CHATBOT

```
import random

import re


def greeting():

    responses = ["Hello!", "Hi there!", "Hey!", "Nice to see you!", "Hi! How can I help you?"]

    return random.choice(responses)


def farewell():

    responses = ["Goodbye!", "See you later!", "Have a great day!", "Bye!"]

    return random.choice(responses)

def thanks():

    return "You're welcome! If you have any more questions or need further assistance in the future, feel free to ask. Happy coding!"

def respond(message):

    if any(word in message.lower() for word in ["hello", "hey", "hi"]):

        return greeting()

    elif any(word in message.lower() for word in ["bye", "see you","goodbye"]):

        return farewell()

    elif "how are you?" in message.lower():

        return "I'm just a bot, but I'm doing well. Thanks for asking!"

    elif "your name" in message.lower():

        return "My name is Panda The Chatbot. How can I assist you today?"

    elif any(word in message.lower() for word in ["thank you","thanks"]):

        return thanks()

    elif re.match(r"(\d+)\s*\+\s*(\d+)", message):  # Addition

        numbers = re.match(r"(\d+)\s*\+\s*(\d+)", message)

        result = int(numbers.group(1)) + int(numbers.group(2))

        return f"The result is {result}."

    elif re.match(r"(\d+)\s*-\s*(\d+)", message):  # Subtraction

        numbers = re.match(r"(\d+)\s*-\s*(\d+)", message)
```

```python
        result = int(numbers.group(1)) - int(numbers.group(2))
        return f"The result is {result}."
    elif re.match(r"(\d+)\s*\*\s*(\d+)", message):  # Multiplication
        numbers = re.match(r"(\d+)\s*\*\s*(\d+)", message)
        result = int(numbers.group(1)) * int(numbers.group(2))
        return f"The result is {result}."
    elif re.match(r"(\d+)\s*/\s*(\d+)", message):  # Division
        numbers = re.match(r"(\d+)\s*/\s*(\d+)", message)
        if int(numbers.group(2)) == 0:
            return "Error: Division by zero is undefined."
        result = int(numbers.group(1)) / int(numbers.group(2))
        return f"The result is {result}."

    else:
        return "I'm sorry, I didn't understand that."

def main():
    print("Chatbot: " + greeting())
    while True:
        user_input = input("You: ")
        if any(word in user_input.lower() for word in ["bye", "see you", "goodbye"]):
            print("Chatbot: " + farewell())
            break
        else:
            print("Chatbot: " + respond(user_input))

if __name__ == "__main__":
    main()
'''if __name__ == "__main__": checks if the script is being run directly by the Python interpreter (as opposed to being imported as a module into another script).'''
```

DFS BFS

```
class Graph:'''undirected graph'''

    def __init__(self):

        self.graph = dict()

'''self is instance of class'''

'''Within the constructor, a dictionary named graph is initialized as an instance variable of the class.

 This dictionary will store the adjacency list representation of the graph.

 self.graph--keys=(vertices : [values representing lists of adjacent vertices]).

 ex-key=(0:[1,2])'''

    def add_edge(self, u, v):

        if u not in self.graph:

            self.graph[u] = [v]

        else:

            self.graph[u].append(v)

        if v not in self.graph:

            self.graph[v] = [u]

        else:

            self.graph[v].append(u)


    def DFS(self, v, visited):

        visited.add(v)

        print(v, end=' ')

        for neighbour in self.graph[v]:

            '''This loop iterates through each adjacent vertex (neighbour) of the current vertex v'''

            '''self.graph[v] gives the list of vertices adjacent to v in the graph.'''

            if neighbour not in visited:

'''neighbour has not been visited, the method recursively calls itself (self.DFS(neighbour, visited)), starting the DFS traversal from neighbour.'''

''' call continues until all vertices reachable from the current vertex v have been visited.'''

                self.DFS(neighbour, visited)


    def BFS(self, s):'''s = strting vertex'''
```

```
        visited = set()

        queue = [s]

        visited.add(s)

        while queue:

            vertex = queue.pop(0)

            print(vertex, end=" ")

            for neighbour in self.graph[vertex]:

                if neighbour not in visited:

                    queue.append(neighbour)

                    visited.add(neighbour)



g = Graph()


num_edges = int(input("Enter the number of edges: "))

print("Now enter the edges (u v):")

for _ in range(num_edges):

    u, v = map(int, input().split())

    g.add_edge(u, v)


print("Depth First Traversal (enter the starting vertex):")

start_vertex = int(input())

g.DFS(start_vertex, set())


print("\nBreadth First Traversal (enter the starting vertex):")

start_vertex = int(input())

g.BFS(start_vertex)
```

