start, goal)
        self.dist = self.GetDistance()

    def GetDistance(self):
            if self.value == self.goal:
                return 0
            dist = 0
            for i in range(len(self.goal)):
                letter = self.goal[i]
                dist += abs(i - self.value.index(letter))
            return dist

    def CreateChildren(self):
            if not self.children:
                for i in range(len(self.goal)-1):
                    val = self.value
                    val = val[:i] + val[i+1] + val[i] + val[i+2:]
                    child = State_String(val, self)
                    self.children.append(child)

# Creating a class that hold the final magic
class A_Star_Solver:
    def __init__(self, start, goal):
        self.path = []
        self.vistedQueue =[]
        self.priorityQueue = PriorityQueue()
        self.start = start
        self.goal = goal
```

```
    def Solve(self):
        startState = State_String(self.start,0,self.start,self.goal)

        count = 0
        self.priorityQueue.put((0,count, startState))
        while(not self.path and self.priorityQueue.qsize()):
            closesetChild = self.priorityQueue.get()[2]
            closesetChild.CreateChildren()
            self.vistedQueue.append(closesetChild.value)
            for child in closesetChild.children:
                if child.value not in self.vistedQueue:
                    count += 1
                    if not child.dist:
                        self.path = child.path
                        break
                    self.priorityQueue.put((child.dist,count,child))
        if not self.path:
            print("Goal Of  is not possible !" + self.goal )
        return self.path

# Calling all the existing stuffs
if __name__ == "__main__":
    start1 = "BHANU"
    goal1 = "NHUBA"
    print("Starting....")
    a = A_Star_Solver(start1,goal1)
    a.Solve()
    for i in range(len(a.path)):
        print("{0}){1}".format(i,a.path[i]))
```

8. Write a program to implement Hill Climbing Algorithm

SOURCE CODE

```
# hill climbing search of the ackley objective function
from numpy import asarray
from numpy import exp
from numpy import sqrt
from numpy import cos
from numpy import e
from numpy import pi
from numpy.random import randn
from numpy.random import rand
from numpy.random import seed

# objective function
def objective(v):
	x, y = v
	return -20.0 * exp(-0.2 * sqrt(0.5 * (x**2 + y**2))) - exp(0.5 * (cos(2 * pi * x) + cos(2 * pi * y))) + e + 20

# check if a point is within the bounds of the search
def in_bounds(point, bounds):
	# enumerate all dimensions of the point
	for d in range(len(bounds)):
		# check if out of bounds for this dimension
		if point[d] < bounds[d, 0] or point[d] > bounds[d, 1]:
			return False
	return True

# hill climbing local search algorithm
def hillclimbing(objective, bounds, n_iterations, step_size):
	# generate an initial point
	solution = None
```

```
    while solution is None or not in_bounds(solution, bounds):
        solution = bounds[:, 0] + rand(len(bounds)) * (bounds[:, 1] - bounds[:, 0])
    # evaluate the initial point
    solution_eval = objective(solution)
    # run the hill climb
    for i in range(n_iterations):
        # take a step
        candidate = None
        while candidate is None or not in_bounds(candidate, bounds):
            candidate = solution + randn(len(bounds)) * step_size
        # evaluate candidate point
        candidte_eval = objective(candidate)
        # check if we should keep the new point
        if candidte_eval <= solution_eval:
            # store the new point
            solution, solution_eval = candidate, candidte_eval
            # report progress
            print('>%d f(%s) = %.5f' % (i, solution, solution_eval))
    return [solution, solution_eval]

# seed the pseudorandom number generator
seed(1)
# define range for input
bounds = asarray([[-5.0, 5.0], [-5.0, 5.0]])
# define the total iterations
n_iterations = 1000
# define the maximum step size
step_size = 0.05
# perform the hill climbing search
best, score = hillclimbing(objective, bounds, n_iterations, step_size)
print('Done!')
```

