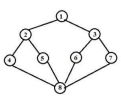
Slip 1

Q1. Python program that demonstrates the hill climbing algorithm to find the maximum of a mathematical function. (For example f(x)=-x^2+4x)

Ans:-

```
import random

def hill_climbing(function, initial_value, step_size, iterations):
    current_value = initial_value

    for _ in range(iterations):
        neighbors = [current_value - step_size, current_value, current_value + step_size]
        neighbors = [val for val in neighbors if val >= 0]  # Ensure x is non-negative
        current_value = max(neighbors, key=function, default=current_value)

    return current_value, function(current_value)

# Define the function
my_function = lambda x: -x**2 + 4*x

# Set initial parameters
initial_x = random.uniform(0, 10)
step_size = 0.1
iterations = 100

# Run hill climbing algorithm
result_x, result_value = hill_climbing(my_function, initial_x, step_size, iterations)

# Print results
print("Maximum found at x =", result_x)
print("Maximum value =", result_value)
```

Q2:-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]



Ans:-

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

# DFS traversal function
def dfs(graph, start, visited):
    if start not in visited:
        print(start, end=' ')
        visited.add(start)
        for neighbor in graph[start]:
            dfs(graph, neighbor, visited)

# Main function to initiate DFS traversal
def main():
    start_node = '1'  # You can change the starting node here
    print("Depth-First Search Traversal:")
    visited = set()
    dfs(graph, start_node, visited)

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

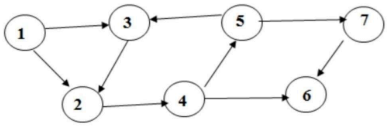
Slip2

Q1. Write a python program to generate Calendar for the given month and year?. [10 Marks]

Ans:-

```
import calendar

def generate_calendar(year, month):
    cal = calendar.monthcalendar(year, month)
    month_name = calendar.month_name[month]

    print(f"Calendar for {month_name} {year}:")

    # Print weekday names
    print("Mo Tu We Th Fr Sa Su")

    # Print each week
    for week in cal:
        week_str = ' '.join(str(day) if day != 0 else '  ' for day in week)
        print(week_str)

# Input: Year and Month
year = int(input("Enter the year: "))
month = int(input("Enter the month (1-12): "))

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].



Ans:-

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

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

# DFS traversal function
def dfs(graph, start, visited):
    if start not in visited:
        print(start, end=' ')
        visited.add(start)
        for neighbor in graph[start]:
            dfs(graph, neighbor, visited)

# Main function to initiate DFS traversal
def main():
    start_node = '1'  # You can change the starting node here
    print("Depth-First Search Traversal:")
    visited = set()
    dfs(graph, start_node, visited)

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

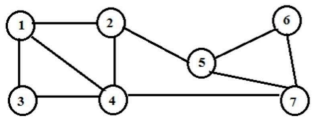
Slip 3

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

```
import string

def remove_punctuation(input_string):
    # Obtain the set of punctuation characters
    punctuation_set = set(string.punctuation)

    # Remove punctuation from the input string
    result_string = ''.join(char for char in input_string if char not in punctuation_set)

    return result_string
```

```
# Example usage
input_string = "Hello, World! This is an example string with punctuations!!!"
result = remove_punctuation(input_string)

print("Original String:", input_string)
print("String without Punctuation:", result)
```

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]



Ans:-

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

 # DFS traversal function
 def dfs(graph, start, visited):
    if start not in visited:
        print(start, end=' ')
        visited.add(start)
        for neighbor in graph[start]:
            dfs(graph, neighbor, visited)

 # Main function to initiate DFS traversal
 def main():
    start_node = '2'  # You can change the starting node here
    print("Depth-First Search Traversal:")
    visited = set()
```

```
    dfs(graph, start_node, visited)

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

Slip 4

Q.1)Write a program to implement Hangman game using python.
Description:

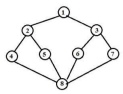
Hangman is a classic word-guessing game. The user should guess the word correctly by entering alphabets of the user choice. The Program will get input as single alphabet from the user and it will matchmaking with the alphabets in the original

Ans:-

```
import random

def choose_word():
    return random.choice(["python", "hangman", "programming", "code", "computer",
"algorithm"])

def display_word(word, guessed):
    return ' '.join(letter if letter in guessed else '_' for letter in word)

def hangman():
    word, guessed, attempts = choose_word(), set(), 6

    print("Welcome to Hangman!")

    while attempts > 0:
        print(display_word(word, guessed))
        guess = input("Enter a letter: ").lower()

        if guess.isalpha() and len(guess) == 1:
            guessed.add(guess)

            if guess not in word:
```

```
            attempts -= 1
            print(f"Wrong guess! Attempts left: {attempts}")
        elif all(letter in guessed for letter in word):
            print(f"Congratulations! You've guessed the word: {word}")
            break
    else:
        print("Please enter a single alphabet.")

if attempts == 0:
    print(f"Sorry, you're out of attempts. The word was: {word}")

if __name__ == "__main__":
    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]



Ans:-

```
from collections import deque

# Define an example graph as an adjacency list
graph = {
    '1': ['2', '3'],
    '2': ['1', '4', '5'],
    '3': ['1', '6','7'],
    '4': ['2','8'],
    '5': ['2', '8'],
    '6': ['3', '8'],
    '7': ['3', '8'],
    '8': ['4', '5','6,'7']
}

# BFS traversal function
def bfs(graph, start):
    visited = set()  # To keep track of visited nodes
```

```
    queue = deque()  # Create a queue for BFS

    visited.add(start)
    queue.append(start)

    while queue:
        node = queue.popleft()
        print(node, end=' ')

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

# Main function to initiate BFS traversal
def main():
    start_node = '1'  # You can change the starting node here
    print("Breadth-First Search Traversal:")
    bfs(graph, start_node)

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

Slip 5

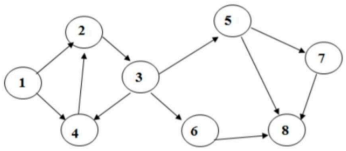
Q.1) Write a python program to implement Lemmatization using NLTK

Ans:-

```
import nltk
from nltk.corpus import wordnet
from nltk.stem import WordNetLemmatizer
from nltk.tokenize import word_tokenize

nltk.download('punkt')
nltk.download('wordnet')

def lemmatize_text(text):
    lemmatizer = WordNetLemmatizer()
    tokens = word_tokenize(text)
```

```
    lemmatized_tokens = [lemmatizer.lemmatize(token, get_pos_tag(token)) for token in
tokens]
    lemmatized_text = ' '.join(lemmatized_tokens)
    return lemmatized_text

def get_pos_tag(word):
    tag = nltk.pos_tag([word])[0][1][0].upper()
    tag_dict = {"J": wordnet.ADJ, "N": wordnet.NOUN, "V": wordnet.VERB, "R":
wordnet.ADV}
    return tag_dict.get(tag, wordnet.NOUN)

