Practical - 1

1.BFS

Code:

```
class Graph:
    def __init__(self):
        self.graph = {}

    def add_edge(self, u, v):
        if u not in self.graph:
            self.graph[u] = []
        if v not in self.graph:
            self.graph[v] = []
        self.graph[u].append(v)
        self.graph[v].append(u)

    def bfs(self, start):
        visited = set()
        queue = [start]
        visited.add(start)

        while queue:
            v = queue.pop(0)
            print(v, end=' ')

            for neighbor in self.graph[v]:
                if neighbor not in visited:
                    queue.append(neighbor)
                    visited.add(neighbor)


# Test the BFS algorithm
g = Graph()
g.add_edge(0, 1)
g.add_edge(0, 2)
g.add_edge(1, 3)
```

```
g.add_edge(2, 3)
g.add_edge(2, 4)
g.add_edge(3, 5)
g.add_edge(4, 5)

print("\nBreadth First Traversal (BFS):")
g.bfs(3)
```

Output:

```
Breadth First Traversal (BFS):
3 1 2 5 0 4
```

2.DFS
Code:

```
class Graph:
    def __init__(self):
        self.graph = {}

    def add_edge(self, u, v):
        if u not in self.graph:
            self.graph[u] = []
        if v not in self.graph:
            self.graph[v] = []
        self.graph[u].append(v)
        self.graph[v].append(u)

    def dfs_util(self, v, visited):
        visited.add(v)
        print(v, end=' ')

        for neighbor in self.graph[v]:
            if neighbor not in visited:
                self.dfs_util(neighbor, visited)
```

```
    def dfs(self, start):
        visited = set()
        self.dfs_util(start, visited)


# Test the DFS algorithm
g = Graph()
g.add_edge(0, 1)
g.add_edge(0, 2)
g.add_edge(1, 3)
g.add_edge(2, 3)
g.add_edge(2, 4)
g.add_edge(3, 5)
g.add_edge(4, 5)


print("Depth First Traversal (DFS):")
g.dfs(3)
```

Output:

```
Depth First Traversal (DFS):
3 1 0 2 4 5
```

Practical -2

Code:

```
import heapq

# Define the puzzle state class
class PuzzleState:
    def __init__(self, state, parent=None, move=None):
        self.state = state
        self.parent = parent
        self.move = move
        self.cost = 0

        if self.parent:
            self.cost = self.parent.cost + 1

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

    # Define the less than operator
    def __lt__(self, other):
        return self.cost < other.cost

    # Hash the state
    def __hash__(self):
        return hash(str(self.state))

    # Get the position of the blank space
    def get_blank_position(self):
        for i in range(3):
            for j in range(3):
                if self.state[i][j] == 0:
                    return (i, j)
```

```
# Get possible moves from a state
def get_children(state):
    children = []
    blank_position = state.get_blank_position()
    moves = [(0, 1), (1, 0), (0, -1), (-1, 0)]

    for move in moves:
        new_x = blank_position[0] + move[0]
        new_y = blank_position[1] + move[1]

        if 0 <= new_x < 3 and 0 <= new_y < 3:
            new_state = [row[:] for row in state.state]
            new_state[blank_position[0]][blank_position[1]] =
new_state[new_x][new_y]
            new_state[new_x][new_y] = 0
            children.append(PuzzleState(new_state, state, move))

    return children

# Define the heuristic function
def heuristic(state, target):
    count = 0
    for i in range(3):
        for j in range(3):
            if state[i][j] != target[i][j]:
                count += 1
    return count

# Implement the A* algorithm
def a_star(start_state, target_state):
    open_list = []
    closed_list = set()

    heapq.heappush(open_list, start_state)

    while open_list:
```

```
        current_state = heapq.heappop(open_list)

        if current_state.state == target_state:
            path = []
            while current_state:
                path.append((current_state.state, current_state.move))
                current_state = current_state.parent
            return path[::-1]

        closed_list.add(current_state)

        for child in get_children(current_state):
            if child in closed_list:
                continue

            child.cost += heuristic(child.state, target_state)

            if child not in open_list:
                heapq.heappush(open_list, child)
            elif child in open_list and child.cost <
open_list[open_list.index(child)].cost:
                open_list.remove(child)
                heapq.heappush(open_list, child)

    return None

# Example usage:
start_state = [[1, 2, 3], [4, 0, 5], [6, 7, 8]]
target_state = [[0, 1, 2], [3, 4, 5], [6, 7, 8]]

start_node = PuzzleState(start_state)
target_node = PuzzleState(target_state)

path = a_star(start_node, target_state)

if path:
    for i, (state, move) in enumerate(path):
```

```
        print(f"Step {i + 1}: Move {move} =>")
        for row in state:
            print(row)
        print()
else:
    print("No solution found.")
```