```
print('f(%s) = %f' % (best, score))
```

9. Build a Chatbot using AWS Lex, Pandora bots.

A **Chatbot Python** is an intelligent piece of software that is capable of communicating and performing actions similar to a human. **Chatbot In Python Project Report** are used a lot in customer interaction, marketing on social network sites and instantly messaging the client. This **Chatbot In Python Tutorial** also includes the downloadable **Python Chatbot Code Download** source code for free.

By the way I have here a simple Live **Chat** System in PHP Free Source Code maybe you are looking for this source code too.

To start creating this **Chatbot In Python Tutorial**, make sure that you have **PyCharm IDE** installed in your computer.

By the way if you are new to python programming and you don't know what would be the the Python IDE to use, I have here a list of Best Python IDE for Windows, Linux, Mac OS that will suit for you.

Steps on how to create a **Chatbot In Python**

Chatbot In Python Tutorial With Source Code

- **Step 1: Create a project name.**
  First when you finished installed the **Pycharm IDE** in your computer, open it and then create a "**project name**" after creating a project name click the "**create**" button.
- **Step 2: Create a python file.**
  Second after creating a project name, "**right click**" your project name and then click "**new**" after that click the "**python file**".
- **Step 3: Name your python file.**
  Third after creating a python file, Name your python file after that click "**enter**".
- **Step 4: The actual code.**
  This is the actual coding on how to create **Chatbot In Python**, and you are free to copy this code and download the full source code given below.

SOURCE CODE

```
def send():
   send = "You:"+ e.get()
```

```
    text.insert(END,"\n" + send)
    if(e.get()=='hi'):
        text.insert(END, "\n" + "Bot: hello")
    elif(e.get()=='hello'):
        text.insert(END, "\n" + "Bot: hi")
    elif (e.get() == 'how are you?'):
        text.insert(END, "\n" + "Bot: i'm fine and you?")
    elif (e.get() == "i'm fine too"):
        text.insert(END, "\n" + "Bot: nice to hear that")
    else:
        text.insert(END, "\n" + "Bot: Sorry I didnt get it.")
text = Text(root,bg='light blue')
text.grid(row=0,column=0,columnspan=2)
e = Entry(root,width=80)
send = Button(root,text='Send',bg='blue',width=20,command=send).grid(row=1,column=1)
e.grid(row=1,column=0)
root = Tk()
root.title('IT SOURCCODE SIMPLE CHATBOT')
root.mainloop()
```

10. Build a bot which provides all the information related to your college.

SOURCE CODE

SOURCE CODE

```
def send():
    send = "You:"+ e.get()
    text.insert(END,"\n" + send)
    if(e.get()=='hi'):
        text.insert(END, "\n" + "Bot: hello")
    elif(e.get()=='hello'):
        text.insert(END, "\n" + "Bot: hi")
    elif (e.get() == 'how are you?'):
```

```
        text.insert(END, "\n" + "Bot: i'm fine and you?")
    elif (e.get() == "i'm fine too"):
        text.insert(END, "\n" + "Bot: nice to hear that")
    else:
        text.insert(END, "\n" + "Bot: Sorry I didnt get it.")
text = Text(root,bg='light blue')
text.grid(row=0,column=0,columnspan=2)
e = Entry(root,width=80)
send = Button(root,text='Send',bg='blue',width=20,command=send).grid(row=1,column=1)
e.grid(row=1,column=0)
root = Tk()
root.title('IT SOURCCODE SIMPLE CHATBOT')
root.mainloop()
```

11. Build a virtual assistant for Wikipedia using Wolfram Alpha and Python.

SOURCE CODE

```
import wolframalpha

# Taking input from user
question = input('Question: ')

# App id obtained by the above steps
app_id =('U5HXGG-79KT69H58Q')

# Instance of wolf ram alpha
# client class
client = wolframalpha.Client(app_id)

# Stores the response from
# wolf ram alpha
res = client.query(question)
```

```
# Includes only text from the response
answer = next(res.results).text

print(answer)
```

12. The following is a function that counts the number of times a string occurs in another string:

```
# Count the number of times string s1 is found in string s2
def countsubstring(s1,s2):
count = 0
for i in range(0,len(s2)-len(s1)+1):
if s1 == s2[i:i+len(s1)]:
count += 1
return count
```

For instance, countsubstring('ab','cabalaba') returns 2.

Write a recursive version of the above function. To get the rest of a string (i.e. everything but the first character).

SOURCE CODE