# Example usage
input_text = "The cats are running and playing in the garden"
lemmatized_text = lemmatize_text(input_text)

print("Original Text:", input_text)
print("Lemmatized Text:", lemmatized_text)
```

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]



Ans:

```
from collections import deque

 # Define an example graph as an adjacency list
 graph = {
     '1': ['2', '4'],
     '2': ['3'],
     '3': ['5', '6'],
     '4': ['2'],
     '5': ['7', '8'],
     '6': ['8'],
```

```
    '7': ['8']
}

# BFS traversal function
def bfs(graph, start):
    visited = set()  # To keep track of visited nodes
    queue = deque()  # Create a queue for BFS

    visited.add(start)
    queue.append(start)

    while queue:
        node = queue.popleft()
        print(node, end=' ')

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

# Main function to initiate BFS traversal
def main():
    start_node = '1'  # You can change the starting node here
    print("Breadth-First Search Traversal:")
    bfs(graph, start_node)

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

Slip 6

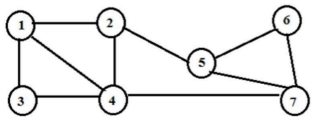
Q.1) Write a python program to remove stop words for a given passage from a text file using NLTK2.

Ans:-

```
import nltk
from nltk.corpus import stopwords
```

```
from nltk.tokenize import word_tokenize

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

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

    stop_words = set(stopwords.words('english'))
    tokens = word_tokenize(passage)
    filtered_tokens = [word for word in tokens if word.lower() not in stop_words]
    filtered_text = ' '.join(filtered_tokens)

    return filtered_text

# Example usage
file_path = 'path/to/your/textfile.txt'  # Replace with the actual path to your text file
result_text = remove_stop_words(file_path)

print("Original Passage:")
with open(file_path, 'r') as file:
    print(file.read())

print("\nPassage after removing stop words:")
print(result_text)
```

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]



Ans:-

```
from collections import deque

 # Define an example graph as an adjacency list
 graph = {
```

```
    '1': ['2', '3','4'],
    '2': ['1', '4', '5'],
    '3': ['1', '4'],
    '4': ['1','2','3'],
    '5': ['2', '6','7,],
    '6': ['5', '7'],
    '7': ['4', '5','6']
}

# BFS traversal function
def bfs(graph, start):
    visited = set()  # To keep track of visited nodes
    queue = deque()  # Create a queue for BFS

    visited.add(start)
    queue.append(start)

    while queue:
        node = queue.popleft()
        print(node, end=' ')

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

# Main function to initiate BFS traversal
def main():
    start_node = 'A'  # You can change the starting node here
    print("Breadth-First Search Traversal:")
    bfs(graph, start_node)

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

Slip 7

Q.1)Write a python program implement tic-tac-toe using alpha beeta pruning [10 Marks]

Ans:-

```
def print_board(board):
    for row in board:
        print(" | ".join(row))
        print("-" * 9)

# Function to check if a player has won
def check_win(board, player):
    for i in range(3):
        if all(board[i][j] == player for j in range(3)):  # Check rows
            return True
        if all(board[j][i] == player for j in range(3)):  # Check columns
            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)):  # Check diagonals
        return True
    return False

# Function to check if the board is full (a draw)
def check_draw(board):
    return all(cell != " " for row in board for cell in row)

# Main function to play the Tic-Tac-Toe game
def main():
    board = [[" " for _ in range(3)] for _ in range(3)]
    player = "X"
    win = False

    print("Tic-Tac-Toe Game:")
    print_board(board)

    while not win and not check_draw(board):
        print(f"Player {player}, enter your move (row and column):")
        row, col = map(int, input().split())

        if 1 <= row <= 3 and 1 <= col <= 3 and board[row - 1][col - 1] == " ":
            board[row - 1][col - 1] = player
            win = check_win(board, player)
```

```
            player = "O" if player == "X" else "X"
            print_board(board)
        else:
            print("Invalid move. Try again.")

    if win:
        print(f"Player {player} wins!")
    else:
        print("It's a draw!")

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

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

```
responses = {
    "hi": "Hello there! How can I help you today?",
    "hello": "Hi! How can I assist you?",
    "hey": "Hey! What can I do for you?",
    "how are you": "I'm just a computer program, but I'm here to help you.",
    "bye": "Goodbye! Have a great day.",
    "exit": "Goodbye! If you have more questions, feel free to come back."
}

# Chatbot function
def chatbot(user_input):
    user_input = user_input.lower()  # Convert the input to lowercase for case-insensitive matching
    response = responses.get(user_input, "I'm not sure how to respond to that. Please choose from the predefined inputs. 'hi', 'hello', 'hey', 'how are you', 'bye', 'exit'")
    return response

# Main loop for user interaction
print("Simple Chatbot: Type 'bye' to exit")
while True:
    user_input = input("You: ")
    if user_input.lower() == "bye" or user_input.lower() == "exit":
        print("Simple Chatbot: Goodbye!")
```

```
        break
    response = chatbot(user_input)
    print("Simple Chatbot:", response)
```

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.
Ans:-

```
def count_upper_lower(input_string):
    upper_count = sum(1 for char in input_string if char.isupper())
    lower_count = sum(1 for char in input_string if char.islower())
    return upper_count, lower_count

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

print(f"Number of uppercase alphabets: {upper}")
print(f"Number of lowercase alphabets: {lower}")
```

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

```
def print_board(board):
    for row in board:
        print(" | ".join(row))
        print("-" * 9)

 # Function to check if a player has won
 def check_win(board, player):
    for i in range(3):
        if all(board[i][j] == player for j in range(3)):  # Check rows
            return True
        if all(board[j][i] == player for j in range(3)):  # Check columns
            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)):  # Check diagonals
        return True
    return False

# Function to check if the board is full (a draw)
def check_draw(board):
    return all(cell != " " for row in board for cell in row)

# Main function to play the Tic-Tac-Toe game
def main():
    board = [[" " for _ in range(3)] for _ in range(3)]
    player = "X"
    win = False

    print("Tic-Tac-Toe Game:")
    print_board(board)

    while not win and not check_draw(board):
        print(f"Player {player}, enter your move (row and column):")
        row, col = map(int, input().split())

        if 1 <= row <= 3 and 1 <= col <= 3 and board[row - 1][col - 1] == " ":
            board[row - 1][col - 1] = player
            win = check_win(board, player)
            player = "O" if player == "X" else "X"
            print_board(board)
        else:
            print("Invalid move. Try again.")

    if win:
        print(f"Player {player} wins!")
    else:
        print("It's a draw!")

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

Slip 9

Q.1) Write python program to solve 8 puzzle problem using A algorithm [10 marks]
Ans:-

