

Name:-

Roll No.:-

Div:-

Title:- Implement DFS and BFS Algorithm. Use an Undirected Graph and develop a Recursive Algorithm for searching all the vertices of the graph or tree data structure.

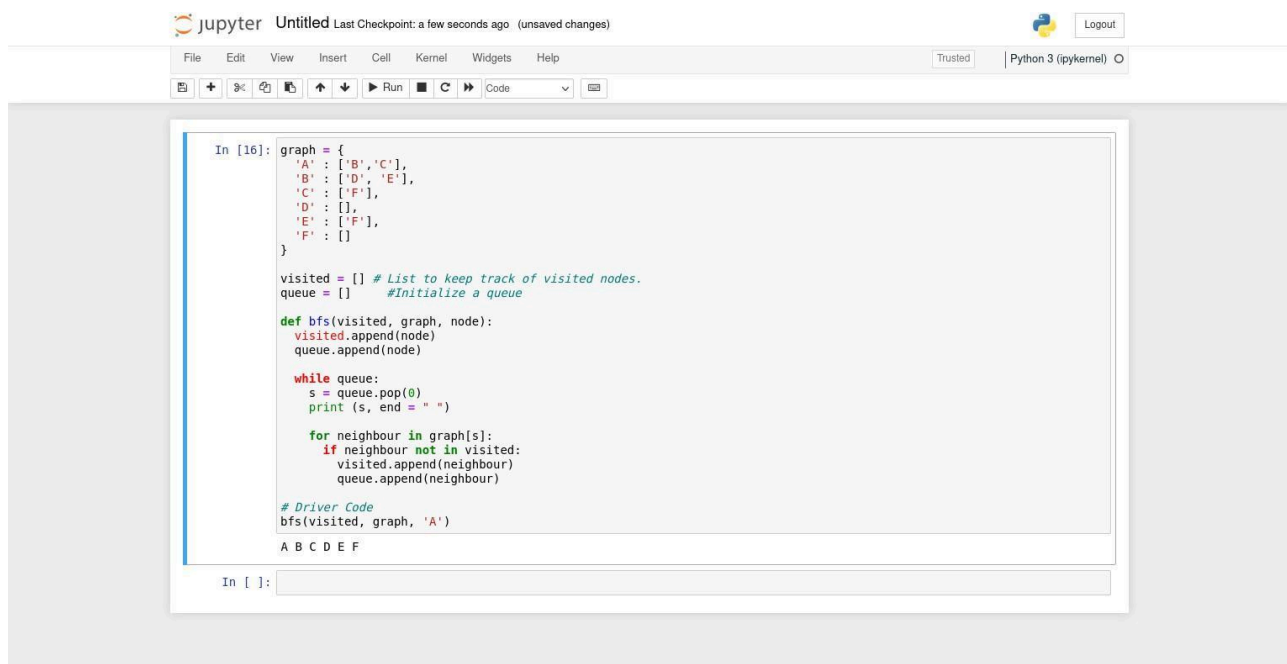
Program :-

Breadth First Search(BFS):-

```
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

print ("Hello")
def bfs(visited, graph, node):
    visited.append(node)
    queue.append(node)
    while queue:
        s = queue.pop(0)
        print (s, end = " ")
        print ("Hello")

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



```
In [16]: 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)

