**GROUP A**

**ASSIGNMENT NO. 1**

**1.** Implement depth first search algorithm and Breadth First Search algorithm, Use an undirected graph and develop a recursive algorithm for searching all the vertices of a graph or tree data structure.

**SOLN :**

**CODE :**

**BFS :**

import java.io.\*;

import java.util.\*;

public class BFSTraversal

{

private int vertex; /\* total number number of vertices in the graph \*/

private LinkedList<Integer> adj[]; /\* adjacency list \*/

private Queue<Integer> que; /\* maintaining a queue \*/

BFSTraversal(int v)

{

vertex = v;

adj = new LinkedList[vertex];

for (int i=0; i<v; i++)

{

adj[i] = new LinkedList<>();

}

que = new LinkedList<Integer>();

}

void insertEdge(int v,int w)

{

adj[v].add(w); /\* adding an edge to the adjacency list (edges are bidirectional in this example) \*/

}

void BFS(int n)

{

boolean nodes[] = new boolean[vertex]; /\* initialize boolean array for holding the data \*/

int a = 0;

nodes[n]=true;

que.add(n); /\* root node is added to the top of the queue \*/

while (que.size() != 0)

{

n = que.poll(); /\* remove the top element of the queue \*/

System.out.print(n+" "); /\* print the top element of the queue \*/

for (int i = 0; i < adj[n].size(); i++) /\* iterate through the linked list and push all neighbors into queue \*/

{

a = adj[n].get(i);

if (!nodes[a]) /\* only insert nodes into queue if they have not been explored already \*/

{

nodes[a] = true;

que.add(a);

}

}

}

}

public static void main(String args[])

{

BFSTraversal graph = new BFSTraversal(10);

graph.insertEdge(0, 1);

graph.insertEdge(0, 2);

graph.insertEdge(0, 3);

graph.insertEdge(1, 3);

graph.insertEdge(2, 4);

graph.insertEdge(3, 5);

graph.insertEdge(3, 6);

graph.insertEdge(4, 7);

graph.insertEdge(4, 5);

graph.insertEdge(5, 2);

graph.insertEdge(6, 5);

graph.insertEdge(7, 5);

graph.insertEdge(7, 8);

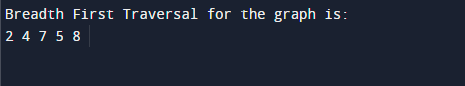
System.out.println("Breadth First Traversal for the graph is:");

graph.BFS(2);

}

}

**OUTPUT :**



DFS :

// Java program to print DFS

// mtraversal from a given given

// graph

import java.io.\*;

import java.util.\*;

// This class represents a

// directed graph using adjacency

// list representation

class Graph {

private int V; // No. of vertices

// Array of lists for

// Adjacency List Representation

private LinkedList<Integer> adj[];

// Constructor

@SuppressWarnings("unchecked") Graph(int v)

{

V = v;

adj = new LinkedList[v];

for (int i = 0; i < v; ++i)

adj[i] = new LinkedList();

}

// Function to add an edge into the graph

void addEdge(int v, int w)

{

adj[v].add(w); // Add w to v's list.

}

// A function used by DFS

void DFSUtil(int v, boolean visited[])

{

// Mark the current node as visited and print it

visited[v] = true;

System.out.print(v + " ");

// Recur for all the vertices adjacent to this

// vertex

Iterator<Integer> i = adj[v].listIterator();

while (i.hasNext()) {

int n = i.next();

if (!visited[n])

DFSUtil(n, visited);

}

}

// The function to do DFS traversal.

// It uses recursive

// DFSUtil()

void DFS(int v)

{

// Mark all the vertices as

// not visited(set as

// false by default in java)

boolean visited[] = new boolean[V];

// Call the recursive helper

// function to print DFS

// traversal

DFSUtil(v, visited);

}

// Driver Code

public static void main(String args[])

{

Graph g = new Graph(4);

g.addEdge(0, 1);

g.addEdge(0, 2);

g.addEdge(1, 2);

g.addEdge(2, 0);

g.addEdge(2, 3);

g.addEdge(3, 3);