```
import heapq

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

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

def manhattan_distance(state):
    return sum(abs((val-1)%3 - i%3) + abs((val-1)//3 - i//3) for i, val in enumerate(state) if
val)

def neighbors(node):
    zero_i = node.state.index(0)
    moves = [1, -1, 3, -3]
    return [PuzzleNode(list(node.state[:zero_i] + [node.state[zero_i + m]] +
node.state[zero_i + m + 1:]), node, m, node.cost + 1, manhattan_distance(node.state))
for m in moves if 0 <= zero_i + m < 9 and (m == 1 or m == -1 or m == 3 or m == -3)]

def print_solution(node):
    moves = []
    while node.parent:
        moves.append("Move right" if node.move == 1 else "Move left" if node.move == -1
else "Move down" if node.move == 3 else "Move up")
        node = node.parent
    moves.reverse()
    print("Solution found!\n" + '\n'.join(moves))

def solve_puzzle(initial_state):
    goal, frontier, explored = [1, 2, 3, 4, 5, 6, 7, 8, 0], [PuzzleNode(initial_state, None,
None, 0, manhattan_distance(initial_state))], set()

    while frontier:
```

```
        current_node = heapq.heappop(frontier)

        if current_node.state == goal:
            print_solution(current_node)
            return

        explored.add(tuple(current_node.state))

        for neighbor in [n for n in neighbors(current_node) if tuple(n.state) not in explored]:
            heapq.heappush(frontier, neighbor)

    print("No solution found.")

if __name__ == "__main__":
    # Example usage:
    solve_puzzle([1, 2, 3, 4, 5, 6, 0, 7, 8])  # Replace with your initial state
```

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.
Ans:-

```
def water_jug_problem(capacity_x, capacity_y, target):
    jug_x = 0
    jug_y = 0

    while jug_x != target and jug_y != target:
        print(f"Jug X: {jug_x}L, Jug Y: {jug_y}L")

        # Fill jug X if it is empty
        if jug_x == 0:
            jug_x = capacity_x
            print("Fill Jug X")

        # Transfer water from jug X to jug Y if jug X is not empty
        elif jug_x > 0 and jug_y < capacity_y:
            transfer = min(jug_x, capacity_y - jug_y)
            jug_x -= transfer
            jug_y += transfer
```

```
            print("Transfer from Jug X to Jug Y")

        # Empty jug Y if it is full
        elif jug_y == capacity_y:
            jug_y = 0
            print("Empty Jug Y")

    print(f"Jug X: {jug_x}L, Jug Y: {jug_y}L")
    print("Solution Found!")

# Main function to initiate the problem
def main():
    capacity_x = 5  # Capacity of jug X
    capacity_y = 7  # Capacity of jug Y
    target = 4     # Amount of water to measure

    print("Solving Water Jug Problem:")
    water_jug_problem(capacity_x, capacity_y, target)

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

Slip 10

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

```
from itertools import permutations

def is_valid_assignment(mapping, word):
    return int(''.join(mapping[ch] for ch in word))

def solve_cryptarithmetic_puzzle():
    puzzle = ["TWO", "TWO", "FOUR"]
    unique_chars = set(''.join(puzzle))
    if len(unique_chars) > 10:
        print("Invalid puzzle: More than 10 unique characters.")
        return
```

```
    for perm in permutations("0123456789", len(unique_chars)):
        mapping = dict(zip(unique_chars, perm))
        if mapping[puzzle[0][0]] != '0' and is_valid_assignment(mapping, puzzle[0]) +
is_valid_assignment(mapping, puzzle[1]) == is_valid_assignment(mapping, puzzle[2]):
            print("Solution found:")
            for word in puzzle:
                print(f"{word}: {is_valid_assignment(mapping, word)}")
            return

    print("No solution found.")

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

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

```
responses = {
    "hi": "Hello there! How can I help you today?",
    "hello": "Hi! How can I assist you?",
    "hey": "Hey! What can I do for you?",
    "how are you": "I'm just a computer program, but I'm here to help you.",
    "bye": "Goodbye! Have a great day.",
    "exit": "Goodbye! If you have more questions, feel free to come back."
}

# Chatbot function
def chatbot(user_input):
    user_input = user_input.lower()  # Convert the input to lowercase for case-insensitive
matching
    response = responses.get(user_input, "I'm not sure how to respond to that. Please
choose from the predefined inputs. 'hi', 'hello', 'hey', 'how are you', 'bye', 'exit'")
    return response

# Main loop for user interaction
print("Simple Chatbot: Type 'bye' to exit")
while True:
    user_input = input("You: ")
```

```
    if user_input.lower() == "bye" or user_input.lower() == "exit":
        print("Simple Chatbot: Goodbye!")
        break
    response = chatbot(user_input)
    print("Simple Chatbot:", response)
```

Slip 11

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

```
def mean_end_analysis(initial, target):
    if len(initial) != len(target):
        print("Strings must have the same length.")
        return

    operations = []

    for i in range(len(initial)):
        if initial[i] != target[i]:
            operations.append(f"Change '{initial[i]}' to '{target[i]}' at position {i + 1}")

    if not operations:
        print("Strings are already the same.")
    else:
        print("Transformation Steps:")
        for operation in operations:
            print(operation)

if __name__ == "__main__":
    initial_string = input("Enter the initial string: ").lower()
    target_string = input("Enter the target string: ").lower()

    mean_end_analysis(initial_string, target_string)
```

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.
Ans:-

```
def water_jug_problem(capacity_x, capacity_y, target):
    jug_x = 0
    jug_y = 0

    while jug_x != target and jug_y != target:
        print(f"Jug X: {jug_x}L, Jug Y: {jug_y}L")

        # Fill jug X if it is empty
        if jug_x == 0:
            jug_x = capacity_x
            print("Fill Jug X")

        # Transfer water from jug X to jug Y if jug X is not empty
        elif jug_x > 0 and jug_y < capacity_y:
            transfer = min(jug_x, capacity_y - jug_y)
            jug_x -= transfer
            jug_y += transfer
            print("Transfer from Jug X to Jug Y")

        # Empty jug Y if it is full
        elif jug_y == capacity_y:
            jug_y = 0
            print("Empty Jug Y")

    print(f"Jug X: {jug_x}L, Jug Y: {jug_y}L")
    print("Solution Found!")