Prims

```
import heapq
```

```
def prim(graph, start_node):
    mst = set([start_node])
    edges = [
        (cost, start_node, to)
        for to, cost in graph[start_node].items()
        if to != start_node
    ]
    heapq.heapify(edges)

    total_cost = 0

    while edges:
        cost, frm, to = heapq.heappop(edges)
        if to not in mst:
            mst.add(to)
            total_cost += cost
            print(f"Edge: {frm} -> {to}, Cost: {cost}")
            for to_next, cost2 in graph[to].items():
                if to_next not in mst and to != start_node:
                    heapq.heappush(edges, (cost2, to, to_next))

    print(f"\nOverall MST Cost: {total_cost}")

num_nodes = int(input("Enter the number of nodes: "))
graph = {}
for i in range(num_nodes):
    node = input(f"Enter node {i+1} name: ")
    graph[node] = {}
    num_neighbours = int(input(f"Enter the number of neighbours for node {node}: "))
    for j in range(num_neighbours):
        neighbour = input(f"Enter neighbour {j+1} name for node {node}: ")
```

```
        cost = int(input(f"Enter the cost of edge between node {node} and neighbour {neighbour}: "))
        graph[node][neighbour] = cost

start_node = input("Enter the start node: ")

print("\nMinimum Spanning Tree edges:")
prim(graph, start_node)
```

Kruskal

```
class Graph:
    def __init__(self, vertices):
        self.V = vertices
        self.graph = []

    def add_edge(self, u, v, w):
        self.graph.append([u, v, w])

    def find(self, parent, i):
        if parent[i] == i:
            return i
        return self.find(parent, parent[i])

    def apply_union(self, parent, rank, x, y):
        xroot = self.find(parent, x)
        yroot = self.find(parent, y)
        if rank[xroot] < rank[yroot]:
            parent[xroot] = yroot
        elif rank[xroot] > rank[yroot]:
            parent[yroot] = xroot
        else:
            parent[yroot] = xroot
            rank[xroot] += 1
```

```
    def kruskal_algo(self):
        result = []
        i, e = 0, 0
        self.graph = sorted(self.graph, key=lambda item: item[2])
        parent = [i for i in range(self.V)]
        rank = [0] * self.V
        while e < self.V - 1:
            u, v, w = self.graph[i]
            i += 1
            x = self.find(parent, u)
            y = self.find(parent, v)
            if x != y:
                e += 1
                result.append([u, v, w])
                self.apply_union(parent, rank, x, y)
        for u, v, weight in result:
            print(f"{u} - {v}: {weight}")

def main():
    num_vertices = int(input("Enter the number of vertices: "))
    g = Graph(num_vertices)
    num_edges = int(input("Enter the number of edges: "))
    for _ in range(num_edges):
        u, v, w = map(int, input("Enter edge (u v w): ").split())
        g.add_edge(u, v, w)
    g.kruskal_algo()

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

Dijkstra

```
import heapq


class Graph:
    def __init__(self, V):
        self.V = V
        self.adj = [[] for _ in range(V)]

    def addEdge(self, u, v, w):
        self.adj[u].append((v, w))



    def shortestPath(self, src):
        pq = []
        heapq.heappush(pq, (0, src))
        dist = [float('inf')] * self.V
        dist[src] = 0

        while pq:
            d, u = heapq.heappop(pq)
            for v, weight in self.adj[u]:
                if dist[v] > dist[u] + weight:
                    dist[v] = dist[u] + weight
                    heapq.heappush(pq, (dist[v], v))

        for i in range(self.V):
            print(f"{i} \t\t {dist[i]}")


if __name__ == "__main__":
    V = int(input("Enter the number of vertices: "))
    g = Graph(V)
```

```
    E = int(input("Enter the number of edges: "))
    for _ in range(E):
        u, v, w = map(int, input("Enter the edge (u, v) and its weight w: ").split())
        g.addEdge(u, v, w)

    src = int(input("Enter the source vertex: "))
    g.shortestPath(src)
```

N queen