```
# Python3 program to count occurrences of pattern in a text.
def KMPSearch(pat, txt):
    M = len(pat)
    N = len(txt)

    # Create lps[] that will hold the longest prefix suffix values for pattern
    lps = [None] * M
    j = 0  # index for pat[]

    # Preprocess the pattern (calculate lps[]  array)
    computeLPSArray(pat, M, lps)

    i = 0  # index for txt[]
    res = 0
    next_i = 0

    while (i < N):
        if pat[j] == txt[i]:
```

```
            j = j + 1
            i = i + 1
        if j == M:

            # When we find pattern first time,we iterate again to check if there exists more pattern
            j = lps[j - 1]
            res = res + 1

            # We start i to check for more than once appearance of pattern, we will reset i to previous
start+1
            if lps[j] != 0:
                next_i = next_i + 1
                i = next_i
                j = 0

        # Mismatch after j matches
        elif ((i < N) and (pat[j] != txt[i])):

            # Do not match lps[0..lps[j-1]] characters, they will match anyway
            if (j != 0):
                j = lps[j - 1]
            else:
                i = i + 1

    return res

def computeLPSArray(pat, M, lps):

    # Length of the previous longest
    # prefix suffix
    len = 0
    i = 1
    lps[0] = 0  # lps[0] is always 0

    # The loop calculates lps[i] for
    # i = 1 to M-1
    while (i < M):
        if pat[i] == pat[len]:
            len = len + 1
            lps[i] = len
            i = i + 1

        else:  # (pat[i] != pat[len])

            # This is tricky. Consider the example.
            # AAACAAAA and i = 7. The idea is similar
```

```
        # to search step.
        if len != 0:
            len = lps[len - 1]

            # Also, note that we do not increment
            # i here

        else:  # if (len == 0)
            lps[i] = len
            i = i + 1

# Driver code
if __name__ == "__main__":

    txt = "WELCOME TO PYTHON WORLD PYTHON PYTHON"
    pat = "THON"
    ans = KMPSearch(pat, txt)

    print(ans)
```

13. Higher order functions. Write a higher-order function count that counts the number of elements in a list that satisfy a given test. For instance: count(lambda x: x>2, [1,2,3,4,5]) should return 3, as there are three elements in the list larger than 2. Solve this task without using any existing higher-order function.

SOURCE CODE

```
# Python3 program to count occurrences of an element if x is present in arr[] thenreturns the
count of occurrencesof x, otherwise returns -1.
def count(arr, x, n):

    # get the index of first occurrence of x
    i = first(arr, 0, n-1, x, n)

    # If x doesn't exist in arr[] then return -1
    if i == -1:
        return i
```

```
    # Else get the index of last occurrence of x. Note that we are only looking in the subarray after first occurrence
    j = last(arr, i, n-1, x, n);

    # return count
    return j-i+1;

# if x is present in arr[] then return the index of FIRST occurrence of x in arr[0..n-1], otherwise returns -1
def first(arr, low, high, x, n):
    if high >= low:

        # low + (high - low)/2
        mid = (low + high)//2

        if (mid == 0 or x > arr[mid-1]) and arr[mid] == x:
            return mid
        elif x > arr[mid]:
            return first(arr, (mid + 1), high, x, n)
        else:
            return first(arr, low, (mid -1), x, n)
    return -1;

# if x is present in arr[] then return the index of LAST occurrence of x in arr[0..n-1], otherwise returns -1
def last(arr, low, high, x, n):
    if high >= low:

        # low + (high - low)/2
        mid = (low + high)//2;
```

```
    if(mid == n-1 or x < arr[mid+1]) and arr[mid] == x :
        return mid
    elif x < arr[mid]:
        return last(arr, low, (mid -1), x, n)
    else:
        return last(arr, (mid + 1), high, x, n)
  return -1

# driver program to test above functions
arr = [1, 2, 2, 3, 3, 3, 3]
x = 3  # Element to be counted in arr[]
n = len(arr)
c = count(arr, x, n)
print ("%d occurs %d times "%(x, c))
```

OUTPUT