# Main function to initiate the problem
def main():
    capacity_x = 4  # Capacity of jug X
    capacity_y = 3  # Capacity of jug Y
    target = 2     # Amount of water to measure

    print("Solving Water Jug Problem:")
    water_jug_problem(capacity_x, capacity_y, target)
```

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

Slip 12

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

```
import calendar

def generate_calendar(year, month):
    cal = calendar.monthcalendar(year, month)
    month_name = calendar.month_name[month]

    print(f"Calendar for {month_name} {year}:")

    # Print weekday names
    print("Mo Tu We Th Fr Sa Su")

    # Print each week
    for week in cal:
        week_str = ' '.join(str(day) if day != 0 else '  ' for day in week)
        print(week_str)

# Input: Year and Month
year = int(input("Enter the year: "))
month = int(input("Enter the month (1-12): "))

generate_calendar(year, month)
```

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

```
def print_chessboard(chessboard):
    for row in chessboard:
        print(" ".join(row))

 # Function to check if it's safe to place a queen at the given position
```

```
def is_safe(chessboard, row, col, n):
    # Check row on the left side
    for i in range(col):
        if chessboard[row][i] == 'Q':
            return False

    # Check upper diagonal on the left side
    for i, j in zip(range(row, -1, -1), range(col, -1, -1)):
        if chessboard[i][j] == 'Q':
            return False

    # Check lower diagonal on the left side
    for i, j in zip(range(row, n, 1), range(col, -1, -1)):
        if chessboard[i][j] == 'Q':
            return False

    return True

# Recursive function to solve the Four Queens problem
def solve_four_queens(chessboard, col, n):
    if col >= n:
        return True  # All queens are placed

    for i in range(n):
        if is_safe(chessboard, i, col, n):
            chessboard[i][col] = 'Q'  # Place a queen

            # Recur to place the rest of the queens
            if solve_four_queens(chessboard, col + 1, n):
                return True

            # If placing a queen doesn't lead to a solution, backtrack
            chessboard[i][col] = '.'

    return False  # No solution exists

# Main function to solve the Four Queens problem
def main():
    n = 4  # Size of the chessboard (8x8)
    chessboard = [['.' for _ in range(n)] for _ in range(n)]
```

```
    if solve_four_queens(chessboard, 0, n):
        print("Solution to the Four Queens Problem:")
        print_chessboard(chessboard)
    else:
        print("No solution found.")

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

Slip 13

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

```
import math

def evaluate(board):
    return sum(row.count('X') - row.count('O') for row in board)

def is_terminal(board):
    return any(' ' not in row for row in board) or evaluate(board) != 0

def get_available_moves(board):
    return [(i, j) for i in range(3) for j in range(3) if board[i][j] == ' ']

def mini_max(board, depth, maximizing_player):
    if is_terminal(board):
        return evaluate(board)

    return max(mini_max(make_move(board, move, 'X'), depth + 1, False) if
maximizing_player else mini_max(make_move(board, move, 'O'), depth + 1, True) for
move in get_available_moves(board))

def find_best_move(board):
    return max(get_available_moves(board), key=lambda move:
mini_max(make_move(board, move, 'X'), 0, False))

def make_move(board, move, player):
```

```
    i, j = move
    new_board = [row.copy() for row in board]
    new_board[i][j] = player
    return new_board

def print_board(board):
    for row in board:
        print(" ".join(cell for cell in row))
    print()

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

    print("Initial Board:")
    print_board(board)

    for _ in range(4):  # Play four moves for demonstration
        player_move = tuple(map(int, input("Enter your move (row and column separated
by space): ").split()))
        board = make_move(board, player_move, 'O')

        print("Updated Board after your move:")
        print_board(board)

        if is_terminal(board):
            print("Game over!")
            break

        print("Computer's move:")
        computer_move = find_best_move(board)
        board = make_move(board, computer_move, 'X')

        print("Updated Board after computer's move:")
        print_board(board)

        if is_terminal(board):
            print("Game over!")
            break

if __name__ == "__main__":
```

```
play_game()
```

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

```
def print_chessboard(chessboard):
    for row in chessboard:
        print(" ".join(row))

# Function to check if it's safe to place a queen at the given position
def is_safe(chessboard, row, col, n):
    # Check row on the left side
    for i in range(col):
        if chessboard[row][i] == 'Q':
            return False

    # Check upper diagonal on the left side
    for i, j in zip(range(row, -1, -1), range(col, -1, -1)):
        if chessboard[i][j] == 'Q':
            return False

    # Check lower diagonal on the left side
    for i, j in zip(range(row, n, 1), range(col, -1, -1)):
        if chessboard[i][j] == 'Q':
            return False

    return True

# Recursive function to solve the Eight Queens problem
def solve_eight_queens(chessboard, col, n):
    if col >= n:
        return True  # All queens are placed

    for i in range(n):
        if is_safe(chessboard, i, col, n):
            chessboard[i][col] = 'Q'  # Place a queen

            # Recur to place the rest of the queens
            if solve_eight_queens(chessboard, col + 1, n):
```

```
                return True

            # If placing a queen doesn't lead to a solution, backtrack
            chessboard[i][col] = '.'

    return False  # No solution exists

# Main function to solve the Eight Queens problem
def main():
    n = 8  # Size of the chessboard (8x8)
    chessboard = [['.' for _ in range(n)] for _ in range(n)]

    if solve_eight_queens(chessboard, 0, n):
        print("Solution to the Eight Queens Problem:")
        print_chessboard(chessboard)
    else:
        print("No solution found.")

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

Slip 14

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

```
def sort_sentence(sentence):
    words = sentence.split()
    sorted_words = sorted(words)
    sorted_sentence = ' '.join(sorted_words)
    return sorted_sentence

if __name__ == "__main__":
    input_sentence = input("Enter a sentence: ")
    result = sort_sentence(input_sentence)
    print("Sorted Sentence:", result)
```

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

Ans:-

```
def print_chessboard(chessboard):
    for row in chessboard:
        print(" ".join(row))

 # Function to check if it's safe to place a queen at the given position
 def is_safe(chessboard, row, col, n):
    # Check the column
    for i in range(row):
        if chessboard[i][col] == 'Q':
            return False

    # Check the upper left diagonal
    for i, j in zip(range(row, -1, -1), range(col, -1, -1)):
        if chessboard[i][j] == 'Q':
            return False

    # Check the upper right diagonal
    for i, j in zip(range(row, -1, -1), range(col, n)):
        if chessboard[i][j] == 'Q':
            return False

    return True

 # Recursive function to solve the N-Queens problem using forward checking
 def solve_nqueens(chessboard, row, n):
    if row >= n:
        return True  # All queens are placed

    for col in range(n):
        if is_safe(chessboard, row, col, n):
            chessboard[row][col] = 'Q'  # Place a queen

            if solve_nqueens(chessboard, row + 1, n):
                return True

            chessboard[row][col] = '.'  # If placing a queen doesn't lead to a solution,
backtrack
```

```
    return False  # No solution exists

# Main function to perform N-Queens puzzle with forward checking
def main():
    n = 8  # Size of the chessboard (8x8)
    chessboard = [['.' for _ in range(n)] for _ in range(n)]

    print("N-Queens Puzzle using Forward Checking:")
    if solve_nqueens(chessboard, 0, n):
        print("\nSolution to the N-Queens Puzzle:")
        print_chessboard(chessboard)
    else:
        print("No solution found.")

