AI Practical Slip Solutions (1)

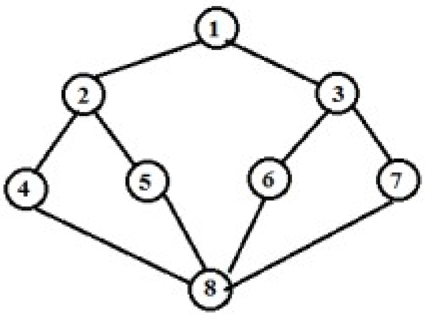
MSc(computer Science) (Savitribai Phule Pune University)

Slip 1:

Q.1) Python program that demonstrates the hill climbing algorithm to find the maximum of a
mathematical function.(For example f(x) = -x^2 + 4x)

```
import numpy as np

def objective_function(x):
    return -x**2 + 4*x
def hill_climbing(initial_x, step_size, iterations):
    current_x = initial_x
    for _ in range(iterations):
        current_value = objective_function(current_x)
        next_x = current_x + step_size
        next_value = objective_function(next_x)
    if next_value > current_value:
        current_x = next_x
    return current_x, objective_function(current_x)

# Example usage
initial_x = 0.0
step_size = 0.1
iterations = 50
max_x, max_value = hill_climbing(initial_x, step_size, iterations)
print(f"Maximum value found at x = {max_x}, f(x) = {max_value}")
```

Q.2) Write a Python program to implement Depth First Search algorithm. Refer the following graph
as an Input for the program. [Initial node=1,Goal node=8]




```
graph = {
    1: [2, 3],
    2: [4, 5],
    3: [6, 7],
    4: [8],
    5: [8],
    6: [8],
    7: [8],
    8: []
}

def dfs(graph, start, goal, visited=None, path=None):
```

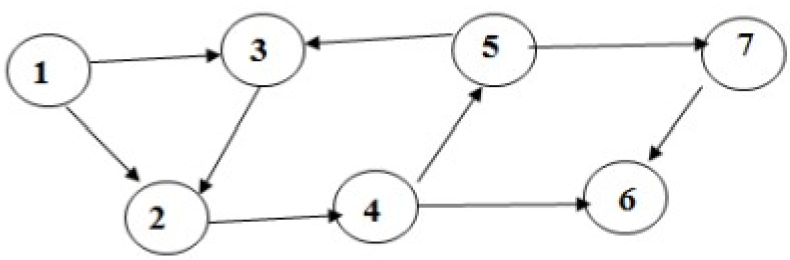
```
    if visited is None:
        visited = set()
    if path is None:
        path = []

    visited.add(start)
    path = path + [start]

    if start == goal:
        print("DFS Path:", path)
    else:
        for neighbor in graph[start]:
            if neighbor not in visited:
                dfs(graph, neighbor, goal, visited, path)

if __name__ == "__main__":
    initial_node = 1
    goal_node = 8
    dfs(graph, initial_node, goal_node)
```

Slip 2:

Q.1) Write a python program to generate Calendar for the given month and year?.

```
import calendar
def generate_calendar(year, month):
    cal = calendar.monthcalendar(year, month)
    print(f"Calendar for {calendar.month_name[month]} {year}:\n")
    print("Mo Tu We Th Fr Sa Su")
    for week in cal:
        week_str = " ".join(str(day) if day != 0 else " " for day in
week)
        print(week_str)
year = 2023
month = 4 # April
generate_calendar(year, month)
```

Q.2)Write a Python program to implement Depth First Search algorithm. Refer the following graph
as an Input for the program.[Initial node=1,Goal node=7].




```
graph = {
    1: [3, 2],
    3: [2],
    2: [4],
    4: [5, 6],
```

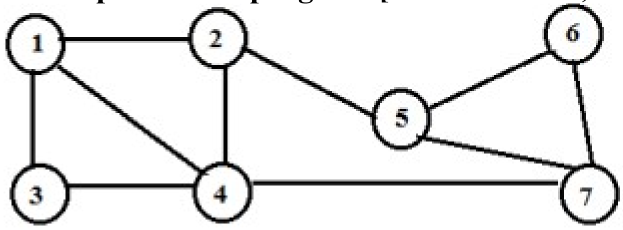
```python
    7: [6],
    5: [7, 3]
}

def dfs(graph, start, goal, visited=None, path=None):
    if visited is None:
        visited = set()
    if path is None:
        path = []

    visited.add(start)
    path = path + [start]

    if start == goal:
        print("DFS Path:", path)
    else:
        for neighbor in graph.get(start, []):
            if neighbor not in visited:
                dfs(graph, neighbor, goal, visited, path)

if __name__ == "__main__":
    initial_node = 1
    goal_node = 7
    dfs(graph, initial_node, goal_node)
```

Slip 3:

Q.1) Write a python program to remove punctuations from the given string?

```python
import string
def remove_punctuation(input_string):
    translator = str.maketrans("", "", string.punctuation)
    cleaned_string = input_string.translate(translator)
    return cleaned_string

input_string = "Hello, World! This is an example string."
cleaned_string = remove_punctuation(input_string)
print(f"Original String: {input_string}")
print(f"String without Punctuation: {cleaned_string}")
```

Q.2) Write a Python program to implement Depth First Search algorithm. Refer the following graph
as an Input for the program.[Initial node=2,Goal node=7]




```python
graph = {
1: [2, 3, 4],
2: [4, 5],
```

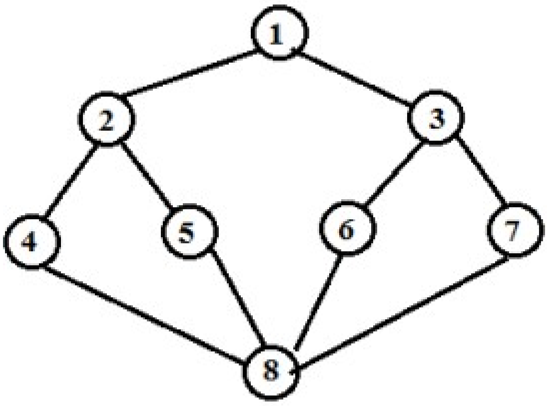
```
5: [6, 7],
6: [7],
4: [7],
3: [4]
}
def dfs(graph, start, goal, visited=None, path=None):
    if visited is None:
        visited = set()
    if path is None:
        path = []
    visited.add(start)
    path = path + [start]
    if start == goal:
        print("DFS Path:", path)
    else:
        for neighbor in graph.get(start, []):
            if neighbor not in visited:
                dfs(graph, neighbor, goal, visited, path)

if __name__ == "__main__":
    initial_node = 1
    goal_node = 7
    dfs(graph, initial_node, goal_node)
```