Output:

```
Step 1: Move None =>
[1, 2, 3]
[4, 0, 5]
[6, 7, 8]

Step 2: Move (0, -1) =>
[1, 2, 3]
[0, 4, 5]
[6, 7, 8]

Step 3: Move (-1, 0) =>
[0, 2, 3]
[1, 4, 5]
[6, 7, 8]

Step 4: Move (0, 1) =>
[2, 0, 3]
[1, 4, 5]
[6, 7, 8]
```

```
Step 5: Move (0, 1) =>
[2, 3, 0]
[1, 4, 5]
[6, 7, 8]

Step 6: Move (1, 0) =>
[2, 3, 5]
[1, 4, 0]
[6, 7, 8]

Step 7: Move (0, -1) =>
[2, 3, 5]
[1, 0, 4]
[6, 7, 8]

Step 8: Move (-1, 0) =>
[2, 0, 5]
[1, 3, 4]
[6, 7, 8]

Step 9: Move (0, -1) =>
[0, 2, 5]
[1, 3, 4]
[6, 7, 8]
```

Practical-3

Code:

```
def selection_sort(arr):
    n = len(arr)

    for i in range(n):
        min_index = i
        for j in range(i+1, n):
            # Find the index of the minimum element in the unsorted part of
the array
            if arr[j] < arr[min_index]:
                min_index = j

        # Swap the minimum element with the first element of the unsorted
part
        arr[i], arr[min_index] = arr[min_index], arr[i]

        # Print the array after each iteration
        print("Iteration", i+1, ":", arr)

    return arr

# Example usage:
arr = [64, 25, 12, 22, 11]
sorted_arr = selection_sort(arr)
print("Sorted array is:", sorted_arr)
```

Output:

```
Iteration 1 : [11, 25, 12, 22, 64]
Iteration 2 : [11, 12, 25, 22, 64]
Iteration 3 : [11, 12, 22, 25, 64]
Iteration 4 : [11, 12, 22, 25, 64]
Iteration 5 : [11, 12, 22, 25, 64]
Sorted array is: [11, 12, 22, 25, 64]
```