# Driver Code
bfs(visited, graph, 'A')

A B C D E F
```

2. Depth-first Search:

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 of graph. def

def dfs(visited, graph, node): #function for dfs

if node not in visited:

print (node)

visited.add(node)

for neighbour in graph[node]:

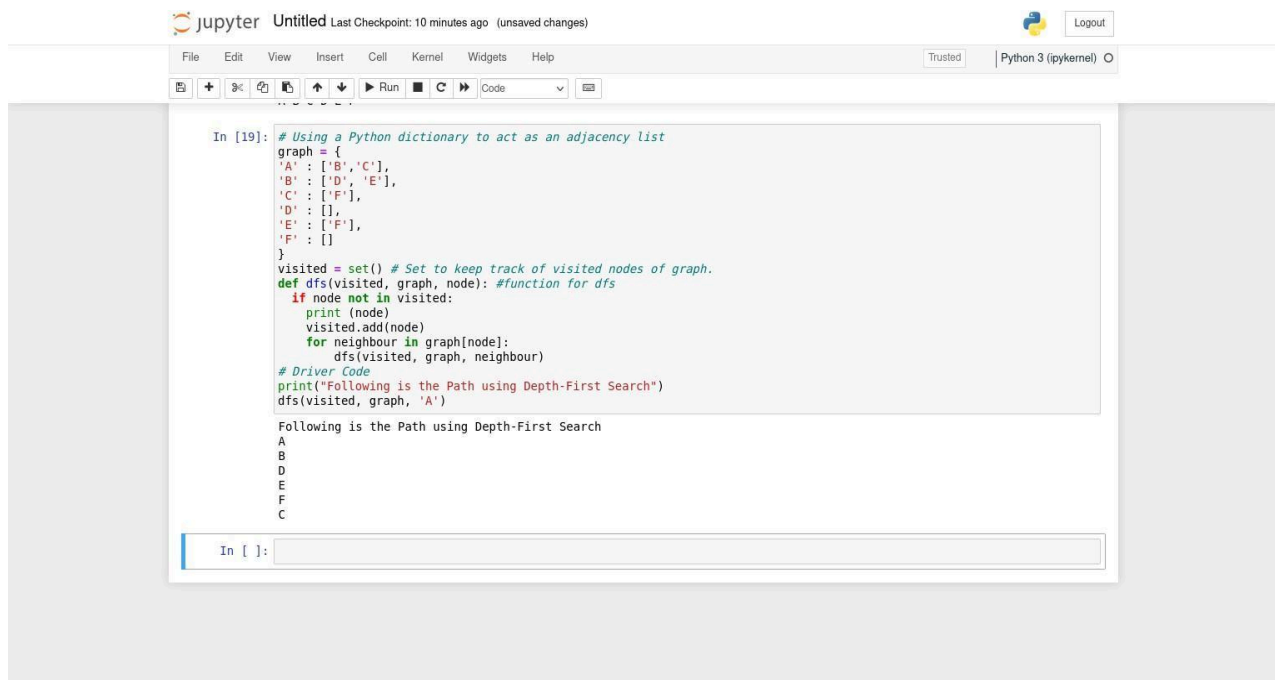
dfs(visited, graph, neighbour)

print("Following is the Path using Depth-First Search")

dfs(visited, graph, 'A')

Driver Code

```
print("Following is the Path using Depth-First Search")  
dfs(visited, graph, 'A')
```



The image shows a Jupyter Notebook interface with a single code cell. The code implements a Depth-First Search (DFS) algorithm on a graph. The graph is represented as a dictionary where keys are nodes and values are lists of adjacent nodes. The nodes are 'A', 'B', 'C', 'D', 'E', and 'F'. The edges are: A-B, A-C, B-D, B-E, C-F. The DFS starts at node 'A' and visits nodes in the order: A, B, D, E, F, C. The output of the code is printed in the cell's output area.

```
In [19]: # 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 of graph.  
def dfs(visited, graph, node): #function for dfs  
    if node not in visited:  
        print (node)  
        visited.add(node)  
        for neighbour in graph[node]:  
            dfs(visited, graph, neighbour)  
  
# Driver Code  
print("Following is the Path using Depth-First Search")  
dfs(visited, graph, 'A')  
  
Following is the Path using Depth-First Search  
A  
B  
D  
E  
F  
C
```

In []:

Name:-

Roll No.:-

Div:-

Title:- Implement A star Algorithm for any game search problem

Program :-

A* Algorithm

