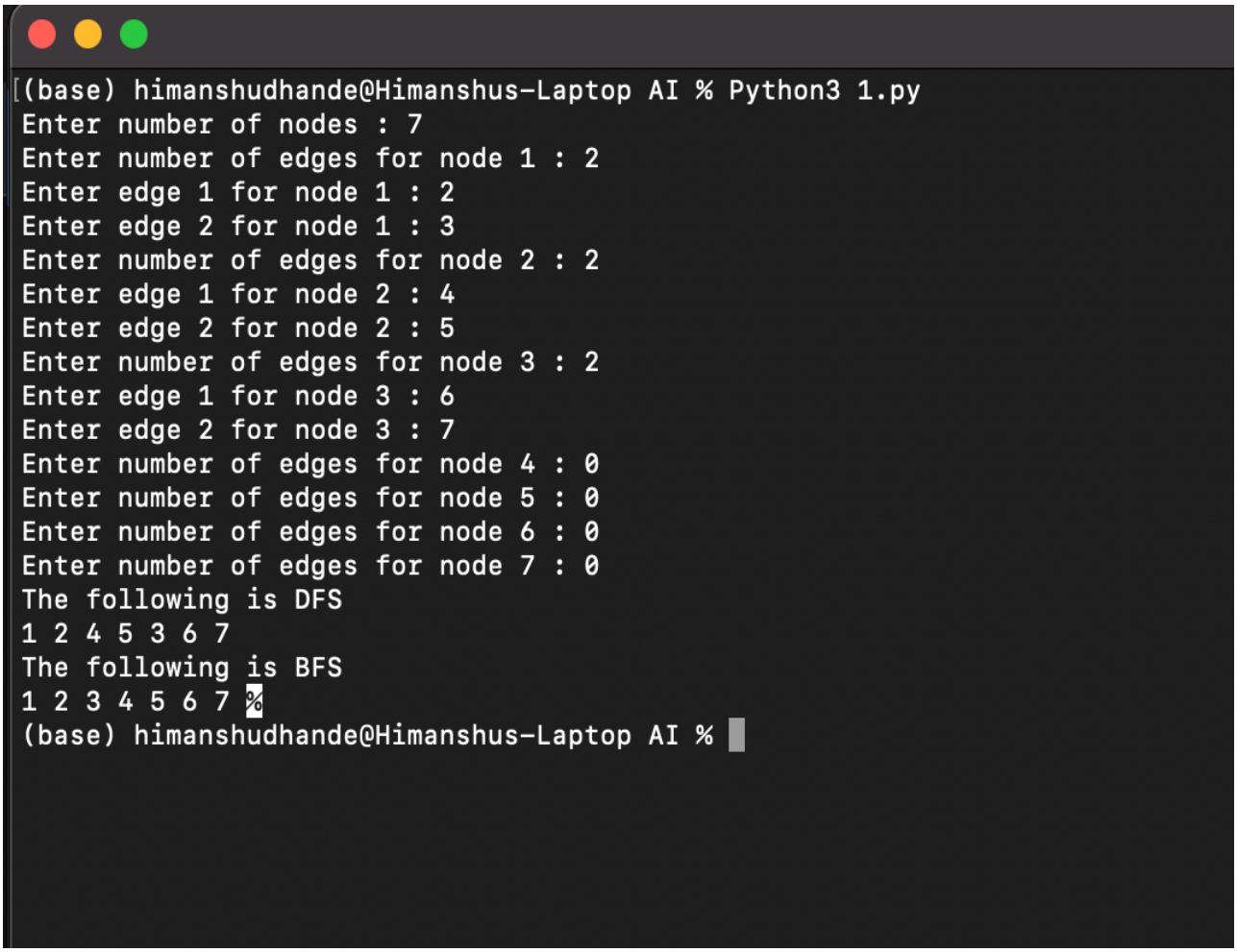
```
'''Assignment 1
Implement DFS and BFS for a given graph.'''

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

def bfs(visited,graph,node,queue):
    visited.add(node)
    queue.append(node)

    while queue:
        s = queue.pop(0)
        print(s,end = " ")
        for neighbour in graph[s]:
            if neighbour not in visited:
                visited.add(neighbour)
                queue.append(neighbour)

def main():
    visited1 = set() # TO keep track of DFS visited nodes
    visited2 = set() # TO keep track of BFS visited nodes
    queue = []       # For BFS
    n = int(input("Enter number of nodes : "))
    graph = dict()

    for i in range(1,n+1):
```

```
        edges = int(input("Enter number of edges for node {} : ".format(i)))
        graph[i] = list()
        for j in range(1,edges+1):
            node = int(input("Enter edge {} for node {} : ".format(j,i)))