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

Slip 15

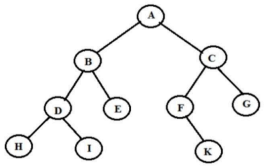
Q.1)Write a Program to Implement Monkey Banana Problem using Python
Ans:-

```
import queue

class State:
    def __init__(self, monkey_row, monkey_col, has_banana):
        self.monkey_row = monkey_row
        self.monkey_col = monkey_col
        self.has_banana = has_banana

def is_valid(state, rows, cols):
    return 0 <= state.monkey_row < rows and 0 <= state.monkey_col < cols

def is_goal(state, banana_row, banana_col):
    return state.monkey_row == banana_row and state.monkey_col == banana_col and
state.has_banana

def move(state, action):
    new_state = State(state.monkey_row, state.monkey_col, state.has_banana)
```

```
    if action == 'UP':
        new_state.monkey_row -= 1
    elif action == 'DOWN':
        new_state.monkey_row += 1
    elif action == 'LEFT':
        new_state.monkey_col -= 1
    elif action == 'RIGHT':
        new_state.monkey_col += 1
    elif action == 'GRAB':
        new_state.has_banana = True

    return new_state

def bfs(start_state, banana_row, banana_col, rows, cols):
    frontier = queue.Queue()
    frontier.put((start_state, []))

    while not frontier.empty():
        current_state, path = frontier.get()

        if is_goal(current_state, banana_row, banana_col):
            return path

        for action in ['UP', 'DOWN', 'LEFT', 'RIGHT', 'GRAB']:
            new_state = move(current_state, action)

            if is_valid(new_state, rows, cols):
                new_path = path + [action]
                frontier.put((new_state, new_path))

    return None

def print_solution(path):
    if path is None:
        print("No solution found.")
    else:
        print("Solution:")
        print(" -> ".join(path))
```

```
if __name__ == "__main__":
    rows = 4
    cols = 4

    monkey_start = (3, 0)
    banana_location = (0, 3)

    start_state = State(monkey_start[0], monkey_start[1], False)

    solution_path = bfs(start_state, banana_location[0], banana_location[1], rows, cols)
    print_solution(solution_path)
```

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



Ans:-

```
class Node:
    def __init__(self, state, children=None):
        self.state = state
        self.children = children if children else []

def depth_limited_dfs(node, goal_state, depth_limit, current_depth=0):
    if current_depth > depth_limit:
        return None

    if node.state == goal_state:
        return [node.state]

    for child in node.children:
        path = depth_limited_dfs(child, goal_state, depth_limit, current_depth + 1)
        if path is not None:
            return [node.state] + path
```

```
    return None

def iterative_deepening_dfs(root, goal_state):
    depth_limit = 0
    while True:
        result = depth_limited_dfs(root, goal_state, depth_limit)
        if result is not None:
            return result
        depth_limit += 1

if __name__ == "__main__":
    # Example usage:
    # Creating a simple tree structure for demonstration
    root = Node("A", [Node("B", [Node("D", [Node("G")])]), Node("C", [Node("E"),
Node("F", [Node("H", [Node("I")])])])])

    goal_node = "G"
    solution_path = iterative_deepening_dfs(root, goal_node)

    if solution_path:
        print("Solution Path:", " -> ".join(solution_path))
    else:
        print("No solution found.")
```

Slip 16

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

```
def tower_of_hanoi(n, source_peg, target_peg, auxiliary_peg):
    if n == 1:
        print(f"Move disk 1 from {source_peg} to {target_peg}")
        return
    tower_of_hanoi(n - 1, source_peg, auxiliary_peg, target_peg)
    print(f"Move disk {n} from {source_peg} to {target_peg}")
    tower_of_hanoi(n - 1, auxiliary_peg, target_peg, source_peg)

if __name__ == "__main__":
    number_of_disks = int(input("Enter the number of disks: "))
```

```
    tower_of_hanoi(number_of_disks, 'A', 'C', 'B')
```

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

```
def print_board(board):
    for row in board:
        print(" | ".join(row))
        print("-" * 9)

# Function to check if a player has won
def check_win(board, player):
    for i in range(3):
        if all(board[i][j] == player for j in range(3)):  # Check rows
            return True
        if all(board[j][i] == player for j in range(3)):  # Check columns
            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)):  # Check diagonals
        return True
    return False

# Function to check if the board is full (a draw)
def check_draw(board):
    return all(cell != " " for row in board for cell in row)

# Main function to play the Tic-Tac-Toe game
def main():
    board = [[" " for _ in range(3)] for _ in range(3)]
    player = "X"
    win = False

    print("Tic-Tac-Toe Game:")
    print_board(board)

    while not win and not check_draw(board):
        print(f"Player {player}, enter your move (row and column):")
        row, col = map(int, input().split())
```

```
        if 1 <= row <= 3 and 1 <= col <= 3 and board[row - 1][col - 1] == " ":
            board[row - 1][col - 1] = player
            win = check_win(board, player)
            player = "O" if player == "X" else "X"
            print_board(board)
        else:
            print("Invalid move. Try again.")

    if win:
        print(f"Player {player} wins!")
    else:
        print("It's a draw!")

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

Slip 17

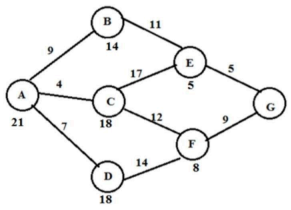
Q.1) Python program that demonstrates the hill climbing algorithm to find the maximum of a mathematical function.

Ans:-

```
def hill_climbing(function, initial_guess, step_size, max_iterations):
    current_solution = initial_guess
    current_value = function(current_solution)

    for _ in range(max_iterations):
        neighbor = current_solution + step_size
        neighbor_value = function(neighbor)

        if neighbor_value > current_value:
            current_solution = neighbor
            current_value = neighbor_value
        else:
            break

    return current_solution, current_value

# Example mathematical function (you can replace this with your own function)
def example_function(x):
```

```
    return -(x - 2) ** 2 + 5

if __name__ == "__main__":
    # Example usage:
    initial_guess = 0  # Initial guess for the maximum
    step_size = 0.1  # Step size for climbing
    max_iterations = 100  # Maximum number of iterations

    result_solution, result_value = hill_climbing(example_function, initial_guess,
step_size, max_iterations)

    print(f"Maximum Solution: {result_solution}")
    print(f"Maximum Value: {result_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]