```
from collections import deque
class Graph:
# example of adjacency list (or rather map)
    # adjacency_list = {
    # 'A': [('B', 1), ('C', 3), ('D', 7)], # 'B': [('D', 5)],
    # 'C': [('D', 12)] # }

    def __init__(self, adjacency_list):
        self.adjacency_list = adjacency_list

    def get_neighbors(self, v):
        print("a")
        return self.adjacency_list[v]

    def h(self, n):
        H = {
            'A': 1,
            'B': 1,
            'C': 1,
            'D': 1
        }
        return H[n]

    def a_star_algorithm(self, start_node, stop_node):
        print("b")
        open_list = set([start_node])
        closed_list = set([])
        g = {}
        g[start_node] = 0
        parents = {}
        parents[start_node] = start_node
        while len(open_list) > 0:
```

```

n = None
for v in open_list:
    if n == None or g[v] + self.h(v) < g[n] + self.h(n):
        n = v;
if n == None:
    print('Path does not exist!')
    return None
if n == stop_node:
    reconst_path = []
    while parents[n] != n:
        reconst_path.append(n)
        n = parents[n]
    reconst_path.append(start_node)
    reconst_path.reverse()
    print('Path found: {}'.format(reconst_path))
    return reconst_path

for (m, weight) in self.get_neighbors(n):
    if m not in open_list and m not in closed_list:
        open_list.add(m)
        parents[m] = n
        g[m] = g[n] + weight
    else:
        if g[m] > g[n] + weight:
            g[m] = g[n] + weight
            parents[m] = n
            if m in closed_list:
                closed_list.remove(m)
            open_list.add(m)
open_list.remove(n)
closed_list.add(n)
print('Path does not exist!')
return None