Slip 4:
Q.1)Write a program to implement Hangman game using python.
Description:
Hangman is a classic word-guessing game. The user should guess the word correctly by entering alphabets of the user choice. The Program will get input as single alphabet from the
user and it will matchmaking with the alphabets in the original

```
import random

def choose_word():
    # List of words for the game
    words = ["python", "hangman", "programming", "challenge",
"computer", "science"]
    # Choose a random word from the list
    return random.choice(words)

def display_word(word, guessed_letters):
    # Display the word with guessed letters and underscores for
unrevealed letters
    display = ""
    for letter in word:
        if letter in guessed_letters:
            display += letter + " "
        else:
            display += "_ "
    return display.strip()
```

```python
def hangman():
    # Welcome message
    print("Welcome to Hangman!")

    # Choose a random word
    secret_word = choose_word()

    # Initialize variables
    guessed_letters = []
    attempts = 6

    while attempts > 0:
        # Display current state of the word
        current_display = display_word(secret_word, guessed_letters)
        print(f"\n{current_display}")

        # Get user input for a letter
        guess = input("Guess a letter: ").lower()

        # Check if the guessed letter is in the word
        if guess.isalpha() and len(guess) == 1:
            if guess in guessed_letters:
                print("You already guessed that letter. Try again.")
            elif guess in secret_word:
                print("Good guess!")
                guessed_letters.append(guess)
            else:
                print("Incorrect guess. Try again.")
                attempts -= 1
        else:
            print("Invalid input. Please enter a single alphabet.")

        # Check if the word has been guessed
        if all(letter in guessed_letters for letter in secret_word):
            print(f"\nCongratulations! You guessed the word:
{secret_word}")
            break

        # Check if the player has run out of attempts
        if attempts == 0:
            print(f"\nSorry, you ran out of attempts. The word was:
{secret_word}")

# Run the Hangman game
hangman()
```

Q.2) Write a Python program to implement Breadth First Search algorithm. Refer the following
graph as an Input for the program.[Initial node=1,Goal node=8]




```python
graph = {
    1: [2, 3],
    2: [4, 5],
    3: [6, 7],
    4: [8],
    5: [8],
    6: [8],
    7: [8],
    8: []  # Goal node has no outgoing edges
}

def dfs(graph, start, goal, visited=None, path=None):
    if visited is None:
        visited = set()
    if path is None:
        path = []

    visited.add(start)
    path = path + [start]

    if start == goal:
        print("DFS Path:", path)
    else:
        for neighbor in graph[start]:
            if neighbor not in visited:
                dfs(graph, neighbor, goal, visited, path)

if __name__ == "__main__":
    initial_node = 1
    goal_node = 8
    dfs(graph, initial_node, goal_node)
```

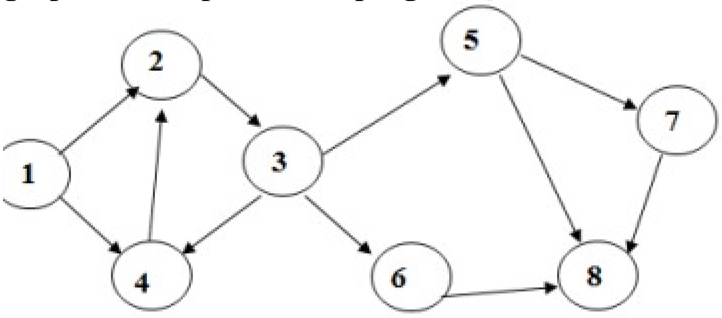
Slip 5:

Q.1) Write a python program to implement Lemmatization using NLTK

```python
import nltk
from nltk.stem import WordNetLemmatizer
from nltk.tokenize import word_tokenize

# Download necessary NLTK resources
```

```python
nltk.download('punkt_tab')       # Tokenizer models
nltk.download('wordnet')  # WordNet for lemmatization
nltk.download('omw-1.4')  # Open Multilingual WordNet (helps with more
languages)

def lemmatize_text(text):
    # Tokenize the input text into words
    words = word_tokenize(text)

    # Initialize the WordNetLemmatizer
    lemmatizer = WordNetLemmatizer()

    # Lemmatize each word in the tokenized list
    lemmatized_words = [lemmatizer.lemmatize(word) for word in words]

    # Join the lemmatized words into a single string
    lemmatized_text = ' '.join(lemmatized_words)

    return lemmatized_text

# Example input text
input_text = "The cats are running and the mice are hiding. The dogs
are barking."

# Call the function to lemmatize the input text
lemmatized_result = lemmatize_text(input_text)

# Output the results
print(f"Original Text: {input_text}")
print(f"Lemmatized Text: {lemmatized_result}")
```

Q.2) Write a Python program to implement Breadth First Search algorithm. Refer the following
graph as an Input for the program.[Initial node=1,Goal node=8]




```python
from collections import deque

graph = {
    1: [2, 4],
    4: [2],
    2: [3],
    3: [4, 5, 6],
```

```
    5: [7, 8],
    6: [8]
}

def bfs(graph, start, goal):
    visited = set()
    queue = deque([[start]])

    while queue:
        path = queue.popleft()
        node = path[-1]

        if node not in visited:
            neighbors = graph.get(node, [])

            for neighbor in neighbors:
                new_path = list(path)
                new_path.append(neighbor)
                queue.append(new_path)

                if neighbor == goal:
                    print("BFS Path:", new_path)
                    return

            visited.add(node)

if __name__ == "__main__":
    initial_node = 1
    goal_node = 8
    bfs(graph, initial_node, goal_node)
```

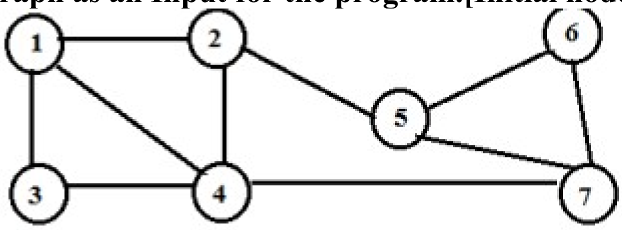
Slip 6:

Q.1) Write a python program to remove stop words for a given passage from a text file using NLTK?.

```
import nltk
from nltk.corpus import stopwords
from nltk.tokenize import word_tokenize

# Download NLTK stop words if not already downloaded
nltk.download('stopwords')

def remove_stop_words(input_text):
    words = word_tokenize(input_text)
    stop_words = set(stopwords.words('english'))
    filtered_words = [word for word in words if word.lower() not in
stop_words]
    filtered_text = ' '.join(filtered_words)

    return filtered_text

file_path = 'sample_text.txt'
```

```python
try:
    with open(file_path, 'r', encoding='utf-8') as file:
        passage = file.read()
    print(f"Original Passage:\n{passage}")
    filtered_passage = remove_stop_words(passage)
    print(f"\nPassage after removing stop words:\n{filtered_passage}")
except FileNotFoundError:
    print(f"Error: File '{file_path}' not found.")
except Exception as e:
    print(f"Error: {e}")
```