```
3 occurs 4 times
```

14. Brute force solution to the Knapsack problem. Write a function that allows you to generate random problem instances for the knapsack program. This function should generate a list of items containing N items that each have a unique name, a random size in the range 1 ..... 5 and a random value in the range 1 ..... 10.

Next, you should perform performance measurements to see how long the given knapsack solver take to solve different problem sizes. You should peform atleast 10 runs with different randomly generated problem instances for the problem sizes 10,12,14,16,18,20 and 22. Use a

backpack size of 2:5 x N for each value problem size N. Please note that the method used to generate random numbers can also affect performance, since different distributions of values can make the initial conditions of the problem slightly more or less demanding.

How much longer time does it take to run this program when we increase the number of items? Does the backpack size affect the answer?

Try running the above tests again with a backpack size of 1 x N and with 4:0 x N.

SOURCE CODE

```
# Python3 program to solve fractional Knapsack Problem

class ItemValue:

    """Item Value DataClass"""

    def __init__(self, wt, val, ind):
        self.wt = wt
        self.val = val
        self.ind = ind
        self.cost = val // wt

    def __lt__(self, other):
        return self.cost < other.cost

# Greedy Approach

class FractionalKnapSack:

    """Time Complexity O(n log n)"""
    @staticmethod
    def getMaxValue(wt, val, capacity):
        """function to get maximum value """
        iVal = []
        for i in range(len(wt)):
            iVal.append(ItemValue(wt[i], val[i], i))

        # sorting items by value
        iVal.sort(reverse=True)

        totalValue = 0
```

```
    for i in iVal:
        curWt = int(i.wt)
        curVal = int(i.val)
        if capacity - curWt >= 0:
            capacity -= curWt
            totalValue += curVal
        else:
            fraction = capacity / curWt
            totalValue += curVal * fraction
            capacity = int(capacity - (curWt * fraction))
            break
    return totalValue


# Driver Code
if __name__ == "__main__":
    wt = [10, 40, 20, 30]
    val = [60, 40, 100, 120]
    capacity = 50

    # Function call
    maxValue = FractionalKnapSack.getMaxValue(wt, val, capacity)
    print("Maximum value in Knapsack =", maxValue)
```

OUTPUT:-

```
Maximum value in Knapsack = 240.0
```

15. Assume that you are organising a party for N people and have been given a list L of people who, for social reasons, should not sit at the same table. Furthermore, assume that you have C tables (that are infinitly large).

Write a function layout(N,C,L) that can give a table placement (ie. a number from 0 : : :C - 1) for each guest such that there will be no social mishaps.

For simplicity we assume that you have a unique number 0 .....N-1 for each guest and that the list of restrictions is of the form [(X,Y), ...] denoting guests X, Y that are not allowed to sit together. Answer with a dictionary mapping each guest into a table assignment, if there are no possible layouts of the guests you should answer False.

SOURCE CODE

```
# Python3 program to solve fractional Knapsack Problem

class ItemValue:

    """Item Value DataClass"""

    def __init__(self, wt, val, ind):
        self.wt = wt
        self.val = val
        self.ind = ind
        self.cost = val // wt

    def __lt__(self, other):
        return self.cost < other.cost

# Greedy Approach
class FractionalKnapSack:

    """Time Complexity O(n log n)"""
    @staticmethod
    def getMaxValue(wt, val, capacity):
        """function to get maximum value """
        iVal = []
        for i in range(len(wt)):
            iVal.append(ItemValue(wt[i], val[i], i))
```

```
    # sorting items by value
    iVal.sort(reverse=True)

    totalValue = 0
    for i in iVal:
        curWt = int(i.wt)
        curVal = int(i.val)
        if capacity - curWt >= 0:
            capacity -= curWt
            totalValue += curVal
        else:
            fraction = capacity / curWt
            totalValue += curVal * fraction
            capacity = int(capacity - (curWt * fraction))
            break
    return totalValue


# Driver Code
if __name__ == "__main__":
    wt = [10, 40, 20, 30]
    val = [60, 40, 100, 120]
    capacity = 50

    # Function call
    maxValue = FractionalKnapSack.getMaxValue(wt, val, capacity)
    print("Maximum value in Knapsack =", maxValue)
```

OUTPUT:-

```
Maximum value in Knapsack = 240.0
```