Ans:-

```
import heapq

# Graph represented as an adjacency list
graph = {
    'A': {'B': 9, 'C': 4, 'D':7},
    'B': {'A': 9, 'E': 11},
    'C': {'A': 4, 'E': 17, 'F': 12},
    'D': {'A': 7, 'F': 14},
    'E': {'B': 11, 'C': 17, 'G': 5},
    'F': {'C': 12, 'D': 14,'G': 9},
    'G': {'E': 5, 'F': 9}
}

# Heuristic function (replace with your own heuristic)
heuristic = {
    'A': 21,
    'B': 14,
```

```
    'C': 18,
    'D': 18,
    'E': 5,
    'F': 8,
    'G': 0
}

def astar(start, goal):
    priority_queue = [(0, start)]
    visited = set()

    while priority_queue:
        current_cost, current_node = heapq.heappop(priority_queue)

        if current_node == goal:
            return current_cost

        if current_node not in visited:
            visited.add(current_node)

            for neighbor, edge_cost in graph[current_node].items():
                heuristic_cost = heuristic[neighbor]
                total_cost = current_cost + edge_cost + heuristic_cost
                heapq.heappush(priority_queue, (total_cost, neighbor))

    return float('inf')  # No path found

if __name__ == "__main__":
    start_vertex = 'A'
    goal_vertex = 'G'

    result_cost = astar(start_vertex, goal_vertex)

    if result_cost != float('inf'):
        print(f"Cost from {start_vertex} to {goal_vertex} using A* algorithm: {result_cost}")
    else:
        print(f"No path found from {start_vertex} to {goal_vertex}.")
```

Slip 18

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

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

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

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

if __name__ == "__main__":
    file_path = 'your_text_file.txt'  # Replace with your text file path

    try:
        with open(file_path, 'r', encoding='utf-8') as file:
            passage = file.read()

        cleaned_passage = remove_stop_words(passage)
        print("Original Passage:")
        print(passage)
        print("\nPassage after removing stop words:")
        print(cleaned_passage)

    except FileNotFoundError:
        print(f"File not found at path: {file_path}")
    except Exception as e:
        print(f"An error occurred: {e}")
```

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)
Ans:-

```
import heapq

class State:
    def __init__(self, room_width, room_height, remaining_objects, current_state=None):
        self.room_width = room_width
        self.room_height = room_height
        self.remaining_objects = remaining_objects

        if current_state:
            self.placed_objects = current_state.placed_objects.copy()
            self.total_wasted_space = current_state.total_wasted_space
        else:
            self.placed_objects = []
            self.total_wasted_space = 0

    def is_goal(self):
        return not self.remaining_objects

    def heuristic(self):
        # Simple heuristic: Minimize wasted space
        return self.total_wasted_space

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

def a_star(room_width, room_height, object_dimensions):
    initial_state = State(room_width, room_height, object_dimensions)
    priority_queue = [initial_state]

    while priority_queue:
        current_state = heapq.heappop(priority_queue)

        if current_state.is_goal():
            return current_state
```

```
        for obj_width, obj_height in current_state.remaining_objects:
            new_state = State(room_width, room_height, current_state.remaining_objects,
current_state)
            if room_width - obj_width >= 0 and room_height - obj_height >= 0:
                new_state.placed_objects.append((obj_width, obj_height))
                new_state.remaining_objects.remove((obj_width, obj_height))
                new_state.total_wasted_space += room_width * room_height - obj_width *
obj_height
                heapq.heappush(priority_queue, new_state)

    return None

if __name__ == "__main__":
    room_width = 10
    room_height = 8
    object_dimensions = [(3, 2), (2, 2), (4, 3), (1, 1), (2, 1)]

    result_state = a_star(room_width, room_height, object_dimensions)

    if result_state:
        print("Optimal arrangement:")
        print(result_state.placed_objects)
        print("Total wasted space:", result_state.total_wasted_space)
    else:
        print("No solution found.")
```

Slip 19

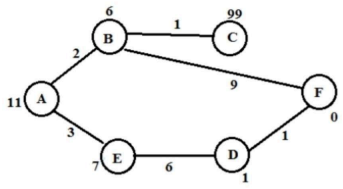
Q.1) Write a program to implement Hangman game using python. Hangman is a classic word-guessing game. The user should guess the word correctly by entering alphabets of the user choice. The Program will get input as single alphabet from the user and it will matchmaking with the alphabets in the original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

```
import random
```

```
def choose_word():
    words = ["python", "hangman", "programming", "developer", "computer"]
    return random.choice(words)

def display_word(word, guessed_letters):
    return ''.join(letter if letter in guessed_letters else '_' for letter in word)

def hangman():
    word_to_guess = choose_word().lower()
    guessed_letters = set()
    attempts_left = 6

    print("Welcome to Hangman!")
    print(display_word(word_to_guess, guessed_letters))

    while attempts_left > 0:
        user_guess = input("Enter a letter: ").lower()

        if len(user_guess) != 1 or not user_guess.isalpha():
            print("Please enter a valid single letter.")
            continue

        if user_guess in guessed_letters:
            print("You've already guessed that letter.")
            continue

        guessed_letters.add(user_guess)

        if user_guess not in word_to_guess:
            attempts_left -= 1
            print(f"Wrong guess! Attempts left: {attempts_left}")
        else:
            print("Correct guess!")

        print(display_word(word_to_guess, guessed_letters))

        if '_' not in display_word(word_to_guess, guessed_letters):
            print("Congratulations! You've guessed the word.")
            break
```

```
    if attempts_left == 0:
        print(f"Sorry, you've run out of attempts. The word was: {word_to_guess}")

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

Q.2) Write a Python program to implement A* algorithm. Refer the following graph as an Input for the program.