            graph[i].append(node)

    print("The following is DFS")
    dfs(visited1, graph, 1)
    print()
    print("The following is BFS")
    bfs(visited2, graph, 1, queue)

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

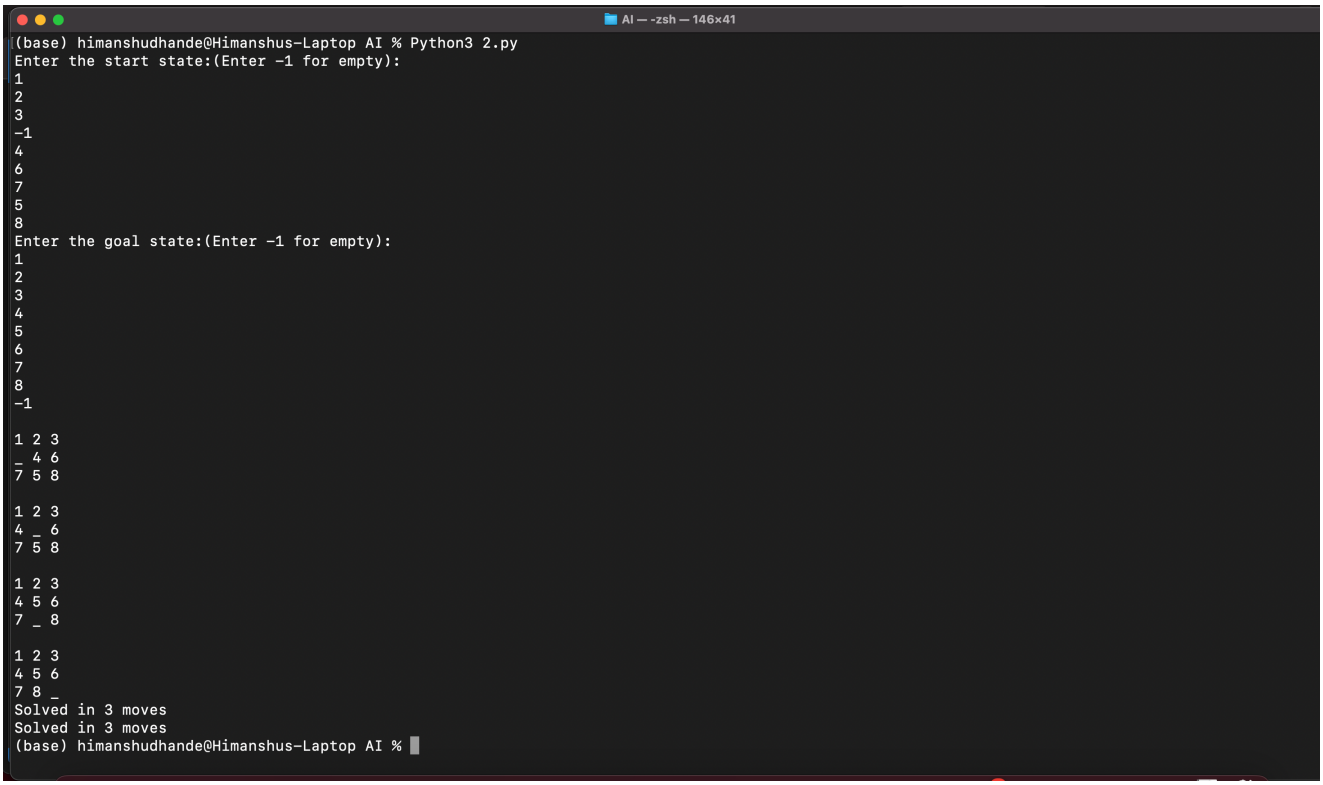

(base) himanshudhande@Himanshus-Laptop AI % Python3 1.py
Enter number of nodes : 7
Enter number of edges for node 1 : 2
Enter edge 1 for node 1 : 2
Enter edge 2 for node 1 : 3
Enter number of edges for node 2 : 2
Enter edge 1 for node 2 : 4
Enter edge 2 for node 2 : 5
Enter number of edges for node 3 : 2
Enter edge 1 for node 3 : 6
Enter edge 2 for node 3 : 7
Enter number of edges for node 4 : 0
Enter number of edges for node 5 : 0
Enter number of edges for node 6 : 0
Enter number of edges for node 7 : 0
The following is DFS
1 2 4 5 3 6 7
The following is BFS
1 2 3 4 5 6 7 %
(base) himanshudhande@Himanshus-Laptop AI %