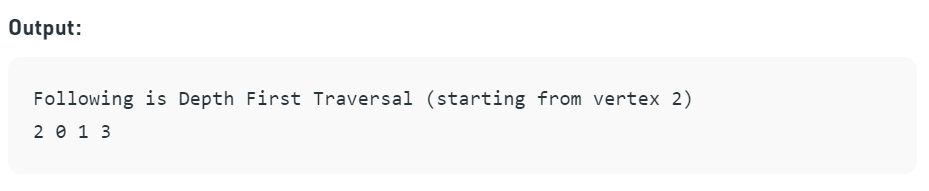
System.out.println(

"Following is Depth First Traversal "

+ "(starting from vertex 2)");

g.DFS(2); }}

**OUTPUT :**



**GROUP A**

**ASSIGNMENT NO. 2**

2. Implement A star Algorithm for any game search

problem.

**SOLN :**

**CODE :**

def aStarAlgo(start\_node, stop\_node):

        open\_set = set(start\_node)

        closed\_set = set()

        g = {} #store distance from starting node

        parents = {}# parents contains an adjacency map of all nodes

        #ditance of starting node from itself is zero

        g[start\_node] = 0

        #start\_node is root node i.e it has no parent nodes

        #so start\_node is set to its own parent node

        parents[start\_node] = start\_node

        while len(open\_set) > 0:

            n = None

            #node with lowest f() is found

            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):

                    #nodes 'm' not in first and last set are added to first

                    #n is set its parent

                    if m not in open\_set and m not in closed\_set:

                        open\_set.add(m)

                        parents[m] = n

                        g[m] = g[n] + weight

                    #for each node m,compare its distance from start i.e g(m) to the

                    #from start through n node

                    else:

                        if g[m] > g[n] + weight:

                            #update g(m)

                            g[m] = g[n] + weight

                            #change parent of m to n

                            parents[m] = n

                            #if m in closed set,remove and add to open

                            if m in closed\_set:

                                closed\_set.remove(m)

                                open\_set.add(m)

            if n == None:

                print('Path does not exist!')

                return None

            # if the current node is the stop\_node

            # then we begin reconstructin the path from it to the start\_node

            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

            # remove n from the open\_list, and add it to closed\_list

            # because all of his neighbors were inspected

            open\_set.remove(n)

            closed\_set.add(n)

        print('Path does not exist!')

        return None

#define fuction to return neighbor and its distance

#from the passed node

def get\_neighbors(v):

    if v in Graph\_nodes:

        return Graph\_nodes[v]

    else:

        return None

#for simplicity we ll consider heuristic distances given

#and this function returns heuristic distance for all nodes

def heuristic(n):

        H\_dist = {

            'A': 11,

            'B': 6,

            'C': 99,

            'D': 1,

            'E': 7,

            'G': 0,

        }

        return H\_dist[n]

#Describe your graph here

Graph\_nodes = {

    'A': [('B', 2), ('E', 3)],

    'B': [('C', 1),('G', 9)],

    'C': None,

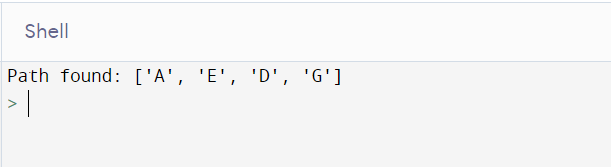
    'E': [('D', 6)],

    'D': [('G', 1)],

}

aStarAlgo('A', 'G')

**OUTPUT :**



**GROUP A**

**ASSIGNMENT NO. 3**

**3.** Implement Greedy search algorithm for Kruskal's Minimal Spanning Tree Algorithm.

**SOLN :**

**CODE :**

import java.util.ArrayList;

import java.util.Comparator;

import java.util.PriorityQueue;

public class KrushkalMST {

static class Edge {

int source;

int destination;

int weight;

public Edge(int source, int destination, int weight) {

this.source = source;

this.destination = destination;

this.weight = weight;

}

}

static class Graph {

int vertices;

ArrayList<Edge> allEdges = new ArrayList<>();

Graph(int vertices) {

this.vertices = vertices;

}

public void addEgde(int source, int destination, int weight) {

Edge edge = new Edge(source, destination, weight);

allEdges.add(edge); //add to total edges

}

public void kruskalMST(){

PriorityQueue<Edge> pq = new PriorityQueue<>(allEdges.size(), Comparator.comparingInt(o -> o.weight));