Ans:-

```
import heapq

# Graph represented as an adjacency list
graph = {
    'A': {'B': 2, 'E': 3},
    'B': {'A': 2, 'C': 1,'F':9},
    'C': {'B': 1},
    'D': {'E': 6, 'F': 1},
    'E': {'A': 3, 'D': 6},
    'F': {'B': 9, 'D': 1},
}

# Heuristic function (replace with your own heuristic)
heuristic = {
    'A': 11,
    'B': 6,
    'C': 99,
    'D': 1,
    'E': 7,
    'F': 0,
    }

def astar(start, goal):
```

```
    priority_queue = [(0, start)]
    visited = set()

    while priority_queue:
        current_cost, current_node = heapq.heappop(priority_queue)

        if current_node == goal:
            return current_cost

        if current_node not in visited:
            visited.add(current_node)

            for neighbor, edge_cost in graph[current_node].items():
                heuristic_cost = heuristic[neighbor]
                total_cost = current_cost + edge_cost + heuristic_cost
                heapq.heappush(priority_queue, (total_cost, neighbor))

    return float('inf')  # No path found

if __name__ == "__main__":
    start_vertex = 'A'
    goal_vertex = 'F'

    result_cost = astar(start_vertex, goal_vertex)

    if result_cost != float('inf'):
        print(f"Cost from {start_vertex} to {goal_vertex} using A* algorithm: {result_cost}")
    else:
        print(f"No path found from {start_vertex} to {goal_vertex}.")
```

Slip 20

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

```
from flask import Flask, request, jsonify

app = Flask(__name__)

# Replace these with your own information
your_name = "Your Name"
```

```
your_program = "Your Program"
your_year = "Your Year"
your_interests = ["Interest 1", "Interest 2", "Interest 3"]

@app.route('/college_bot', methods=['POST'])
def college_bot():
    data = request.get_json()

    if 'action' in data:
        action = data['action']

        if action == 'get_info':
            response = {
                'name': your_name,
                'program': your_program,
                'year': your_year,
                'interests': your_interests
            }
            return jsonify(response)
        else:
            return jsonify({'error': 'Invalid action'})

    return jsonify({'error': 'Action not provided'})

if __name__ == '__main__':
    app.run(debug=True)
```

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

```
import math

def mini_max(board, depth, is_maximizing_player):
    if check_winner(board):
        return evaluate(board)

    if is_maximizing_player:
        max_eval = -math.inf
        for move in possible_moves(board):
            board[move] = 'X'
```

```
            eval = mini_max(board, depth + 1, False)
            board[move] = ' '  # undo the move
            max_eval = max(max_eval, eval)
        return max_eval
    else:
        min_eval = math.inf
        for move in possible_moves(board):
            board[move] = 'O'
            eval = mini_max(board, depth + 1, True)
            board[move] = ' '  # undo the move
            min_eval = min(min_eval, eval)
        return min_eval

def check_winner(board):
    # Check for a winner or draw (you need to define this based on your game)
    return False

def evaluate(board):
    # Evaluate the current state of the board (you need to define this based on your
game)
    return 0

def possible_moves(board):
    # Return a list of possible moves (you need to define this based on your game)
    return []

if __name__ == "__main__":
    # Example usage:
    initial_board = [' '] * 9  # Assume a Tic-Tac-Toe board for simplicity
    best_move = -1
    best_value = -math.inf

    for move in possible_moves(initial_board):
        initial_board[move] = 'X'
        move_value = mini_max(initial_board, 0, False)
        initial_board[move] = ' '  # undo the move

        if move_value > best_value:
            best_value = move_value
            best_move = move
```

```
    print(f"The best move is {best_move} with a value of {best_value}")
```

Slip 21

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

```
import string

def remove_punctuation(input_string):
    return ''.join(char for char in input_string if char not in string.punctuation)

if __name__ == "__main__":
    input_string = "Hello, world! This is an example string."

    result = remove_punctuation(input_string)

    print("Original String:", input_string)
    print("String without Punctuation:", result)
```

Q.2) Write a Python program for the following Cryptarithmetic problems. GO + TO = OUT
Ans:-

```
from itertools import permutations

def is_solution(mapping):
    go = mapping['G'] * 10 + mapping['O']
    to = mapping['T'] * 10 + mapping['O']
    out = mapping['O'] * 100 + mapping['U'] * 10 + mapping['T']
    return go + to == out

def solve_cryptarithmetic():
    for p in permutations(range(10), 5):
        mapping = {'G': p[0], 'O': p[1], 'T': p[2], 'U': p[3], 'N': p[4]}
        if is_solution(mapping):
            return mapping
    return None
```

```
if __name__ == "__main__":
    solution = solve_cryptarithmetic()

    if solution:
        print("Solution found:")
        print(f"  G = {solution['G']}")
        print(f"  O = {solution['O']}")
        print(f"  T = {solution['T']}")
        print(f"  U = {solution['U']}")
        print(f"  N = {solution['N']}")
        print("\n  GO")
        print("+ TO")
        print("------")
        print(f" OUT")
    else:
        print("No solution found.")
```

Slip 22

Q.1) Write a Program to Implement Alpha-Beta Pruning using Python
Ans:-

```
import math

def alpha_beta_pruning(board, depth, alpha, beta, is_maximizing_player):
    if depth == 0 or game_over(board):
        return evaluate(board)

    if is_maximizing_player:
        max_eval = -math.inf
        for move in possible_moves(board):
            board[move] = 'X'
            eval = alpha_beta_pruning(board, depth - 1, alpha, beta, False)
            board[move] = ' '  # undo the move
            max_eval = max(max_eval, eval)
            alpha = max(alpha, eval)
            if beta <= alpha:
                break  # Beta cut-off
```

```
        return max_eval
    else:
        min_eval = math.inf
        for move in possible_moves(board):
            board[move] = 'O'
            eval = alpha_beta_pruning(board, depth - 1, alpha, beta, True)
            board[move] = ' '  # undo the move
            min_eval = min(min_eval, eval)
            beta = min(beta, eval)
            if beta <= alpha:
                break  # Alpha cut-off
        return min_eval

def game_over(board):
    # Implement your game-over condition (e.g., check for a winner or a draw)
    return False

def evaluate(board):
    # Implement your evaluation function based on the current state of the board
    return 0

def possible_moves(board):
    # Implement generating a list of possible moves based on the current state of the
board
    return []

if __name__ == "__main__":
    # Example usage:
    initial_board = [' '] * 9  # Assume a Tic-Tac-Toe board for simplicity
    depth_limit = 3

    best_move = -1
    best_value = -math.inf

    for move in possible_moves(initial_board):
        initial_board[move] = 'X'
        move_value = alpha_beta_pruning(initial_board, depth_limit - 1, -math.inf, math.inf,
False)
        initial_board[move] = ' '  # undo the move
```

```
        if move_value > best_value:
            best_value = move_value
            best_move = move

    print(f"The best move is {best_move} with a value of {best_value}")
```

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

```
responses = {
    "hi": "Hello there! How can I help you today?",
    "hello": "Hi! How can I assist you?",
    "hey": "Hey! What can I do for you?",
    "how are you": "I'm just a computer program, but I'm here to help you.",
    "bye": "Goodbye! Have a great day.",
    "exit": "Goodbye! If you have more questions, feel free to come back."
}

# Chatbot function
def chatbot(user_input):
    user_input = user_input.lower()  # Convert the input to lowercase for case-insensitive
matching
    response = responses.get(user_input, "I'm not sure how to respond to that. Please
choose from the predefined inputs. 'hi', 'hello', 'hey', 'how are you', 'bye', 'exit'")
    return response

# Main loop for user interaction
print("Simple Chatbot: Type 'bye' to exit")
while True:
    user_input = input("You: ")
    if user_input.lower() == "bye" or user_input.lower() == "exit":
        print("Simple Chatbot: Goodbye!")
        break
    response = chatbot(user_input)
    print("Simple Chatbot:", response)
```