```
'''Assignment 2
Implement A* algorithm for 8 puzzle problem.'''
g=0
def print_board(elements):
    for i in range(9):
        if i%3 == 0:
            print()
        if elements[i]==-1:
            print("_", end = " ")
        else:
            print(elements[i], end = " ")
    print()

def solvable(start):
    inv=0

    for i in range(9):
        if start[i] <= 1:
            continue
        for j in range(i+1,9):
            if start[j]==-1:
                continue
            if start[i]>start[j]:
                inv+=1
    if inv%2==0:
        return True
    return False

def heuristic(start,goal):
    global g
```

```
    h = 0
    for i in range(9):
        for j in range(9):
            if start[i] == goal[j] and start[i] != -1:
                h += (abs(j-i))//3 + (abs(j-i))%3
    return h + g

def moveleft(start,position):
    start[position],start[position-1]= start[position-1],start[position]

def moveright(start,position):
    start[position],start[position+1]= start[position+1],start[position]

def moveup(start,position):
    start[position],start[position-3]= start[position-3],start[position]

def movedown(start,position):
    start[position],start[position+3]= start[position+3],start[position]

def movetile(start,goal):
    emptyat= start.index(-1)
    row = emptyat//3
    col = emptyat%3
    t1,t2,t3,t4 = start[:],start[:],start[:],start[:]
    f1,f2,f3,f4 = 100,100,100,100