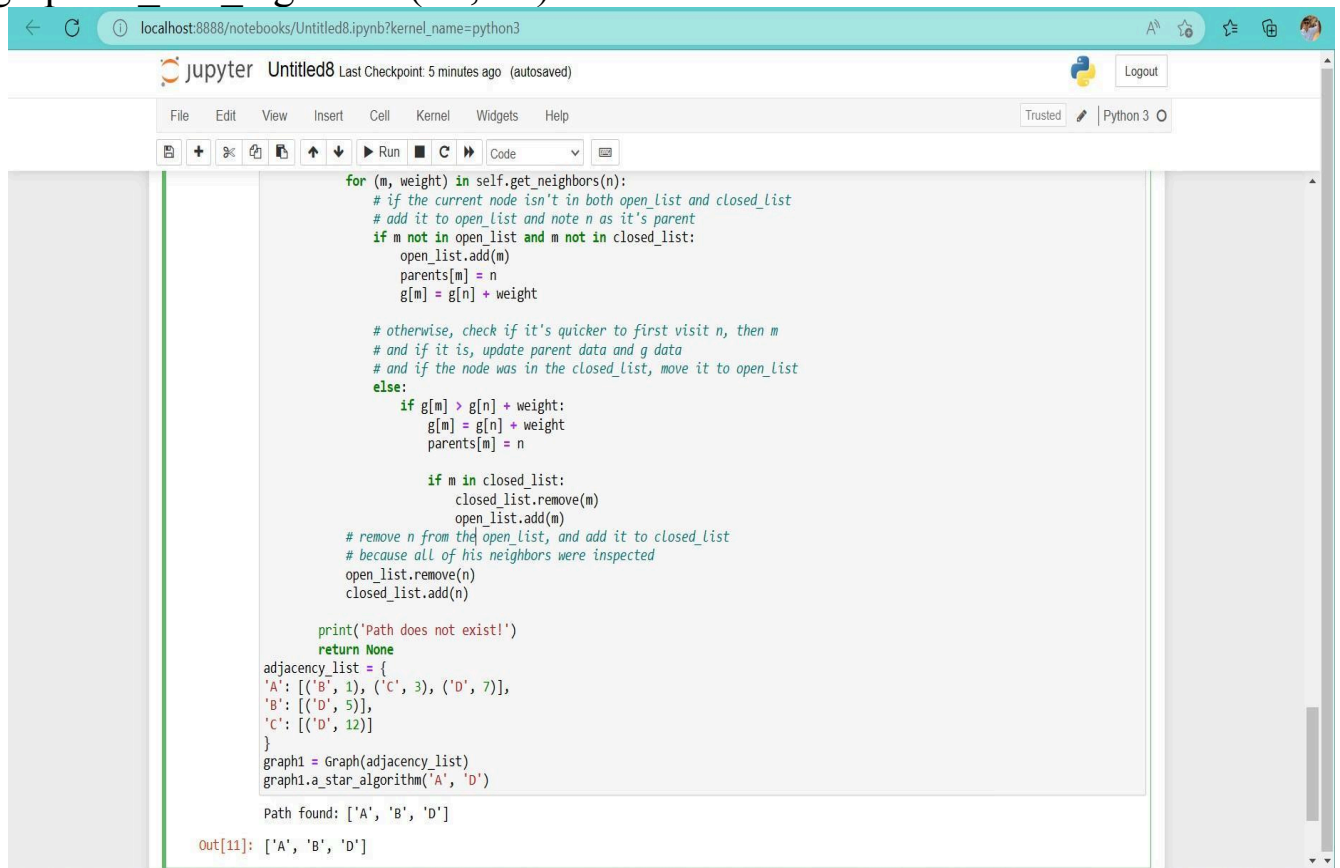
```

```

adjacency_list = {
'A': [('B', 1), ('C', 3), ('D', 7)],
'B': [('D', 5)],
'C': [('D', 12)]
}
graph1 = Graph(adjacency_list)

```

graph1.a_star_algorithm('A', 'D')



The image shows a Jupyter Notebook interface with a teal header bar. The address bar shows 'localhost:8888/notebooks/Untitled8.ipynb?kernel_name=python3'. The notebook title is 'Untitled8' with a 'Last Checkpoint: 5 minutes ago (autosaved)' status. The interface includes a menu bar (File, Edit, View, Insert, Cell, Kernel, Widgets, Help) and a toolbar with icons for file operations, running, and code execution. The main area contains a Python code cell with an A* algorithm implementation. The code defines a graph with nodes A, B, and C, and their neighbors with weights. It implements the A* search algorithm, finding the shortest path from A to D. The output shows the path found: ['A', 'B', 'D'].

```
for (m, weight) in self.get_neighbors(n):
    # if the current node isn't in both open_list and closed_list
    # add it to open_list and note n as it's parent
    if m not in open_list and m not in closed_list:
        open_list.add(m)
        parents[m] = n
        g[m] = g[n] + weight

    # otherwise, check if it's quicker to first visit n, then m
    # and if it is, update parent data and g data
    # and if the node was in the closed_list, move it to open_list
    else:
        if g[m] > g[n] + weight:
            g[m] = g[n] + weight
            parents[m] = n

            if m in closed_list:
                closed_list.remove(m)
                open_list.add(m)
        # remove n from the open_list, and add it to closed_list
        # because all of his neighbors were inspected
        open_list.remove(n)
        closed_list.add(n)

    print('Path does not exist!')
    return None

adjacency_list = {
    'A': [('B', 1), ('C', 3), ('D', 7)],
    'B': [('D', 5)],
    'C': [('D', 12)]
}

graph1 = Graph(adjacency_list)
graph1.a_star_algorithm('A', 'D')

Path found: ['A', 'B', 'D']

Out[11]: ['A', 'B', 'D']
```

Name:-

Roll No.:-

Div:-

Title:- Implement Greedy search algorithm for selection sort

Program :-

Greedy search Algorithm for selection sort program

```
# Selection sort in Python
# time complexity O(n*n)
# sorting by finding min_index
def selectionSort(array, size):

    for ind in range(size):
        min_index = ind

        for j in range(ind + 1, size):

            # select the minimum element in every iteration
            if array[j] < array[min_index]:
                min_index = j