Slip 23

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

Ans:-

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

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

Q.2) Write a Python program for the following Cryptarithmetic problems SEND + MORE = MONEY

Ans:-

```
from itertools import permutations

def is_solution(mapping):
    send = mapping['S'] * 1000 + mapping['E'] * 100 + mapping['N'] * 10 + mapping['D']
    more = mapping['M'] * 1000 + mapping['O'] * 100 + mapping['R'] * 10 + mapping['E']
    money = mapping['M'] * 10000 + mapping['O'] * 1000 + mapping['N'] * 100 +
mapping['E'] * 10 + mapping['Y']
    return send + more == money

def solve_cryptarithmetic():
    for p in permutations(range(10), 8):
        mapping = {'S': p[0], 'E': p[1], 'N': p[2], 'D': p[3], 'M': p[4], 'O': p[5], 'R': p[6], 'Y': p[7]}
        if is_solution(mapping):
            return mapping
    return None

if __name__ == "__main__":
    solution = solve_cryptarithmetic()

    if solution:
        print("Solution found:")
```

```
        print(f"  S = {solution['S']}")
        print(f"  E = {solution['E']}")
        print(f"  N = {solution['N']}")
        print(f"  D = {solution['D']}")
        print(f"  M = {solution['M']}")
        print(f"  O = {solution['O']}")
        print(f"  R = {solution['R']}")
        print(f"  Y = {solution['Y']}")
        print("\n  SEND")
        print("+ MORE")
        print("------")
        print(f" MONEY")
    else:
        print("No solution found.")
```

Slip 24

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

```
def sort_sentence(sentence):
    words = sentence.split()
    sorted_words = sorted(words)
    sorted_sentence = ' '.join(sorted_words)
    return sorted_sentence

if __name__ == "__main__":
    input_sentence = "This is a sample sentence to sort alphabetically."

    sorted_sentence = sort_sentence(input_sentence)

    print("Original Sentence:", input_sentence)
    print("Sorted Sentence:", sorted_sentence)
```

Q.2) Write a Python program for the following Crypt arithmetic problems
CROSS+ROADS = DANGER
Ans:-

```
from itertools import permutations

def is_solution(mapping):
    cross = mapping['C'] * 10000 + mapping['R'] * 1000 + mapping['O'] * 100 +
mapping['S'] * 10 + mapping['S']
    roads = mapping['R'] * 10000 + mapping['O'] * 1000 + mapping['A'] * 100 +
mapping['D'] * 10 + mapping['S']
    danger = mapping['D'] * 100000 + mapping['A'] * 10000 + mapping['N'] * 1000 +
mapping['G'] * 100 + mapping['E'] * 10 + mapping['R']
    return cross + roads == danger

def solve_cryptarithmetic():
    for p in permutations(range(10), 8):
        mapping = {'C': p[0], 'R': p[1], 'O': p[2], 'S': p[3], 'A': p[4], 'D': p[5], 'N': p[6], 'G': p[7],
'E': p[8]}
        if is_solution(mapping):
            return mapping
    return None

if __name__ == "__main__":
    solution = solve_cryptarithmetic()

    if solution:
        print("Solution found:")
        print(f"  C = {solution['C']}")
        print(f"  R = {solution['R']}")
        print(f"  O = {solution['O']}")
        print(f"  S = {solution['S']}")
        print(f"  A = {solution['A']}")
        print(f"  D = {solution['D']}")
        print(f"  N = {solution['N']}")
        print(f"  G = {solution['G']}")
        print(f"  E = {solution['E']}")
        print("\n  CROSS")
        print("+ ROADS")
        print("------")
        print(f" DANGER")
    else:
        print("No solution found.")
```

Slip 25

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

```
from flask import Flask, request, jsonify

app = Flask(__name__)

college_info = {
    "name": "Sample College",
    "location": "City, Country",
    "programs": ["Computer Science", "Business Administration", "Engineering"],
    "facilities": ["Library", "Gym", "Sports Fields"],
}

@app.route('/college_chatbot', methods=['POST'])
def college_chatbot():
    data = request.get_json()

    if 'query' in data:
        query = data['query'].lower()

        if 'name' in query:
            response = f"The college's name is {college_info['name']}."
        elif 'location' in query:
            response = f"The college is located in {college_info['location']}."
        elif 'programs' in query:
            response = f"The college offers programs in {', '.join(college_info['programs'])}."
        elif 'facilities' in query:
            response = f"The college provides facilities such as {',
'.join(college_info['facilities'])}."
        else:
            response = "I'm sorry, I don't understand that query."

        return jsonify({"response": response})

    return jsonify({'error': 'Query not provided'})

if __name__ == '__main__':
    app.run(debug=True)
```

Q.2) Write a Python program to solve 8-puzzle problem.
Ans:-

```
def print_chessboard(chessboard):
    for row in chessboard:
        print(" ".join(row))

# Function to check if it's safe to place a queen at the given position
def is_safe(chessboard, row, col, n):
    # Check row on the left side
    for i in range(col):
        if chessboard[row][i] == 'Q':
            return False

    # Check upper diagonal on the left side
    for i, j in zip(range(row, -1, -1), range(col, -1, -1)):
        if chessboard[i][j] == 'Q':
            return False

    # Check lower diagonal on the left side
    for i, j in zip(range(row, n, 1), range(col, -1, -1)):
        if chessboard[i][j] == 'Q':
            return False

    return True

# Recursive function to solve the Eight Queens problem
def solve_eight_queens(chessboard, col, n):
    if col >= n:
        return True  # All queens are placed

    for i in range(n):
        if is_safe(chessboard, i, col, n):
            chessboard[i][col] = 'Q'  # Place a queen

            # Recur to place the rest of the queens
            if solve_eight_queens(chessboard, col + 1, n):
                return True
```

```
            # If placing a queen doesn't lead to a solution, backtrack
            chessboard[i][col] = '.'

    return False  # No solution exists

# Main function to solve the Eight Queens problem
def main():
    n = 8  # Size of the chessboard (8x8)
    chessboard = [['.' for _ in range(n)] for _ in range(n)]

    if solve_eight_queens(chessboard, 0, n):
        print("Solution to the Eight Queens Problem:")
        print_chessboard(chessboard)
    else:
        print("No solution found.")

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