    if col -1 >=0:
        moveleft(t1, emptyat)
        f1 = heuristic(t1, goal)
    if col+1<3:
```

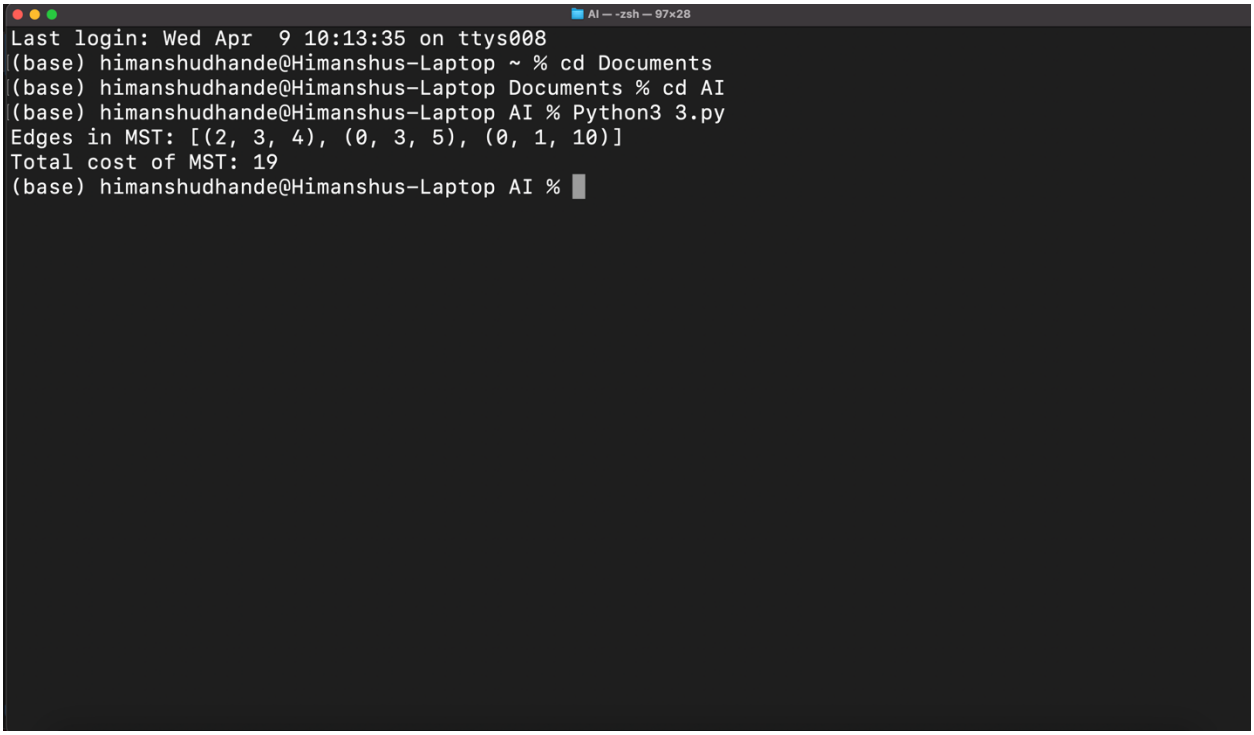
```
        moveright(t2, emptyat)
        f2 = heuristic(t2, goal)
    if row + 1 <3:
        movedown(t3, emptyat)
        f3 = heuristic(t3, goal)
    if row-1>=0:
        moveup(t4, emptyat)
        f4 = heuristic(t4, goal)

    min_heuristic = min(f1, f2,f3,f4)

    if f1==min_heuristic:
        moveleft(start, emptyat)
    elif f2==min_heuristic:
        moveright(start, emptyat)
    elif f3==min_heuristic:
        movedown(start, emptyat)
    elif f4 == min_heuristic:
        moveup(start, emptyat)



def solveEight(start,goal):
    global g
    g+=1
    movetile(start,goal)
    print_board(start)
    f = heuristic(start,goal)
    if f == g:
        print("Solved in {} moves".format(f))
        return
```

```
    solveEight(start,goal)

def main():
    global g
    start = list()
    goal = list()
    print("Enter the start state:(Enter -1 for empty):")
    for i in range(9):
        start.append(int(input()))

    print("Enter the goal state:(Enter -1 for empty):")
    for i in range(9):
        goal.append(int(input()))

    print_board(start)

    # To check if solvable
    if solvable(start):
        solveEight(start,goal)
        print("Solved in {} moves".format(g))
    else:
        print("Not possible to solve")

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


AI — -zsh — 146×41
(base) himanshudhande@Himanshus-Laptop AI % Python3 2.py
Enter the start state:(Enter -1 for empty):
1
2
3
-1
4
6
7
5
8
Enter the goal state:(Enter -1 for empty):
1
2
3
4
5
6
7
8
-1
1 2 3
_ 4 6
7 5 8
1 2 3
4 _ 6
7 5 8
1 2 3
4 5 6
7 _ 8
1 2 3
4 5 6
7 8 _
Solved in 3 moves
Solved in 3 moves
(base) himanshudhande@Himanshus-Laptop AI %

