Practical No. 1

```
#BFS

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

print("following path is Breadth-First Algorithm")
bfs(visited,graph,'A')
```

```
#DFS
graph = {
'A':['B','C'],
'B':['D','E'],
'C':['F'],
'D':[],
'E':['F'],
'F':[]
}
visited = set()

def dfs(visited,graph,node):
    if node  not in  visited:
        print(node,end=" \n")
        visited.add(node)

        for neighbour in graph[node]:
            dfs (visited,graph,neighbour)
```

```
print("\nfollowing path is Depth-First Algorithm")

dfs(visited,graph,'A')
```

OUTPUT:

lab314@lab314-ThinkCentre-M70s:~$ python3 ass1.py
following path is Breadth-First Algorithm
A B C D E F

following path is Depth-First Algorithm
A
B
D
E
F
C

PRACTICAL2

```python
import math
import heapq

class Graph:
    def __init__(self, vertices):
        self.V = vertices
        self.adj_matrix = [[-1 for _ in range(vertices)] for _ in range(vertices)]
        self.coordinates = [(0, 0)] * vertices  # Placeholder for coordinates

    def add_edge(self, u, v, w):
        self.adj_matrix[u][v] = w
        self.adj_matrix[v][u] = w  # Undirected graph

    def set_coordinates(self, node, x, y):
        self.coordinates[node] = (x, y)

    def heuristic(self, a, b):
        x1, y1 = self.coordinates[a]
        x2, y2 = self.coordinates[b]
        return math.hypot(x2 - x1, y2 - y1)

    def a_star(self, start, goal):
        g = [float('inf')] * self.V
        f = [float('inf')] * self.V
        visited = [False] * self.V
        parent = [-1] * self.V

        g[start] = 0
        f[start] = self.heuristic(start, goal)
```

```
pq = [(f[start], start)]  # (f_score, node)

while pq:
    _, current = heapq.heappop(pq)

    if visited[current]:
        continue
    visited[current] = True

    if current == goal:
        break

    for neighbor in range(self.V):
        weight = self.adj_matrix[current][neighbor]
        if weight != -1 and not visited[neighbor]:
            tentative_g = g[current] + weight
            if tentative_g < g[neighbor]:
                g[neighbor] = tentative_g
                f[neighbor] = g[neighbor] + self.heuristic(neighbor, goal)
                parent[neighbor] = current
                heapq.heappush(pq, (f[neighbor], neighbor))

if visited[goal]:
    print(f"Shortest path cost from {start} to {goal}: {g[goal]}")
    path = []
    node = goal
    while node != -1:
        path.append(node)
        node = parent[node]
    print("Path:", ' -> '.join(map(str, reversed(path))))
else:
```

```
        print(f"No path found from {start} to {goal}")

# Example usage
g = Graph(5)
g.set_coordinates(0, 0, 0)
g.set_coordinates(1, 1, 2)
g.set_coordinates(2, 2, 1)
g.set_coordinates(3, 3, 3)
g.set_coordinates(4, 4, 0)

g.add_edge(0, 1, 5)
g.add_edge(0, 2, 10)
g.add_edge(1, 2, 3)
g.add_edge(2, 3, 7)
g.add_edge(1, 3, 2)
g.add_edge(3, 4, 1)

g.a_star(0, 3)
```

OUTPUT

```
Shortest path cost from 0 to 3: 7
Path: 0 -> 1 -> 3
```

PRACTICAL3

```
def SelectionSort(arr):
    for i in range(len(arr)):
        min_index = i
        for j in range(i + 1, len(arr)):
            if arr[j] < arr[min_index]:
                min_index = j
        arr[i], arr[min_index] = arr[min_index], arr[i]
    return arr

# Take user input
user_input = input("Enter numbers separated by spaces: ")
arr = list(map(int, user_input.split()))

print("Selection Sort is:")
print(SelectionSort(arr))
```