Q.2) Write a Python program to implement Breadth First Search algorithm. Refer the following
graph as an Input for the program.[Initial node=1,Goal node=8].




```python
graph = {
    1: [2, 3, 4],
    2: [4, 5],
    5: [6, 7],
    6: [7],
    4: [7],
    3: [4]
}

def dfs(graph, start, goal, visited=None, path=None):
    if visited is None:
        visited = set()
    if path is None:
        path = []

    visited.add(start)
    path = path + [start]

    if start == goal:
        print("DFS Path:", path)
    else:
        for neighbor in graph.get(start, []):
            if neighbor not in visited:
                dfs(graph, neighbor, goal, visited, path)

if __name__ == "__main__":
    initial_node = 1
    goal_node = 7
```

```
    dfs(graph, initial_node, goal_node)
```

Slip 7:

Q.1)Write a python program implement tic-tac-toe using alpha beeta pruning

```
import math

# Function to print the game board
def print_board(board):
    for row in board:
        print(" | ".join(row))
        print("-" * 5)

# Check if a player has won the game
def is_winner(board, player):
    # Check rows, columns, and diagonals for a win
    for i in range(3):
        if all(cell == player for cell in board[i]) or all(board[j][i]
== player for j in range(3)):
            return True
    if all(board[i][i] == player for i in range(3)) or all(board[i][2 -
i] == player for i in range(3)):
        return True
    return False

# Check if the board is full
def is_full(board):
    return all(cell != " " for row in board for cell in row)

# Evaluate the current board state (1 for O win, -1 for X win, 0 for
draw)
def evaluate(board):
    if is_winner(board, 'O'):
        return 1
    elif is_winner(board, 'X'):
        return -1
    else:
        return 0

# Alpha-Beta Pruning minimax algorithm
def minimax(board, depth, is_maximizing, alpha, beta):
    score = evaluate(board)

    # If the game is over (win or draw), return the score
    if score == 1 or score == -1 or is_full(board):
        return score

    if is_maximizing:
        max_eval = -math.inf
        for i in range(3):
            for j in range(3):
                if board[i][j] == " ":
```

```
                    board[i][j] = 'O'  # AI's turn
                    eval = minimax(board, depth + 1, False, alpha,
beta)
                    board[i][j] = " "
                    max_eval = max(max_eval, eval)
                    alpha = max(alpha, eval)
                    if beta <= alpha:
                        break
        return max_eval
    else:
        min_eval = math.inf
        for i in range(3):
            for j in range(3):
                if board[i][j] == " ":
                    board[i][j] = 'X'  # Human player's turn
                    eval = minimax(board, depth + 1, True, alpha, beta)
                    board[i][j] = " "
                    min_eval = min(min_eval, eval)
                    beta = min(beta, eval)
                    if beta <= alpha:
                        break
        return min_eval

# Function to find the best move for the AI using Alpha-Beta Pruning
def find_best_move(board):
    best_val = -math.inf
    best_move = (-1, -1)

    for i in range(3):
        for j in range(3):
            if board[i][j] == " ":
                board[i][j] = 'O'  # AI's turn
                move_val = minimax(board, 0, False, -math.inf,
math.inf)
                board[i][j] = " "
                if move_val > best_val:
                    best_move = (i, j)
                    best_val = move_val
    return best_move

# Function to play the game
def play_tic_tac_toe():
    board = [[" " for _ in range(3)] for _ in range(3)]
    current_player = 'X'  # Human starts first

    while True:
        print_board(board)

        if current_player == 'X':  # Human player
            row = int(input("Enter row (0, 1, or 2): "))
            col = int(input("Enter column (0, 1, or 2): "))
```

```
            if 0 <= row < 3 and 0 <= col < 3 and board[row][col] == "
":
                board[row][col] = current_player
                if is_winner(board, current_player):
                    print_board(board)
                    print("Player X wins!")
                    break
                current_player = 'O'
            else:
                print("Invalid move. Try again.")
        else:  # AI (Player 'O')
            print("AI's turn:")
            row, col = find_best_move(board)
            board[row][col] = current_player
            if is_winner(board, current_player):
                print_board(board)
                print("Player O (AI) wins!")
                break
            current_player = 'X'

        if is_full(board):
            print_board(board)
            print("It's a draw!")
            break

# Start the game
play_tic_tac_toe()
```

Q.2) Write a Python program to implement Simple Chatbot.

```
import random

def simple_chatbot():
    responses = {
        "hello": ["Hi there!", "Hello!", "Hey!"],
        "how are you": ["I'm good, thanks!", "I'm doing well.", "All
good!"],
        "bye": ["Goodbye!", "See you later!", "Bye!"],
        "default": ["I'm not sure how to respond.", "Could you say that
again?", "Sorry, I didn't get that."]
    }

    print("Simple Chatbot: Hi! Type 'bye' to exit.")

    while True:
        user_input = input("You: ").lower()

        if user_input == 'bye':
            print("Simple Chatbot: Goodbye!")
            break
        else:
            response = responses.get(user_input, responses["default"])
```

```
            print("Simple Chatbot:", random.choice(response))

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

Slip 8:
Q.1) Write a Python program to accept a string. Find and print the number of upper case alphabets
and lower case alphabets.

```
def count_upper_lower(input_string):
    # Initialize counters
    upper_count = 0
    lower_count = 0

    # Iterate through each character in the string
    for char in input_string:
        # Check if the character is an uppercase letter
        if char.isupper():
            upper_count += 1
        # Check if the character is a lowercase letter
        elif char.islower():
            lower_count += 1

    # Print the results
    print(f"Number of uppercase letters: {upper_count}")
    print(f"Number of lowercase letters: {lower_count}")

# Example usage
user_input = input("Enter a string: ")
count_upper_lower(user_input)

def print_board(board):
    for row in board:
        print(" | ".join(row))
        print("-" * 5)

def check_winner(board, player):
    for i in range(3):
        if all(board[i][j] == player for j in range(3)) or all(board[j]
[i] == player for j in range(3)):
            return True
    if all(board[i][i] == player for i in range(3)) or all(board[i][2 -
i] == player for i in range(3)):
        return True
    return False

def is_board_full(board):
    return all(cell != ' ' for row in board for cell in row)

def tic_tac_toe():
    board = [[' ' for _ in range(3)] for _ in range(3)]
```

```python
    current_player = 'X'

    while True:
        print_board(board)

        # Input validation for row and column
        try:
            row, col = map(int, input(f"Player {current_player}, enter
row and column (0, 1, or 2): ").split())

            # Check if input is within bounds
            if row not in range(3) or col not in range(3):
                print("Invalid input. Please enter values between 0 and
2 for both row and column.")
                continue

            if board[row][col] == ' ':
                board[row][col] = current_player
                if check_winner(board, current_player):
                    print_board(board)
                    print(f"Player {current_player} wins!")
                    break
                elif is_board_full(board):
                    print_board(board)
                    print("It's a draw!")
                    break
                current_player = 'O' if current_player == 'X' else 'X'
            else:
                print("Cell already occupied. Try again.")
        except ValueError:
            print("Invalid input. Please enter two integers separated
by a space.")