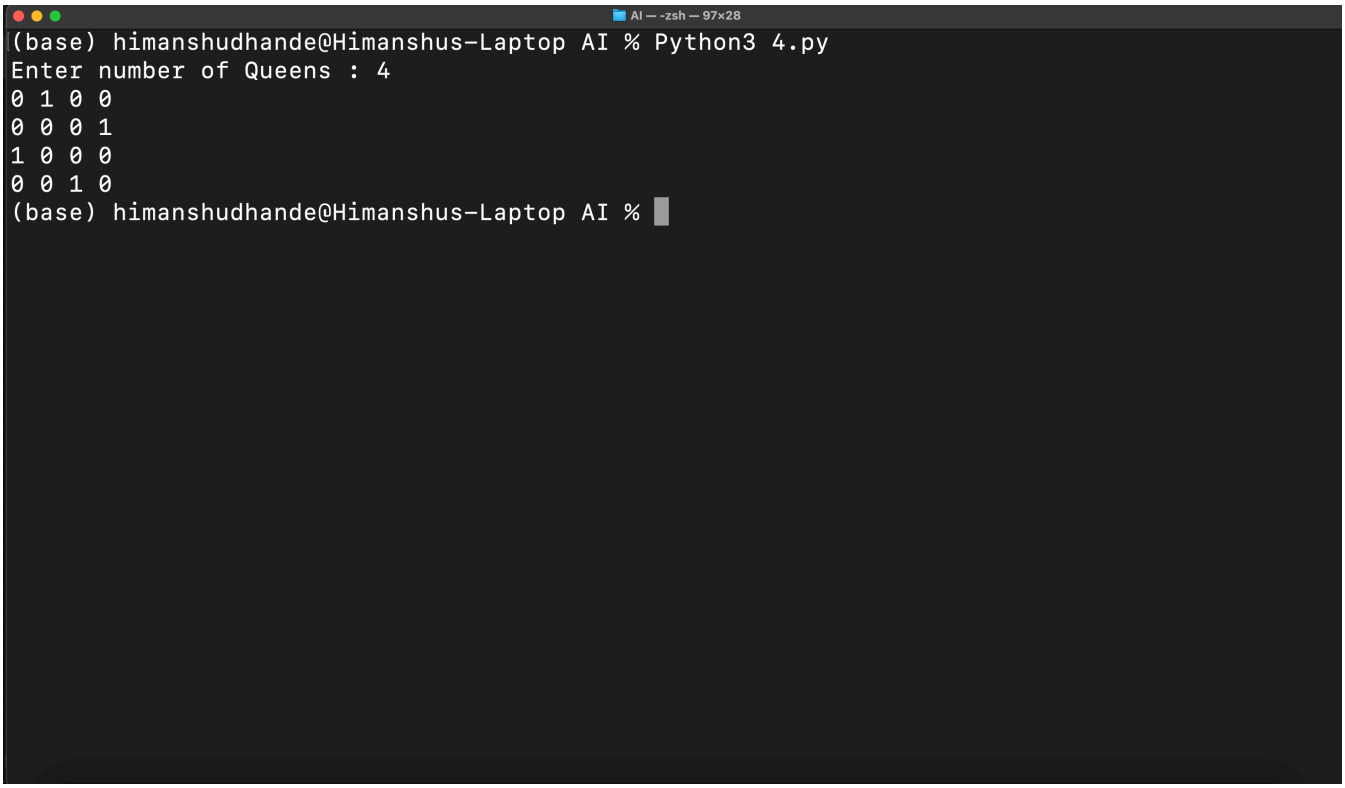
```
'''Assignment 3: Kruskal's Algorithm'''
# Disjoint Set Union (Union-Find)
class DisjointSet:
    def __init__(self, n):
        self.parent = list(range(n))
        self.rank = [0] * n

    def find(self, u):
        if self.parent[u] != u:
            self.parent[u] = self.find(self.parent[u])  # Path compression
        return self.parent[u]

    def union(self, u, v):
        u_root = self.find(u)
        v_root = self.find(v)

        if u_root == v_root:
            return False  # Already connected

        # Union by rank
        if self.rank[u_root] < self.rank[v_root]:
            self.parent[u_root] = v_root
        elif self.rank[u_root] > self.rank[v_root]:
            self.parent[v_root] = u_root
        else:
            self.parent[v_root] = u_root
            self.rank[u_root] += 1

        return True


# Kruskal's Algorithm
def kruskal(n, edges):
    ds = DisjointSet(n)
    mst = []
    total_cost = 0

    # Sort edges by weight
    edges.sort(key=lambda x: x[2])  # x = (u, v, weight)

    for u, v, weight in edges:
        if ds.union(u, v):
            mst.append((u, v, weight))
            total_cost += weight

    return mst, total_cost
```

```
# Example graph
edges = [
    (0, 1, 10),
    (0, 2, 6),
    (0, 3, 5),
    (1, 3, 15),
    (2, 3, 4)
]

num_nodes = 4  # Nodes labeled 0 to 3

mst, cost = kruskal(num_nodes, edges)
print("Edges in MST:", mst)
print("Total cost of MST:", cost)
```



```
'''Assignment 4 nQueen Problem'''
def issafe(arr,x,y,n):
    for row in range(x):
        if arr[row][y] ==1:
            # Checking column attack
            return False
    row = x
    col = y
    #Checking Diagonal Attack
    while row>=0 and col>=0:
        if arr[row][col]==1:
            return False
        row-=1
        col-=1

    row = x
    col = y
    #Checking Anti Diagonal Attack
    while row>=0 and col<n:
        if arr[row][col]==1:
            return False
        row-=1
        col+=1

    return True

def nQueen(arr,x,n):
    if x>=n:
        return True

    for col in range(n):
        if issafe(arr,x,col,n):
            arr[x][col]=1
            if nQueen(arr,x+1,n):
                return True
            arr[x][col] = 0

    return False

def main():
    n = int(input("Enter number of Queens : "))
    arr = [[0]*n for i in range(n)]

    if nQueen(arr,0,n):
        for i in range(n):
            for j in range(n):
```

```
            print(arr[i][j],end=" ")
        print()

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