            # swapping the elements to sort the array
            (array[ind], array[min_index]) = (array[min_index], array[ind])

arr = [-2, 45, 0, 11, -9, 88, -97, -202, 747]

size = len(arr)
selectionSort(arr, size)
print('The array after sorting in Ascending Order by selection sort is:')

print(arr)
```

localhost:8888/notebooks/Untitled9.ipynb?kernel_name=python3

jupyter Untitled9 Last Checkpoint: 3 minutes ago (autosaved) Logout

File Edit View Insert Cell Kernel Widgets Help Trusted Python 3

In [1]:

```
# Selection sort in Python
# time complexity O(n*n)
# sorting by finding min_index
def selectionSort(array, size):

    for ind in range(size):
        min_index = ind

        for j in range(ind + 1, size):
            # select the minimum element in every iteration
            if array[j] < array[min_index]:
                min_index = j
            # swapping the elements to sort the array
            (array[ind], array[min_index]) = (array[min_index], array[ind])

arr = [-2, 45, 0, 11, -9, 88, -97, -202, 747]
size = len(arr)
selectionSort(arr, size)
print('The array after sorting in Ascending Order by selection sort is:')
print(arr)
```

The array after sorting in Ascending Order by selection sort is:
[-202, -97, -9, -2, 0, 11, 45, 88, 747]

In []:

Name:-

Roll No.:-

Div:-

Title:- Implement a solution for a constraint satisfaction problem using branch and bound and backtracking for n-queens problem or a graph coloring problem

Program :-

n-queens problem

```
# Python program to solve N Queen
# Problem using backtracking
```

```
global N
N = 4
```

```
def printSolution(board):
    for i in range(N):
        for j in range(N):
            print (board[i][j],end=' ')
        print()
```

```
# A utility function to check if a queen can
# be placed on board[row][col]. Note that this
# function is called when "col" queens are
# already placed in columns from 0 to col -1.
# So we need to check only left side for
# attacking queens
```

```
def isSafe(board, row, col):
```

```
    # Check this row on left side
    for i in range(col):
        if board[row][i] == 1:
            return False
```

```
    # Check upper diagonal on left side
```

```
for i, j in zip(range(row, -1, -1), range(col, -1, -1)):
    if board[i][j] == 1:
        return False
```

```
# Check lower diagonal on left side
```

```
for i, j in zip(range(row, N, 1), range(col, -1, -1)):
    if board[i][j] == 1:
        return False
```

```
return True
```

```
def solveNQUtil(board, col):
```

```
    # base case: If all queens are
    # placed # then return true
    if col >= N:
        return True
```

```
    # Consider this column and try placing
    # this queen in all rows one by one
    for i in range(N):
```

```
        if isSafe(board, i, col):
            # Place this queen in board[i][col]
            board[i][col] = 1
```

```
        # recur to place rest of the queens
```

```
        if solveNQUtil(board, col + 1) == True:
            return True
```

```
        # If placing queen in board[i][col]
        # doesn't lead to a solution, then
        # queen from board[i][col]
        board[i][col] = 0
```

```
    # if the queen can not be placed in any row in
    # this column col then return false
    return False
```

```
# This function solves the N Queen problem using
```

```
# Backtracking. It mainly uses solveNQUtil() to
# solve the problem. It returns false if queens
# cannot be placed, otherwise return true and
# placement of queens in the form of 1s.
# note that there may be more than one
```