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

Slip 9:

Q.1) Write python program to solve 8 puzzle problem using A* algorithm

```python
import heapq

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

    def __lt__(self, other):
        return self.cost < other.cost
```

```python
    def heuristic(self):
        goal = [[1, 2, 3], [4, 5, 6], [7, 8, 0]]
        distance = 0
        for i in range(3):
            for j in range(3):
                value = self.state[i][j]
                if value != 0:
                    goal_row, goal_col = divmod(value - 1, 3)
                    distance += abs(i - goal_row) + abs(j - goal_col)
        return distance

    def get_neighbors(self):
        neighbors = []
        zero_row, zero_col = [(i, row.index(0)) for i, row in
enumerate(self.state) if 0 in row][0]
        moves = [(0, 1), (0, -1), (1, 0), (-1, 0)]

        for move in moves:
            new_row, new_col = zero_row + move[0], zero_col + move[1]
            if 0 <= new_row < 3 and 0 <= new_col < 3:
                new_state = [row[:] for row in self.state]
                new_state[zero_row][zero_col], new_state[new_row]
[new_col] = new_state[new_row][new_col], new_state[zero_row][zero_col]
                neighbors.append(PuzzleNode(new_state, self, move,
self.depth + 1))
        return neighbors

    def print_path(self):
        if self.parent:
            self.parent.print_path()
        print(f"Move {self.move}:")
        print(self)

    def __str__(self):
        return "\n".join(" ".join(map(str, row)) for row in self.state)

def solve_8_puzzle(initial_state):
    initial_node = PuzzleNode(initial_state)
    goal_state = [[1, 2, 3], [4, 5, 6], [7, 8, 0]]

    # A* search (using a priority queue)
    heap = [initial_node]
    visited = set()

    while heap:
        current_node = heapq.heappop(heap)
        if current_node.state == goal_state:
            current_node.print_path()
            return
        visited.add(tuple(map(tuple, current_node.state)))

        for neighbor in current_node.get_neighbors():
```

```
            if tuple(map(tuple, neighbor.state)) not in visited:
                heapq.heappush(heap, neighbor)

    print("No solution found.")

# Example usage
initial_puzzle = [[1, 2, 3], [4, 5, 6], [0, 7, 8]]  # Initial state
solve_8_puzzle(initial_puzzle)
```

Q.2) Write a Python program to solve water jug problem. 2 jugs with capacity 5 gallon and 7 gallon
are given with unlimited water supply respectively. The target to achieve is 4 gallon of water in
second jug.

```
def water_jug_problem(capacity_jug1, capacity_jug2, target):
    jug1 = 0
    jug2 = 0
    steps = []

    while jug2 != target:
        steps.append((jug1, jug2))

        if jug2 == 0:  # Fill jug2
            jug2 = capacity_jug2
        elif jug1 < capacity_jug1:  # Pour from jug2 to jug1
            transfer = min(jug2, capacity_jug1 - jug1)
            jug1 += transfer
            jug2 -= transfer
        else:  # Empty jug1
            jug1 = 0

    steps.append((jug1, jug2))  # Final step
    for step in steps:
        print(f"Jug 1: {step[0]} gallons, Jug 2: {step[1]} gallons")
    print(f"Target of {target} gallons achieved in Jug 2!")

if __name__ == "__main__":
    water_jug_problem(5, 7, 4)  # Capacities of the jugs and target
amount in jug2
```

Slip 10:
Q.1) Write Python program to implement crypt arithmetic problem TWO+TWO=FOUR

```
from itertools import permutations

def cryptarithmetic():
    # All possible digits (0 to 9)
    digits = range(10)

    # All the letters involved
```

```
    letters = 'TWOFRU'

    # Generate all possible permutations of digits for the 6 letters
    for perm in permutations(digits, len(letters)):
        # Create a mapping of letters to digits
        letter_to_digit = dict(zip(letters, perm))

        # Get the numbers corresponding to TWO, TWO, and FOUR
        TWO = letter_to_digit['T'] * 100 + letter_to_digit['W'] * 10 +
letter_to_digit['O']
        FOUR = letter_to_digit['F'] * 1000 + letter_to_digit['O'] * 100
+ letter_to_digit['U'] * 10 + letter_to_digit['R']

        # Check if TWO + TWO equals FOUR
        if TWO + TWO == FOUR:
            # If the equation is satisfied, print the solution
            print(f"TWO + TWO = FOUR")
            print(f"{TWO} + {TWO} = {FOUR}")
            print(f"Letter to digit mapping: {letter_to_digit}")
            return

    print("No solution found.")

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

Q.2) Write a Python program to implement Simple Chatbot.

```
import random

def simple_chatbot():
    responses = {
        "hello": ["Hi there!", "Hello!", "Hey!"],
        "how are you": ["I'm good, thanks!", "I'm doing well.", "All
good!"],
        "bye": ["Goodbye!", "See you later!", "Bye!"],
        "default": ["I'm not sure how to respond.", "Could you say that
again?", "Sorry, I didn't get that."]
    }

    print("Simple Chatbot: Hi! Type 'bye' to exit.")

    while True:
        user_input = input("You: ").lower()

        if user_input == 'bye':
            print("Simple Chatbot: Goodbye!")
            break
        else:
            response = responses.get(user_input, responses["default"])
            print("Simple Chatbot:", random.choice(response))
```

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

Slip 11:

Q.1) Write a python program using mean end analysis algorithmproblem of transforming a string of lowercase letters into another string.