//add all the edges to priority queue, //sort the edges on weights

for (int i = 0; i <allEdges.size() ; i++) {

pq.add(allEdges.get(i));

}

//create a parent []

int [] parent = new int[vertices];

//makeset

makeSet(parent);

ArrayList<Edge> mst = new ArrayList<>();

//process vertices - 1 edges

int index = 0;

while(index<vertices-1){

Edge edge = pq.remove();

//check if adding this edge creates a cycle

int x\_set = find(parent, edge.source);

int y\_set = find(parent, edge.destination);

if(x\_set==y\_set){

//ignore, will create cycle

}else {

//add it to our final result

mst.add(edge);

index++;

union(parent,x\_set,y\_set);

}

}

//print MST

System.out.println("Minimum Spanning Tree: ");

printGraph(mst);

}

public void makeSet(int [] parent){

//Make set- creating a new element with a parent pointer to itself.

for (int i = 0; i <vertices ; i++) {

parent[i] = i;

}

}

public int find(int [] parent, int vertex){

//chain of parent pointers from x upwards through the tree

// until an element is reached whose parent is itself

if(parent[vertex]!=vertex)

return find(parent, parent[vertex]);;

return vertex;

}

public void union(int [] parent, int x, int y){

int x\_set\_parent = find(parent, x);

int y\_set\_parent = find(parent, y);

//make x as parent of y

parent[y\_set\_parent] = x\_set\_parent;

}

public void printGraph(ArrayList<Edge> edgeList){

for (int i = 0; i <edgeList.size() ; i++) {

Edge edge = edgeList.get(i);

System.out.println("Edge-" + i + " source: " + edge.source +

" destination: " + edge.destination +

" weight: " + edge.weight);

}

}

}

public static void main(String[] args) {

int vertices = 6;

Graph graph = new Graph(vertices);

graph.addEgde(0, 1, 4);

graph.addEgde(0, 2, 3);

graph.addEgde(1, 2, 1);

graph.addEgde(1, 3, 2);

graph.addEgde(2, 3, 4);

graph.addEgde(3, 4, 2);

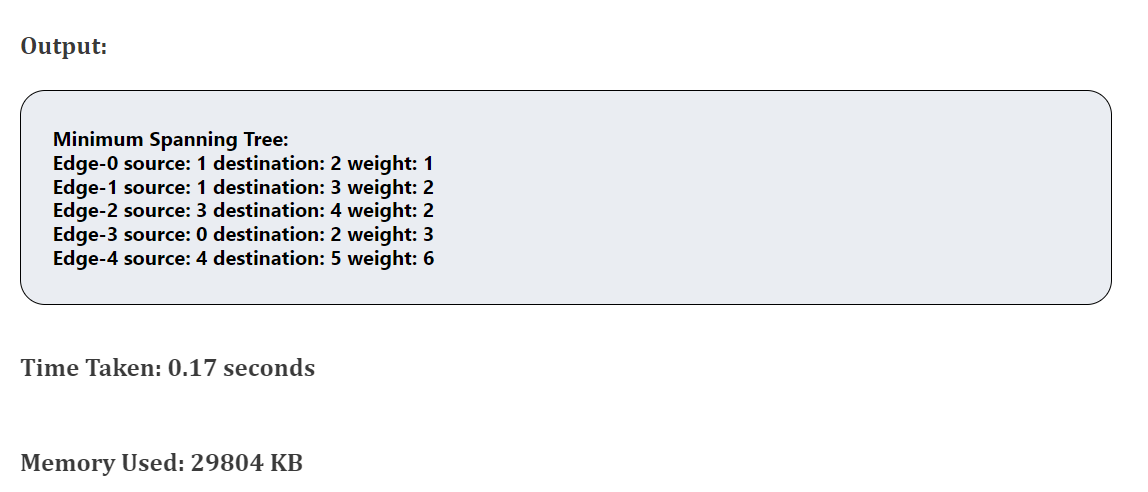
graph.addEgde(4, 5, 6);

graph.kruskalMST();

}

}

**OUTPUT :**



**GROUP B**

**ASSIGNMENT NO. 4**

**4.** Implement a solution for a Constraint Satisfaction Problem using Branch and Bound and Backtracking for n-queens problem or a graph coloring problem.