```
# solutions, this function prints one of the
# feasible solutions.
```

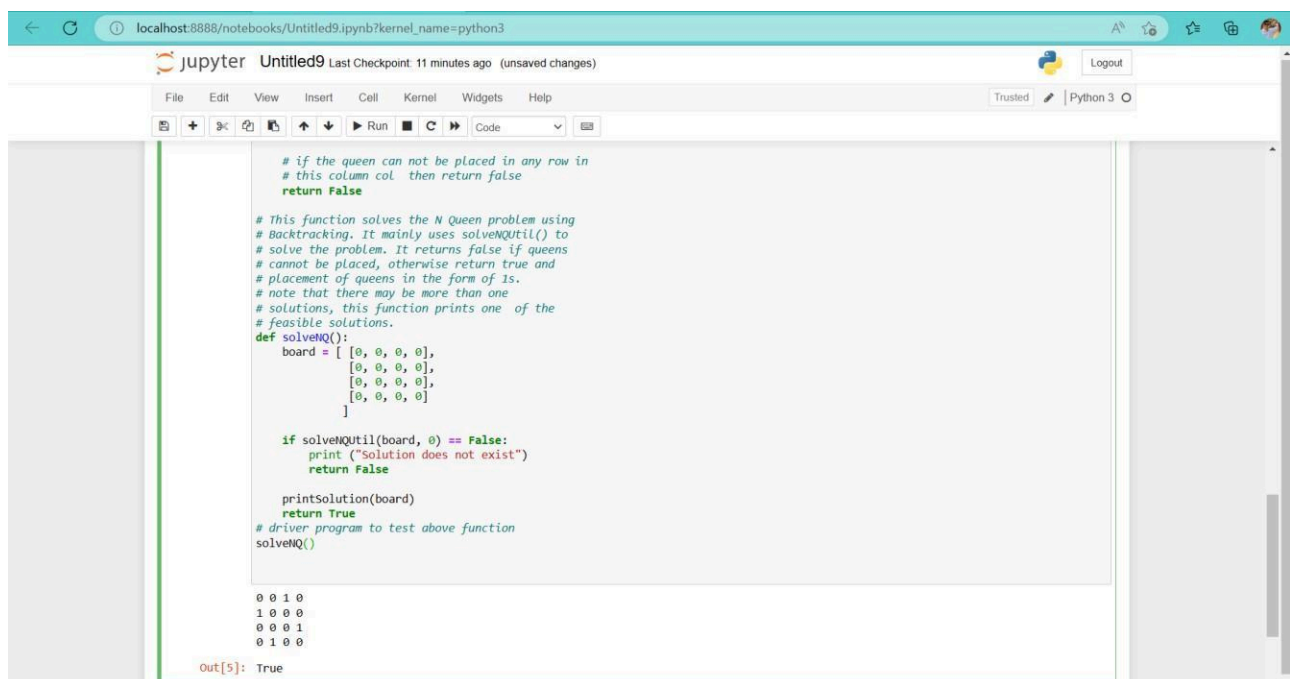
```
def solveNQ():
```

```
    board = [ [0, 0, 0, 0],
               [0, 0, 0, 0],
               [0, 0, 0, 0],
               [0, 0, 0, 0]
             ]
```

```
    if solveNQUtil(board, 0) == False:
        print ("Solution does not exist")
        return False
```

```
    printSolution(board)
    return True
```

```
# driver program to test above function
solveNQ()
```



The screenshot shows a Jupyter Notebook titled 'Untitled9' running on a local host. The code cell contains the implementation of the N-Queens problem using backtracking. The board is initialized as a 4x4 grid of zeros. The function `solveNQ()` calls `solveNQUtil(board, 0)`. Since the board is all zeros, the utility function returns `False`, and the program prints 'Solution does not exist'. The output cell shows the result: `Out[5]: True`.

```
# if the queen can not be placed in any row in
# this column col then return false
return False

# This function solves the N Queen problem using
# Backtracking. It mainly uses solveNQUtil() to
# solve the problem. It returns false if queens
# cannot be placed, otherwise return true and
# placement of queens in the form of 1s.
# note that there may be more than one
# solutions, this function prints one of the
# feasible solutions.
def solveNQ():
    board = [ [0, 0, 0, 0],
               [0, 0, 0, 0],
               [0, 0, 0, 0],
               [0, 0, 0, 0]
             ]

    if solveNQUtil(board, 0) == False:
        print ("Solution does not exist")
        return False

    printSolution(board)
    return True
# driver program to test above function
solveNQ()
```