```
def mean_end_analysis(start_str, target_str):
    current_str = start_str
    target_str_len = len(target_str)
    steps = []  # List to store the transformation steps

    # Continue until the current string matches the target string
    while current_str != target_str:
        print(f"Current string: {current_str}")
        print(f"Target string: {target_str}")

        # If the current string is shorter than the target, insert
characters
        if len(current_str) < target_str_len:
            # Insert characters from the target string
            for i in range(len(current_str), target_str_len):
                current_str += target_str[i]  # Add the next character
from target to current_str
                steps.append(f"Insert '{target_str[i]}' at position
{i+1}")
                break  # Insert one character at a time and then break
out

        # If the current string is longer than the target, delete extra
characters
        elif len(current_str) > target_str_len:
            current_str = current_str[:-1]  # Remove last character
            steps.append(f"Delete character '{current_str[-1]}'")

        # If characters at the same position are different, replace
them
        else:
            for i in range(len(current_str)):
                if current_str[i] != target_str[i]:
                    current_str = current_str[:i] + target_str[i] +
current_str[i+1:]
                    steps.append(f"Replace '{current_str[i]}' with
'{target_str[i]}' at position {i+1}")
                    break

    # Final output
    print(f"\nFinal transformed string: {current_str}")
    print(f"Steps taken:")
    for step in steps:
        print(step)
```

```
if __name__ == "__main__":
    # Example input: Transform "kitten" to "sitting"
    start_str = "kitten"
    target_str = "sitting"
    mean_end_analysis(start_str, target_str)
```

Q.2) Write a Python program to solve water jug problem. Two jugs with capacity 4 gallon and 3
gallon are given with unlimited water supply respectively. The target is to achieve 2 gallon of
water in second jug.

```
from collections import deque

def water_jug_problem(capacity1, capacity2, target):
    visited = set()
    queue = deque([(0, 0)])
    parent_map = { (0, 0): None }
    operations = [
        ("Fill Jug 1", lambda x, y: (capacity1, y)),
        ("Fill Jug 2", lambda x, y: (x, capacity2)),
        ("Empty Jug 1", lambda x, y: (0, y)),
        ("Empty Jug 2", lambda x, y: (x, 0)),
        ("Pour Jug 1 into Jug 2", lambda x, y: (x - min(x, capacity2 -
y), y + min(x, capacity2 - y))),
        ("Pour Jug 2 into Jug 1", lambda x, y: (x + min(y, capacity1 -
x), y - min(y, capacity1 - x)))
    ]

    while queue:
        x, y = queue.popleft()
        if y == target:
            solution = []
            while (x, y) != (0, 0):
                prev_x, prev_y = parent_map[(x, y)]
                for op_name, op_func in operations:
                    if op_func(prev_x, prev_y) == (x, y):
                        solution.append(op_name)
                        break
                x, y = prev_x, prev_y
            solution.reverse()
            return solution

        for op_name, op_func in operations:
            new_x, new_y = op_func(x, y)
            if (new_x, new_y) not in visited:
                visited.add((new_x, new_y))
                parent_map[(new_x, new_y)] = (x, y)
                queue.append((new_x, new_y))

    return None
```

```
capacity1 = 4
capacity2 = 3
target = 2

solution = water_jug_problem(capacity1, capacity2, target)

if solution:
    print("Steps to reach the goal:")
    for step in solution:
        print(step)
else:
    print("No solution exists")
```

Slip 12:

Q.1) Write a python program to generate Calendar for the given month and year?.

```
import calendar
def generate_calendar(year, month):
    cal = calendar.monthcalendar(year, month)
    print(f"Calendar for {calendar.month_name[month]} {year}:\n")
    print("Mo Tu We Th Fr Sa Su")
    for week in cal:
        week_str = " ".join(str(day) if day != 0 else " " for day in
week)
        print(week_str)
year = 2023
month = 4 # April
generate_calendar(year, month)
```

Q.2)Write a Python program to simulate 4-Queens problem.

```
def is_safe(board, row, col):
    for i in range(row):
        if board[i] == col or board[i] - i == col - row or board[i] + i
== col + row:
            return False
    return True

def solve_4_queens(board, row, solutions):
    if row == len(board):
        solutions.append(board[:])
        return

    for col in range(len(board)):
        if is_safe(board, row, col):
            board[row] = col
            solve_4_queens(board, row + 1, solutions)
            board[row] = -1

def print_solutions(solutions):
    for solution in solutions:
        print("Solution:")
        for row in solution:
```

```
            board_row = ['Q' if i == row else '.' for i in
range(len(solution))]
            print(" ".join(board_row))
        print()

def main():
    n = 4
    board = [-1] * n
    solutions = []
    solve_4_queens(board, 0, solutions)
    print_solutions(solutions)

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

Slip 13:

Q.1)Write a Python program to implement Mini-Max Algorithm.

```
import math

PLAYER_X = 1
PLAYER_O = -1
EMPTY = 0

def print_board(board):
    for row in board:
        print(" | ".join(str(cell) if cell != EMPTY else " " for cell
in row))
        print("---------")

def check_winner(board, player):
    for row in range(3):
        if all([board[row][col] == player for col in range(3)]): return
True
    for col in range(3):
        if all([board[row][col] == player for row in range(3)]): return
True
    if all([board[i][i] == player for i in range(3)]): return True
    if all([board[i][2-i] == player for i in range(3)]): return True
    return False

def is_game_over(board):
    return check_winner(board, PLAYER_X) or check_winner(board,
PLAYER_O) or all(cell != EMPTY for row in board for cell in row)

def evaluate(board):
    if check_winner(board, PLAYER_X): return 1
    if check_winner(board, PLAYER_O): return -1
    return 0

def available_moves(board):
```

```
    return [(row, col) for row in range(3) for col in range(3) if
board[row][col] == EMPTY]

def minimax(board, is_maximizing_player):
    if is_game_over(board): return evaluate(board)

    if is_maximizing_player:
        best = -math.inf
        for row, col in available_moves(board):
            board[row][col] = PLAYER_X
            best = max(best, minimax(board, False))
            board[row][col] = EMPTY
        return best
    else:
        best = math.inf
        for row, col in available_moves(board):
            board[row][col] = PLAYER_O
            best = min(best, minimax(board, True))
            board[row][col] = EMPTY
        return best

def find_best_move(board):
    best_val = -math.inf
    best_move = None
    for row, col in available_moves(board):
        board[row][col] = PLAYER_X
        move_val = minimax(board, False)
        board[row][col] = EMPTY
        if move_val > best_val:
            best_val = move_val
            best_move = (row, col)
    return best_move

def play_game():
    board = [[EMPTY for _ in range(3)] for _ in range(3)]
    while not is_game_over(board):
        print_board(board)
        if sum(row.count(PLAYER_X) for row in board) <=
sum(row.count(PLAYER_O) for row in board):
            row, col = find_best_move(board)
            board[row][col] = PLAYER_X
        else:
            row, col = find_best_move(board)
            board[row][col] = PLAYER_O
    print_board(board)
    if check_winner(board, PLAYER_X):
        print("Player X wins!")
    elif check_winner(board, PLAYER_O):
        print("Player O wins!")
    else:
        print("It's a draw!")
```

```
play_game()
```