**SOLN :**

**CODE :**

**BRANCH AND BOUND FOR N-QUEENS :**

""" Python program to solve N Queen Problem

using Branch or Bound """

N = 8

""" A utility function to print solution """

def printSolution(board):

for i in range(N):

for j in range(N):

print(board[i][j], end = " ")

print()

""" A Optimized function to check if

a queen can be placed on board[row][col] """

def isSafe(row, col, slashCode, backslashCode,

rowLookup, slashCodeLookup,

backslashCodeLookup):

if (slashCodeLookup[slashCode[row][col]] or

backslashCodeLookup[backslashCode[row][col]] or

rowLookup[row]):

return False

return True

""" A recursive utility function

to solve N Queen problem """

def solveNQueensUtil(board, col, slashCode, backslashCode,

rowLookup, slashCodeLookup,

backslashCodeLookup):

""" base case: If all queens are

placed then return True """

if(col >= N):

return True

for i in range(N):

if(isSafe(i, col, slashCode, backslashCode,

rowLookup, slashCodeLookup,

backslashCodeLookup)):

""" Place this queen in board[i][col] """

board[i][col] = 1

rowLookup[i] = True

slashCodeLookup[slashCode[i][col]] = True

backslashCodeLookup[backslashCode[i][col]] = True

""" recur to place rest of the queens """

if(solveNQueensUtil(board, col + 1,

slashCode, backslashCode,

rowLookup, slashCodeLookup,

backslashCodeLookup)):

return True

""" If placing queen in board[i][col]

doesn't lead to a solution,then backtrack """

""" Remove queen from board[i][col] """

board[i][col] = 0

rowLookup[i] = False

slashCodeLookup[slashCode[i][col]] = False

backslashCodeLookup[backslashCode[i][col]] = False

""" If queen can not be place in any row in

this column col then return False """

return False

""" This function solves the N Queen problem using

Branch or Bound. It mainly uses solveNQueensUtil()to

solve the problem. It returns False if queens

cannot be placed,otherwise return True or

prints placement of queens in the form of 1s.

Please note that there may be more than one

solutions,this function prints one of the

feasible solutions."""

def solveNQueens():

board = [[0 for i in range(N)]

for j in range(N)]

# helper matrices

slashCode = [[0 for i in range(N)]

for j in range(N)]

backslashCode = [[0 for i in range(N)]

for j in range(N)]

# arrays to tell us which rows are occupied

rowLookup = [False] \* N

# keep two arrays to tell us

# which diagonals are occupied

x = 2 \* N - 1

slashCodeLookup = [False] \* x

backslashCodeLookup = [False] \* x

# initialize helper matrices

for rr in range(N):

for cc in range(N):

slashCode[rr][cc] = rr + cc

backslashCode[rr][cc] = rr - cc + 7

if(solveNQueensUtil(board, 0, slashCode, backslashCode,

rowLookup, slashCodeLookup,

backslashCodeLookup) == False):

print("Solution does not exist")