```
0 0 1 0
1 0 0 0
0 0 0 1
0 1 0 0

Out[5]: True
```

Name:-

Roll No.:-

Div:-

Title:- Develop an elementary chatbot for any suitable customer interaction application

Program :-

Chatbot Program

```
def greet(bot_name, birth_year):  
    print("Hello! My name is {0}.".format(bot_name))  
    print("I was created in {0}.".format(birth_year))
```

```
def remind_name():  
    print('Please, remind me your name.')  
    name = input()  
    print("What a great name you have, {0}!".format(name))
```

```
def guess_age():  
    print('Let me guess your age.')  
    print('Enter remainders of dividing your age by 3, 5 and 7.')  
  
    rem3 =  
int(input())    rem5  
= int(input())  
rem7 =  
int(input())  
age = (rem3 * 70 + rem5 * 21 + rem7 * 15) % 105  
  
    print("Your age is {0}; that's a good time to start  
programming!".format(age))
```

```
def count():  
    print('Now I will prove to you that I can count to any number you
```

want.')

num = int(input())

```
counter = 0
```

```
while counter <= num:
```

```
    print(" {0} !".format(counter))
```

```
    counter += 1
```

```
def test():
```

```
    print("Let's test your programming knowledge.")
```

```
    print("Why do we use methods?")
```

```
    print("1. To repeat a statement multiple times.")
```

```
    print("2. To decompose a program into several small
```

```
    subroutines.") print("3. To determine the execution time of a
```

```
    program.")
```

```
    print("4. To interrupt the execution of a program.")
```

```
    answer = 2
```

```
    guess = int(input())
```

```
    while guess != answer:
```

```
        print("Please, try again.")
```

```
        guess = int(input())
```

```
    print('Completed, have a nice
```

```
    day!') print('                ')
```

```
    print('                ')
```

```
    print('                ')
```

```
def end():
```

```
    print('Congratulations, have a nice
```

```
    day!') print('                ')
```

```
    print('                ')
```

```
    print('                ')
```

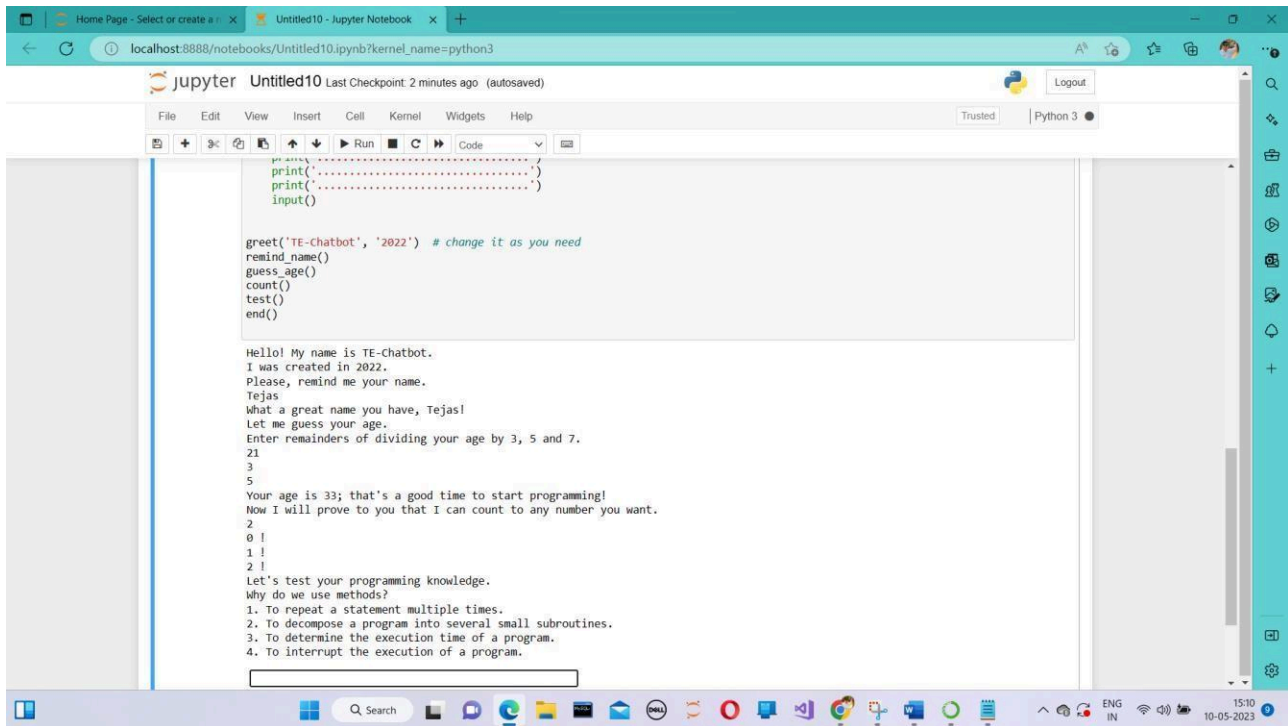
```
    input()
```