Q.2) Write a Python program to simulate 8-Queens problem.

```
def print_solution(board):
    for row in board:
        print(" ".join("Q" if x else "." for x in row))

def is_safe(board, row, col):
    for i in range(row):
        if board[i][col] == 1:
            return False
        if col - (row - i) >= 0 and board[i][col - (row - i)] == 1:
            return False
        if col + (row - i) < len(board) and board[i][col + (row - i)]
== 1:
            return False
    return True

def solve_n_queens(board, row):
    if row == len(board):
        print_solution(board)
        return True

    for col in range(len(board)):
        if is_safe(board, row, col):
            board[row][col] = 1
            if solve_n_queens(board, row + 1):
                return True
            board[row][col] = 0
    return False

def n_queens(n):
    board = [[0 for _ in range(n)] for _ in range(n)]
    solve_n_queens(board, 0)

n_queens(8)
```

Slip 14:

Q.1) Write a python program to sort the sentence in alphabetical order?

```
def sort_sentence(sentence):
    words = sentence.split()
    words.sort()
    return " ".join(words)

sentence = "the quick brown fox jumps over the lazy dog"
sorted_sentence = sort_sentence(sentence)
print(sorted_sentence)
```

Q.2) Write a Python program to simulate n-Queens problem.

```
def print_solution(board):
    for row in board:
        print(" ".join("Q" if x else "." for x in row))

def is_safe(board, row, col):
    for i in range(row):
        if board[i][col] == 1:
            return False
        if col - (row - i) >= 0 and board[i][col - (row - i)] == 1:
            return False
        if col + (row - i) < len(board) and board[i][col + (row - i)]
== 1:
            return False
    return True

def solve_n_queens(board, row):
    if row == len(board):
        print_solution(board)
        return True

    for col in range(len(board)):
        if is_safe(board, row, col):
            board[row][col] = 1
            if solve_n_queens(board, row + 1):
                return True
            board[row][col] = 0
    return False

def n_queens(n):
    board = [[0 for _ in range(n)] for _ in range(n)]
    solve_n_queens(board, 0)

n_queens(8)
```

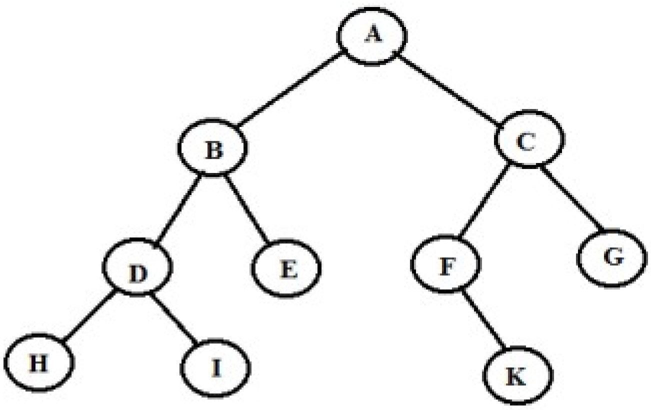
Slip 15:

Q.1)Write a Program to Implement Monkey Banana Problem using Python

```
class MonkeyBanana:
    def __init__(self):
        self.monkey_position = 'ground'
        self.box_position = 'floor'
        self.banana_position = 'high_shelf'
        self.monkey_has_banana = False

    def push_box(self):
        if self.box_position == 'floor':
            print("Monkey pushes the box to reach the shelf.")
            self.box_position = 'near_banana'

    def climb_box(self):
        if self.box_position == 'near_banana':
            print("Monkey climbs the box.")
```

```python
            self.monkey_position = 'on_box'

    def reach_banana(self):
        if self.monkey_position == 'on_box':
            print("Monkey grabs the banana from the shelf!")
            self.monkey_has_banana = True

    def perform_actions(self):
        print("Monkey is on the ground.")
        self.push_box()
        self.climb_box()
        self.reach_banana()

# Run the simulation
monkey_problem = MonkeyBanana()
monkey_problem.perform_actions()
```

Q.2) Write a program to implement Iterative Deepening DFS algorithm.
[Goal Node =G]




```python
class IterativeDeepeningDFS:
    def __init__(self, graph, start, goal):
        self.graph = graph
        self.start = start
        self.goal = goal

    def dfs(self, node, depth, visited):
        if depth == 0:
            if node == self.goal:
                return [node]
            return None
        if depth > 0:
            visited.add(node)
            for neighbor in self.graph.get(node, []):
                if neighbor not in visited:
                    path = self.dfs(neighbor, depth - 1, visited)
                    if path:
                        return [node] + path
        return None
```

```
    def iddfs(self):
        depth = 0
        while True:
            visited = set()
            result = self.dfs(self.start, depth, visited)
            if result:
                return result
            depth += 1


# Example usage:

# Graph provided by the user
graph = {
    'A': ['B', 'C'],
    'B': ['D', 'E'],
    'C': ['F', 'G'],
    'D': ['H', 'I'],
    'E': ['G'],
    'F': ['K'],
    'G': []
}

# Goal is to find node 'G' starting from node 'A'
iddfs = IterativeDeepeningDFS(graph, 'A', 'G')
path = iddfs.iddfs()

if path:
    print(f"Path to goal: {' -> '.join(path)}")
else:
    print("Goal not found.")
```

slip 16

Q.1) Write a Program to Implement Tower of Hanoi using Python

```
def tower_of_hanoi(n, source, destination, auxiliary):
    if n == 1:
        print(f"Move disk 1 from {source} to {destination}")
        return
    tower_of_hanoi(n-1, source, auxiliary, destination)
    print(f"Move disk {n} from {source} to {destination}")
    tower_of_hanoi(n-1, auxiliary, destination, source)

if __name__ == "__main__":
    n = int(input("Enter the number of disks: "))
    tower_of_hanoi(n, 'A', 'C', 'B')
```