```
#include<iostream>
using namespace std;
int grid[10][10];

void print(int n)
{
        for (int i = 0;i <= n-1; i++)
        {
        for (int j = 0;j <= n-1; j++)
                {
                        cout <<grid[i][j]<< " ";
                }
        cout<<endl;
        }
        cout<<endl;
        cout<<endl;
}
bool isSafe(int col, int row, int n)
{
        for (int i = 0; i < row; i++)
        {
                if (grid[i][col])
```

```
            {
                return false;
            }
        }

        for (int i = row,j = col;i >= 0 && j >= 0; i--,j--)
        {
            if (grid[i][j])
            {
                return false;
            }
        }
        for (int i = row, j = col; i >= 0 && j < n; j++, i--)
        {
            if (grid[i][j])
            {
                return false;
            }
        }
        return true;
}
bool solve (int n, int row)
{
            if (n == row)
            {
                print(n);
                return true;
            }

            bool res = false;
            for (int i = 0;i <=n-1;i++)
```

```
        {
            if (isSafe(i, row, n))
            {
                grid[row][i] = 1;
                res = solve(n, row+1) || res;
                grid[row][i] = 0;
            }
        }
    return res;
}
int main()
{
    int n;
    char ch;

    do
    {

    cout<<"Enter the number of queen"<<endl;
    cin >> n;
    for (int i = 0;i < n;i++)
    {
        for (int j = 0;j < n;j++)
        {
            grid[i][j] = 0;
        }
    }
    bool res = solve(n, 0);
    if(res == false)
    {
        cout << "Not possible" << endl;
```

```
        }
        else
        {
                cout << endl;
        }
        cout<<"Do you want to Continue:";
        cin>>ch;
}while(ch=='Y'||ch=='y');
        return 0;
}
```

A*

```
import heapq

class PuzzleNode:
   def __init__(self, state, parent=None, move=0, depth=0):
      self.state = state
      self.parent = parent
      self.move = move
      self.depth = depth

   def __lt__(self, other):
      return (self.depth + self.heuristic()) < (other.depth + other.heuristic())

   def __eq__(self, other):
      return self.state == other.state

   def __hash__(self):
      return hash(str(self.state))
```

```
    def __str__(self):
        return str(self.state)

    def heuristic(self):
        count = 0
        goal = [[1, 2, 3], [4, 5, 6], [7, 8, 0]]
        for i in range(3):
            for j in range(3):
                if self.state[i][j] != goal[i][j]:
                    count += 1
        return count

    def get_neighbors(self):
        neighbors = []
        i, j = self.find_blank()
        for x, y in [(i-1, j), (i+1, j), (i, j-1), (i, j+1)]:
            if 0 <= x < 3 and 0 <= y < 3:
                neighbor_state = [row[:] for row in self.state]
                neighbor_state[i][j], neighbor_state[x][y] = neighbor_state[x][y], neighbor_state[i][j]
                neighbors.append(PuzzleNode(neighbor_state, parent=self, move=neighbor_state[x][y],
depth=self.depth+1))
        return neighbors

    def find_blank(self):
        for i in range(3):
            for j in range(3):
                if self.state[i][j] == 0:
                    return i, j

def reconstruct_path(node):
    path = []
```

```
    while node:
        path.append(node)
        node = node.parent
    return path[::-1]

def astar(start_state):
    start_node = PuzzleNode(start_state)
    frontier = [start_node]
    explored = set()

    while frontier:
        node = heapq.heappop(frontier)
        if node.state == [[1, 2, 3], [4, 5, 6], [7, 8, 0]]:
            return reconstruct_path(node)
        explored.add(node)
        for neighbor in node.get_neighbors():
            if neighbor not in explored and neighbor not in frontier:
                heapq.heappush(frontier, neighbor)
            elif neighbor in frontier:
                existing_neighbor = frontier[frontier.index(neighbor)]
                if neighbor < existing_neighbor:
                    frontier.remove(existing_neighbor)
                    heapq.heappush(frontier, neighbor)
    return None

def print_solution(path):
    for i, node in enumerate(path):
        print(f"Step {i}:")
        for row in node.state:
            print(row)
        print()
```

```
def get_user_input():
    print("Enter the start state of the puzzle (use 0 for the blank tile):")
    start_state = []
    for i in range(3):
        row = input(f"Enter row {i+1} separated by spaces: ").strip().split()
        start_state.append([int(x) for x in row])
    return start_state


if __name__ == "__main__":
    start_state = get_user_input()
    path = astar(start_state)
    if path:
        print("Solution found!")
        print_solution(path)
    else:
        print("No solution found.")
```