Practical No. 4

CODE:

```
def is_safe(board, row, col):
    n = len(board)

    # Check the row
    for i in range(col):
        if board[row][i] == 1:
            return False

    # Check the upper diagonal
    r, c = row, col
    while r >= 0 and c >= 0:
        if board[r][c] == 1:
            return False
        r -= 1
        c -= 1

    # Check the lower diagonal
    r, c = row, col
    while r < n and c >= 0:
        if board[r][c] == 1:
            return False
        r += 1
        c -= 1

    # If no conflicts found, it's safe to place a queen
    return True

def backtrack(board, col, solutions):
    n = len(board)

    # If all queens are placed, a valid solution is found
    if col == n:
        solutions.append([row[:] for row in board])
        return

    # Explore all possible positions in the current column
    for row in range(n):
        if is_safe(board, row, col):
            board[row][col] = 1  # Place a queen

            # Recursively move to the next column
            backtrack(board, col + 1, solutions)

            board[row][col] = 0  # Remove the queen (backtrack)

def solve_nqueens(n):
    board = [[0] * n for _ in range(n)]
```

```python
    solutions = []
    backtrack(board, 0, solutions)
    return solutions

# Example usage
n = 4
solutions = solve_nqueens(n)

print(f"Total solutions for {n}-queens problem: {len(solutions)}")
for i, solution in enumerate(solutions):
    print(f"Solution {i+1}:")
    for row in solution:
        print(row)
    print()
```

<u>OUTPUT:</u>

```
Total solutions for 4-queens problem: 2
Solution 1:
[0, 0, 1, 0]
[1, 0, 0, 0]
[0, 0, 0, 1]
[0, 1, 0, 0]

Solution 2:
[0, 1, 0, 0]
[0, 0, 0, 1]
[1, 0, 0, 0]
[0, 0, 1, 0]


=== Code Execution Successful ===
```

Practical No. 5

CODE:

```
import datetime
import random  # Required for random.choice


greetings = ['Hello!', 'Hi!']
salutations = ['Bye!', 'See you soon!', 'Have a good day!']
others = {
    "weather": ["sunny", "chilly", "rainy"],
    "name": ["My name is HarshBot", "Myself HarshBot", "People call me HarshBot"]
}


def greet():
    print(random.choice(greetings))


def farewell():
    print(random.choice(salutations))


def date():
    print("The date is", str(datetime.datetime.now())[:10])


def time():
    print("The time is", str(datetime.datetime.now())[11:16])


def process(inp):
    if "hello" in inp.lower():
        greet()
    elif "bye" in inp.lower():
        farewell()
        return True
    elif "date" in inp.lower():
        date()
```

```
    elif "time" in inp.lower():
        time()
    else:
        found_response = False
        for key, value in others.items():
            if key in inp.lower():
                print(random.choice(value))
                found_response = True
                break
        if not found_response:
            print("idk")

    return False

finished = False
while not finished:
    inp = input("> ")
    finished = process(inp)
```

OUTPUT:

```
> hello
Hi!
> what's the date
The date is 2025-04-08
> tell me the weather
sunny
> bye
Bye!

=== Code Execution Successful ===
```

Practical No. 6

CODE:

```
rules = {
    "rule1": {
        "condition" : lambda data : data["attendance"] >=7.5 and data["creative"] >= 7,
        "output" : "Excellent Employee!"
    },
    "rule2" : {
        "condition" : lambda  data : data["attendance"] >= 5 and data["creative"] >=6 ,
        "output" : "Average Employee"
    },
    "rule3" : {
        "condition" : lambda data: data["attendance"] <5 and data["creative"] <5,
        "output" : "Poor"
    }
}
def evaluate(data):
    for rule in rules.values():
        if rule['condition'](data):
            return rule['output']
    return "Need more details"

data = {
    "attendance" : int(input("attendance: ")),
    "creative" : int(input("creative: "))
    }

perf = evaluate(data)
print(perf)
```

OUTPUT:

```
attendance: 9
creative: 10
Excellent Employee!
```