Q.2) Write a Python program to solve tic-tac-toe problem.

```
def print_board(board):
    for row in board:
        print(" | ".join(row))
        print("-" * 5)
```

```
def check_winner(board, player):
    for i in range(3):
        if all([board[i][j] == player for j in range(3)]) or
all([board[j][i] == player for j in range(3)]):
            return True
    if board[0][0] == player and board[1][1] == player and board[2][2]
== player:
        return True
    if board[0][2] == player and board[1][1] == player and board[2][0]
== player:
        return True
    return False

def is_full(board):
    return all([board[i][j] != " " for i in range(3) for j in
range(3)])

def play_game():
    board = [[" " for _ in range(3)] for _ in range(3)]
    current_player = "X"
    while True:
        print_board(board)
        try:
            row = int(input("Enter row (0, 1, or 2): "))
            col = int(input("Enter column (0, 1, or 2): "))
        except ValueError:
            continue
        if row not in range(3) or col not in range(3) or board[row]
[col] != " ":
            continue
        board[row][col] = current_player
        if check_winner(board, current_player):
            print_board(board)
            print(f"Player {current_player} wins!")
            break
        if is_full(board):
            print_board(board)
            print("It's a draw!")
            break
        current_player = "O" if current_player == "X" else "X"

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

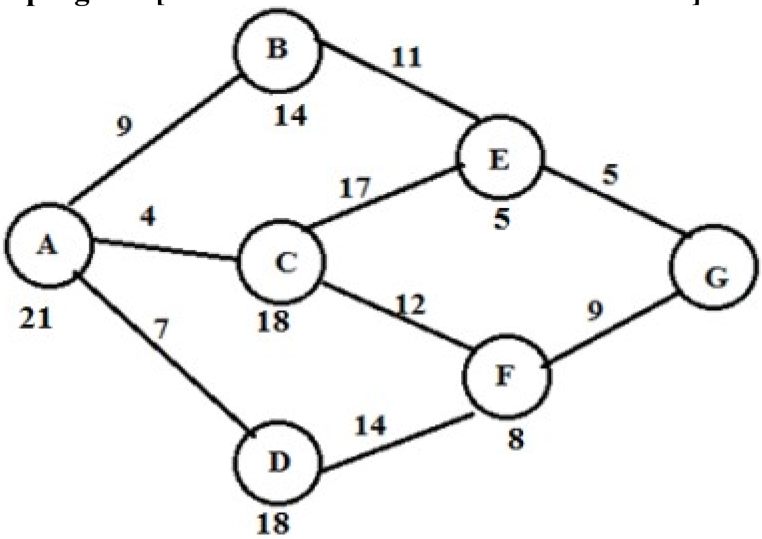
Slip 17

Q.1) Python program that demonstrates the hill climbing algorithm to find the maximum of a mathematical function.

```
import random
```

```python
def objective_function(x):
    return -(x - 3)**2 + 5

def hill_climbing(start, step_size, max_iterations):
    current = start
    for _ in range(max_iterations):
        neighbors = [current - step_size, current + step_size]
        next_move = max(neighbors, key=objective_function)
        if objective_function(next_move) <=
objective_function(current):
            break
        current = next_move
    return current, objective_function(current)

if __name__ == "__main__":
    start_point = random.uniform(-10, 10)
    step_size = 0.1
    max_iterations = 100
    solution, value = hill_climbing(start_point, step_size,
max_iterations)
    print(f"Best solution found: x = {solution}, f(x) = {value}")
```

Q.2) Write a Python program to implement A* algorithm. Refer the following graph as an Input for
the program.[Start vertex is A and Goal Vertex is G]



Slip 18

Q.1).Write a python program to remove stop words for a given passage from a text file using
NLTK?.

```python
import nltk
from nltk.corpus import stopwords
from nltk.tokenize import word_tokenize
import string

nltk.download('punkt')
nltk.download('stopwords')
```

```
def remove_stop_words(file_path):
    with open(file_path, 'r') as file:
        text = file.read()

    words = word_tokenize(text)
    stop_words = set(stopwords.words('english'))
    filtered_words = [word for word in words if word.lower() not in
stop_words and word.isalpha()]
    return ' '.join(filtered_words)

if __name__ == "__main__":
    file_path = 'input.txt'
    filtered_text = remove_stop_words(file_path)
    print("Filtered Text:")
    print(filtered_text)
```

Q.2) Implement a system that performs arrangement of some set of objects in a room. Assume that
you have only 5 rectangular, 4 square-shaped objects. Use A* approach for the placement of
the objects in room for efficient space utilisation. Assume suitable heuristic, and dimensions of
objects and rooms. (Informed Search)

```
import heapq

# Room dimensions (10x10)
room_width = 10
room_height = 10

# Dimensions of rectangular and square objects
rectangular_objects = [(2, 4)] * 5  # 5 objects of size 2x4
square_objects = [(3, 3)] * 4       # 4 objects of size 3x3

# Heuristic function: the remaining free area in the room
def heuristic(placed_objects):
    used_area = sum([w * h for _, _, w, h in placed_objects])  # Unpack
(x, y, width, height)
    total_area = room_width * room_height
    return total_area - used_area

# State representation: positions and orientations of objects
class State:
    def __init__(self, placed_objects, g_cost=0):
        self.placed_objects = placed_objects  # List of (x, y, width,
height) of placed objects
        self.g_cost = g_cost                  # Number of steps taken
(i.e., objects placed)
        self.h_cost = heuristic(placed_objects)  # Heuristic based on
remaining space
        self.f_cost = self.g_cost + self.h_cost  # Total cost (g + h)
```

```python
    def __lt__(self, other):
        return self.f_cost < other.f_cost

# Generate valid placements for an object
def generate_actions(placed_objects):
    actions = []
    all_objects = rectangular_objects + square_objects

    for idx, (w, h) in enumerate(all_objects):
        if (w, h) not in placed_objects:
            for x in range(room_width - w + 1):  # Try placing
horizontally
                for y in range(room_height - h + 1):  # Try placing
vertically
                    # Check for overlap
                    overlap = False
                    for px, py, pw, ph in placed_objects:
                        if not (x + w <= px or px + pw <= x or y + h <=
py or py + ph <= y):
                            overlap = True
                            break
                    if not overlap:
                        actions.append((x, y, w, h))  # Add valid
position to place this object
    return actions

# A* algorithm to find the optimal arrangement
def a_star():
    initial_state = State(placed_objects=[])
    open_list = []
    closed_list = set()
    heapq.heappush(open_list, initial_state)

    while open_list:
        current_state = heapq.heappop(open_list)

        # Check if goal reached (all objects placed)
        if len(current_state.placed_objects) ==
len(rectangular_objects) + len(square_objects):
            return current_state.placed_objects

        if tuple(current_state.placed_objects) in closed_list:
            continue

        closed_list.add(tuple(current_state.placed_objects))

        # Generate possible actions (placements of objects)
        actions = generate_actions(current_state.placed_objects)
        for action in actions:
            new_placed_objects = current_state.placed_objects +
[(action[0], action[1], action[2], action[3])]
```

```
            new_state = State(new_placed_objects, 
g_cost=current_state.g_cost + 1)
            if tuple(new_placed_objects) not in closed_list:
                heapq.heappush(open_list, new_state)

    return None

# Start the A* search to arrange the objects
placed_objects = a_star()

if placed_objects:
    print("Optimal placement found:")
    for placement in placed_objects:
        print(f"Object placed at x={placement[0]}, y={placement[1]} 
with width={placement[2]} and height={placement[3]}")
else:
    print("No valid arrangement found.")
```

slip 19
Write a program to implement Hangman game using python.
Description:
Hangman is a classic word-guessing game. The user should guess the word correctly by entering alphabets of the user choice. The Program will get input as single alphabet from the
user and it will matchmaking with the alphabets in the original word.