return False

# solution found

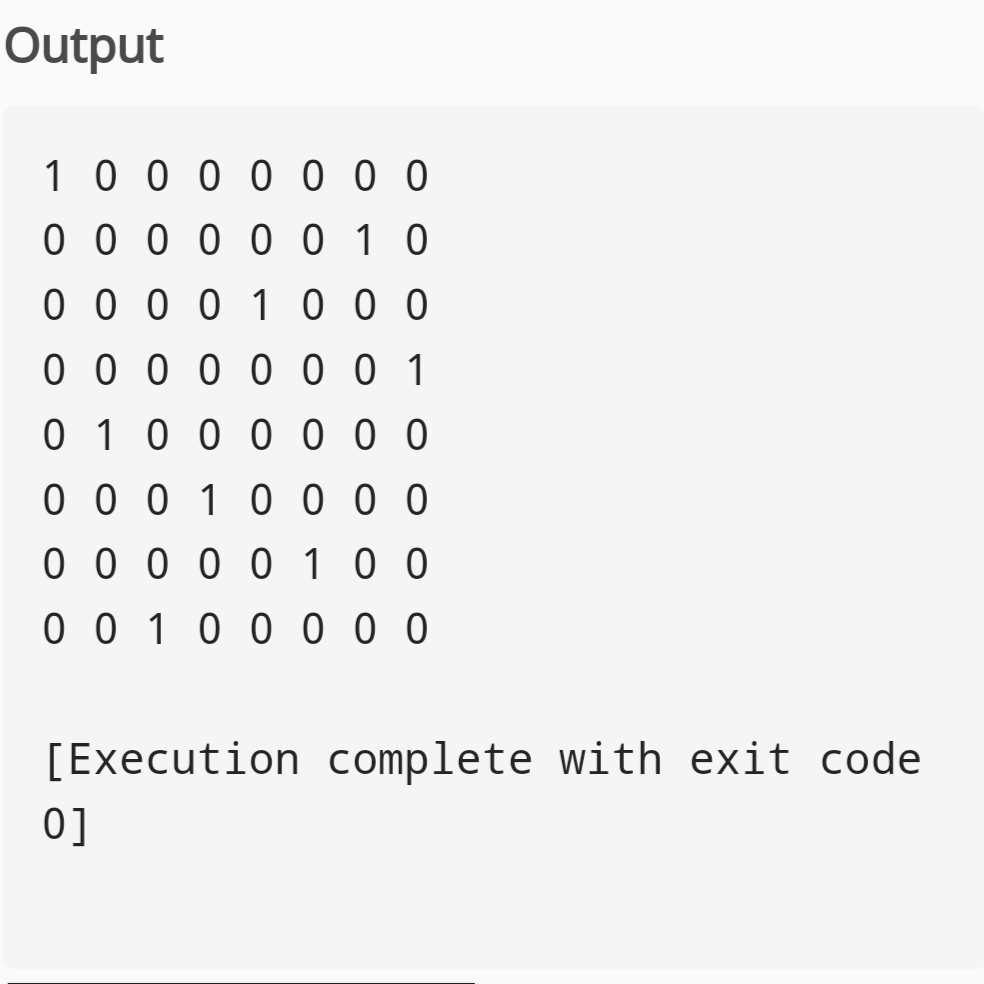
printSolution(board)

return True

# Driver Cde

solveNQueens()

**OUTPUT :**



**BACKTRACKING FOR N-QUEENS :**

global N

N = 4

def printSolution(board):

for i in range(N):

for j in range(N):

print(board[i][j],end = " ")

print()

def isSafe(board, row, col):

for i in range(col):

if board[row][i] == 1:

return False

for i,j in zip(range(row, -1, -1), range(col, -1, -1)):

if board[i][j] == 2:

return False

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):

if col >= N:

return True

for i in range(N):

if isSafe(board,i,col):

board[i][col] = 1

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

return True

board[i][col] = 0

return False

def solveNQ():

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

[0,0,0,0],

[0,0,0,0],

[0,0,0,0]

]

if solveNQUtil(board,1) == False:

print("Solution does not exist")

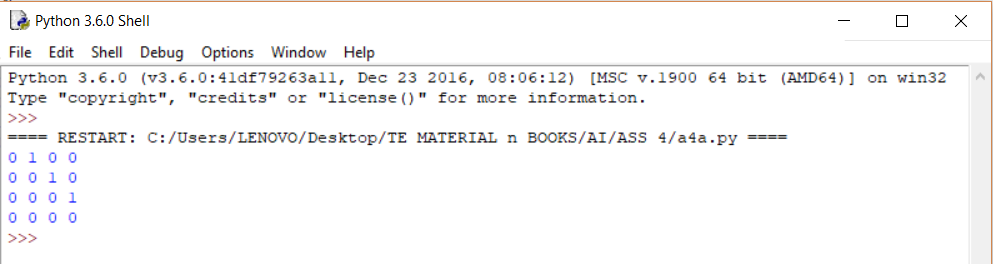
return False

printSolution(board)

return True

solveNQ()