```
'''Assignment 5: Grocery Store Chatbot
This chatbot can assist users with grocery-related queries, such as checking prices, order status, and store information.'''
import re

def chatbot_response(user_input):
    user_input = user_input.lower()

    responses = {
        r"\b(1|hello|hi|hey)\b": " Hello! Welcome to our grocery store. How can I help you today?",
        r"\b(2|how are you)\b": "I'm just a bot, but I'm here to assist you with groceries!",
        r"\b(3|order status|track order)\b": "Please provide your order ID to check the status.",
        r"\b(4|shipping time|delivery time)\b": "We offer same-day delivery and standard shipping (3-5 business days).",
        r"\b(5|return policy)\b": "You can return items within 7 days if unopened. Would you like help with a return?",
        r"\b(6|thank you|thanks)\b": " You're welcome! Let me know if you need anything else.",
        r"\b(7|price|cost)\b": "Please specify the product name to check its price.",
        r"\b(8|milk)\b": "Milk is 30rs per liter.",
        r"\b(9|eggs)\b": "A dozen eggs cost 80rs.",
        r"\b(10|rice)\b": "Rice is 50rs per kg.",
        r"\b(11|vegetables|veggies)\b": "We have fresh vegetables available. What are you looking for?",
        r"\b(12|fruits)\b": "We have apples, bananas, and oranges in stock. Which one do you need?",
        r"\b(13|snacks)\b": "We have chips, biscuits, and chocolates available.",
        r"\b(14|beverages|drinks)\b": "We have soft drinks, juices, and bottled water. What would you like?",
        r"\b(15|buy|order)\b": "You can place an order on our website or visit our store.",
        r"\b(16|payment methods)\b": "We accept cash, credit/debit cards, and UPI payments.",
        r"\b(17|store hours|timing)\b": "Our store is open from 8 AM to 10 PM every day.",
        r"\b(18|location|address)\b": "We are located at XYZ Market, Main Street, City.",
        r"\b(19|bye|exit)\b": "Goodbye! Happy shopping! 🛍️"
    }

    for pattern, response in responses.items():
        if re.search(pattern, user_input):
            return response

    return "I am sorry, I didn't understand that. Can you rephrase or ask about a specific grocery item?"

# Chatbot interaction loop
print("Welcome to our Grocery Chatbot! Type 'exit' to end the conversation.")

while True:
```

```
user_message = input("You: ")
if user_message.lower() in ["bye", "exit"]:
    print("Chatbot: Goodbye! Happy shopping! 🛍️")
    break
response = chatbot_response(user_message)
print("Chatbot:", response)
```