```
import random

words_list = ['python', 'java', 'javascript', 'hangman', 'computer', 
'programming', 'developer', 'algorithm', 'data']

def display_word(word, guessed_letters):
    display = ''
    for letter in word:
        if letter in guessed_letters:
            display += letter + ' '
        else:
            display += '_ '
    return display.strip()

def hangman():
    word = random.choice(words_list).lower()
    guessed_letters = set()
    max_attempts = 6
    attempts_left = max_attempts
    word_set = set(word)

    print("Welcome to Hangman!")
    print(f"Try to guess the word. You have {max_attempts} attempts.")

    while attempts_left > 0:
        print("\nWord: " + display_word(word, guessed_letters))
```

```
        print(f"Guessed Letters: {', '.join(sorted(guessed_letters))}")
        print(f"Attempts Left: {attempts_left}")

        guess = input("Guess a letter: ").lower()

        if len(guess) != 1 or not guess.isalpha():
            print("Please enter a single alphabet.")
            continue

        if guess in guessed_letters:
            print("You already guessed that letter. Try again.")
            continue

        guessed_letters.add(guess)

        if guess in word_set:
            print(f"Good job! {guess} is in the word.")
        else:
            attempts_left -= 1
            print(f"Oops! {guess} is not in the word.")

        if word_set.issubset(guessed_letters):
            print(f"\nCongratulations! You've guessed the word:
{word}")
            break

    if attempts_left == 0:
        print(f"\nGame Over! The word was: {word}")

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

Slip 20

Q.1) Build a bot which provides all the information related to you in college

```
class CollegeBot:
    def __init__(self, name, college_name, courses, timetable, grades):
        self.name = name
        self.college_name = college_name
        self.courses = courses
        self.timetable = timetable
        self.grades = grades

    def get_personal_info(self):
        return f"Name: {self.name}\nCollege: {self.college_name}"

    def get_courses(self):
        courses_str = "\n".join(self.courses)
        return f"Courses enrolled:\n{courses_str}"

    def get_timetable(self):
```

```
        timetable_str = "\n".join([f"{day}: {schedule}" for day,
schedule in self.timetable.items()])
        return f"Your timetable:\n{timetable_str}"

    def get_grades(self):
        grades_str = "\n".join([f"{course}: {grade}" for course, grade
in self.grades.items()])
        return f"Your grades:\n{grades_str}"

    def respond(self, query):
        query = query.lower()
        if "name" in query or "who are you" in query:
            return self.get_personal_info()
        elif "courses" in query or "enrolled" in query:
            return self.get_courses()
        elif "timetable" in query or "schedule" in query:
            return self.get_timetable()
        elif "grades" in query or "marks" in query:
            return self.get_grades()
        else:
            return "Sorry, I didn't understand that. Please ask
something else."

def main():
    name = "John Doe"
    college_name = "XYZ University"
    courses = ["Mathematics", "Computer Science", "Physics", "History"]
    timetable = {
        "Monday": "9:00 AM - Math, 11:00 AM - CS",
        "Tuesday": "9:00 AM - Physics, 1:00 PM - History",
        "Wednesday": "10:00 AM - CS, 2:00 PM - Math",
        "Thursday": "9:00 AM - Physics, 1:00 PM - History",
        "Friday": "10:00 AM - CS, 12:00 PM - Math"
    }
    grades = {
        "Mathematics": "A",
        "Computer Science": "B+",
        "Physics": "A-",
        "History": "B"
    }

    bot = CollegeBot(name, college_name, courses, timetable, grades)

    print("Hello! I am your College Bot. You can ask me about your
college information.")
    print("Type 'exit' to end the conversation.\n")

    while True:
        user_query = input("You: ")

        if user_query.lower() == 'exit':
            print("Goodbye! Have a great day!")
```

```
            break

        response = bot.respond(user_query)
        print(f"Bot: {response}\n")

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

Q.2) Write a Python program to implement Mini-Max Algorithm.

```
import math

PLAYER_X = 1
PLAYER_O = -1
EMPTY = 0

def print_board(board):
    for row in board:
        print(" | ".join(str(cell) if cell != EMPTY else " " for cell
in row))
        print("---------")

def check_winner(board, player):
    for row in range(3):
        if all([board[row][col] == player for col in range(3)]): return
True
    for col in range(3):
        if all([board[row][col] == player for row in range(3)]): return
True
    if all([board[i][i] == player for i in range(3)]): return True
    if all([board[i][2-i] == player for i in range(3)]): return True
    return False

def is_game_over(board):
    return check_winner(board, PLAYER_X) or check_winner(board,
PLAYER_O) or all(cell != EMPTY for row in board for cell in row)

def evaluate(board):
    if check_winner(board, PLAYER_X): return 1
    if check_winner(board, PLAYER_O): return -1
    return 0

def available_moves(board):
    return [(row, col) for row in range(3) for col in range(3) if
board[row][col] == EMPTY]

def minimax(board, is_maximizing_player):
    if is_game_over(board): return evaluate(board)

    if is_maximizing_player:
        best = -math.inf
        for row, col in available_moves(board):
```

```python
            board[row][col] = PLAYER_X
            best = max(best, minimax(board, False))
            board[row][col] = EMPTY
        return best
    else:
        best = math.inf
        for row, col in available_moves(board):
            board[row][col] = PLAYER_O
            best = min(best, minimax(board, True))
            board[row][col] = EMPTY
        return best

def find_best_move(board):
    best_val = -math.inf
    best_move = None
    for row, col in available_moves(board):
        board[row][col] = PLAYER_X
        move_val = minimax(board, False)
        board[row][col] = EMPTY
        if move_val > best_val:
            best_val = move_val
            best_move = (row, col)
    return best_move

def play_game():
    board = [[EMPTY for _ in range(3)] for _ in range(3)]
    while not is_game_over(board):
        print_board(board)
        if sum(row.count(PLAYER_X) for row in board) <=
sum(row.count(PLAYER_O) for row in board):
            row, col = find_best_move(board)
            board[row][col] = PLAYER_X
        else:
            row, col = find_best_move(board)
            board[row][col] = PLAYER_O
    print_board(board)
    if check_winner(board, PLAYER_X):
        print("Player X wins!")
    elif check_winner(board, PLAYER_O):
        print("Player O wins!")
    else:
        print("It's a draw!")
play_game()
```