```
greet('TE-Chatbot', '2022') # change it as you need
```

```
remind_name()
```

```
guess_age()  
count()
```

test()
end()



The screenshot shows a Jupyter Notebook titled 'Untitled10' running on a local host. The notebook contains a Python script for a chatbot named 'TE-Chatbot'. The script includes functions for greeting, reminding of a name, guessing age, counting, testing, and ending. The output of the script is displayed below the code, showing the chatbot's interactions with the user.

```
print('.....')
print('.....')
print('.....')
Input()

greet('TE-Chatbot', '2022') # change it as you need
remind_name()
guess_age()
count()
test()
end()
```

Hello! My name is TE-Chatbot.
I was created in 2022.
Please, remind me your name.
Tejas
What a great name you have, Tejas!
Let me guess your age.
Enter remainders of dividing your age by 3, 5 and 7.
21
3
5
Your age is 33; that's a good time to start programming!
Now I will prove to you that I can count to any number you want.
2
0!
1!
2!
Let's test your programming knowledge.
Why do we use methods?
1. To repeat a statement multiple times.
2. To decompose a program into several small subroutines.
3. To determine the execution time of a program.
4. To interrupt the execution of a program.

Name:-

Roll No.:-

Div:-

Title:- Implement of Expert system Help Desk Management System

Program :-

Help Desk Management System

Define a dictionary of common problems and their solutions

```
problem_dict = {  
    "Printer not working": "Check that it's turned on and connected to the  
network",  
    "Can't log in": "Make sure you're using the correct username and  
password",  
    "Software not installing": "Check that your computer meets the system  
requirements",  
    "Internet connection not working": "Restart your modem or router",  
    "Email not sending": "Check that you're using the correct email server  
settings"  
}
```

Define a function to handle user requests

```
def handle_request(user_input):  
    if user_input.lower() == "exit":  
        return "Goodbye!"  
    elif user_input in problem_dict:  
        return problem_dict[user_input]  
    else:  
        return "I'm sorry, I don't know how to help with that problem."
```

Main loop to prompt user for input

```
while True:  
    user_input = input("What's the problem? Type 'exit' to quit. ")  
    response = handle_request(user_input)  
    print(response)
```

Output:-

```
What's the problem? Type 'exit' to quit. Printer not working
Check that it's turned on and connected to the network
What's the problem? Type 'exit' to quit. Can't log in
Make sure you're using the correct username and password
What's the problem? Type 'exit' to quit. Software not installing
Check that your computer meets the system requirements
What's the problem? Type 'exit' to quit. Internet connection not working
Restart your modem or router
What's the problem? Type 'exit' to quit. Email not sending
I'm sorry, I don't know how to help with that problem.
What's the problem? Type 'exit' to quit. exit
Goodbye!
What's the problem? Type 'exit' to quit. █
```