**OUTPUT :**



**GRAPH COLOURING :**

class Graph:

def \_init\_(self, edges, n):

self.adjList = [[] for \_ in range(n)]

for (src, dest) in edges:

self.adjList[src].append(dest)

self.adjList[dest].append(src)

def colorGraph(graph, n):

result = {}

for u in range(n):

assigned = set([result.get(i) for i in graph.adjList[u] if i in result])

color = 1

for c in assigned:

if color != c:

break

color = color + 1

result[u] = color

for v in range(n):

print(f'Color assigned to vertex {v} is {colors[result[v]]}')

if \_name\_ == '\_main\_':

colors = ['', 'BLUE', 'GREEN', 'RED', 'YELLOW', 'ORANGE', 'PINK',

'BLACK', 'BROWN', 'WHITE', 'PURPLE', 'VOILET']

edges = [(0, 1), (0, 4), (0, 5), (4, 5), (1, 4), (1, 3), (2, 3), (2, 4)]

n = 6

graph = Graph(edges, n)

colorGraph(graph, n)

**OUTPUT :**



**GROUP B**

**ASSIGNMENT NO. 5**

**Aim :** Develop an elementry catboat for any suitable customer interaction application.

**Soln :**

**CODE IN PYTHON :**

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('Sbot', '2021') # change it as you need

remind\_name()

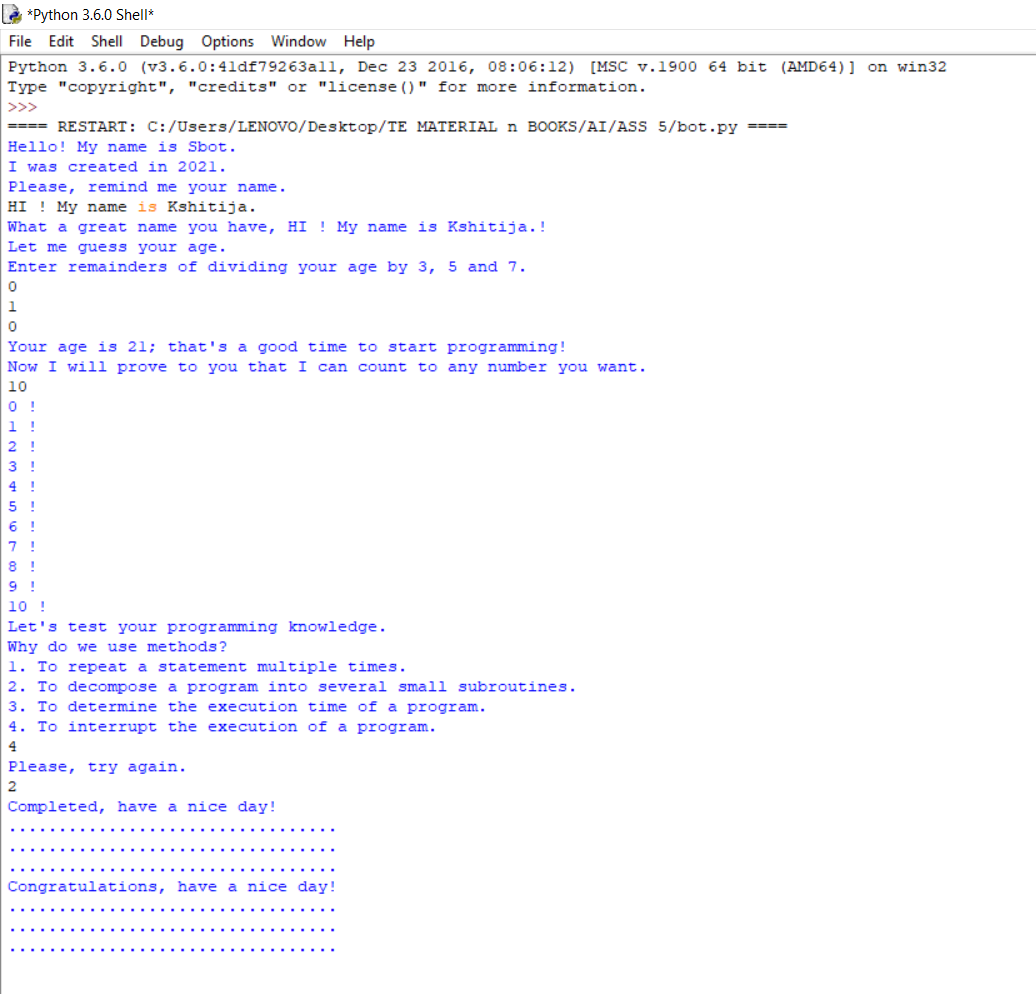
guess\_age()

count()

test()

end()

**OUTPUT :**

****