Practical-4

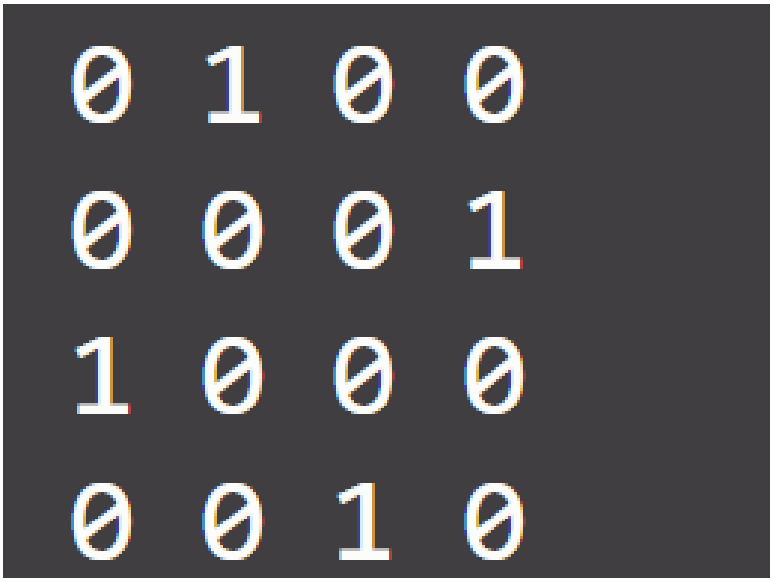
1. **Backtracking**

Code:

```
def is_safe(board, row, col, N):
    # Check if there is a queen in the same column
    for i in range(row):
        if board[i][col] == 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 upper diagonal on right side
    for i, j in zip(range(row, -1, -1), range(col, N)):
        if board[i][j] == 1:
            return False

    return True

def solve_n_queens(board, row, N):
    if row >= N:
        return True

    for col in range(N):
        if is_safe(board, row, col, N):
            board[row][col] = 1

            if solve_n_queens(board, row + 1, N):
                return True

            board[row][col] = 0

    return False
```

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

def n_queens_backtracking(N):
    board = [[0 for _ in range(N)] for _ in range(N)]
    if not solve_n_queens(board, 0, N):
        print("Solution does not exist")
        return False
    print_solution(board, N)
    return True

# Example usage:
n_queens_backtracking(4)
```

Output:



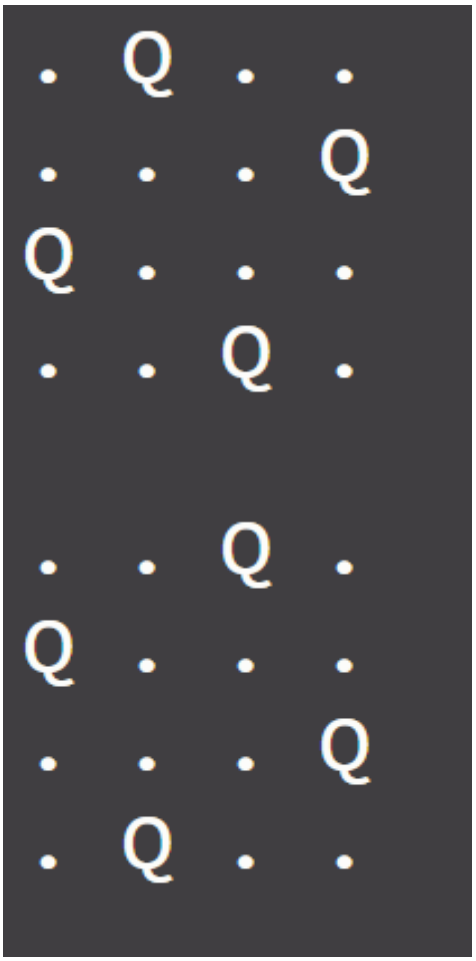
2. **Branch and Bound**

Code:

```
def is_safe(board, row, col, N):
    # Check if there is a queen in the same column
    for i in range(row):
        if board[i] == col or \
           board[i] == col - (row - i) or \
           board[i] == col + (row - i):
            return False
    return True

def solve_n_queens(board, row, N, solution):
    if row >= N:
        solution.append(board[:])
        return

    for col in range(N):
        if is_safe(board, row, col, N):
            board[row] = col
            solve_n_queens(board, row + 1, N, solution)

def n_queens_branch_and_bound(N):
    solution = []
    board = [-1] * N
    solve_n_queens(board, 0, N, solution)
    if not solution:
        print("Solution does not exist")
        return False
    for sol in solution:
        for col in sol:
            row_str = ['.'] * N
            row_str[col] = 'Q'
            print(' '.join(row_str))
        print()
    return True
```

# Example usage:
n_queens_branch_and_bound(4)

Output:



Practical-5

Code:

```
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

```
Hello! My name is Sbot.
I was created in 2021.
Please, remind me your name.
soham
What a great name you have, soham!
Let me guess your age.
Enter remainders of dividing your age by 3, 5 and 7.
2
0
6
Your age is 20; that's a good time to start programming!
Now I will prove to you that I can count to any number you want
```

```
Now I will prove to you that I can count to any number you want
.
10
0 !
1 !
2 !
3 !
4 !
5 !
6 !
7 !
8 !
9 !
10 !
```

```
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.
1
Please, try again.
2
Completed, have a nice day!
.....................................
.....................................
.....................................
Congratulations, have a nice day!
.....................................
```