Slip 1,17

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

```
def objective_function(x):
    return -x**2 + 4*x
def hill_climbing(initial_x, step_size, max_iterations):
    current_x = initial_x
for iteration in range(max_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
        else:
            break
return current_x, objective_function(current_x)
initial_x = 0.0
step_size = 0.1
max_iterations = 100
result_x, result_value = hill_climbing(initial_x, step_size, max_iterations)
print(f"Maximum value found at x = {result_x}, f(x) = {result_value}")
```

Slip 1,2,3

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': ['1', '4', '5'],
    '3': ['1', '6','7'],
    '4': ['2','8'],
    '5': ['2', '8'],
    '6': ['3', '8'],
    '7':['3','8'],
    '8':['4','5','6','7']
}
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)
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 2,12

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:
        for day in week:
            if day == 0:
                print("   ", end=" ")  # Print empty spaces for days before the 1st
            else:
                print(f"{day:2} ", end=" ")  # Print day with padding
        print()  # Move to the next line after each week
# Input: Year and Month
year = int(input("Enter the year: "))
month = int(input("Enter the month (1-12): "))
generate_calendar(year, month)
```

Slip 3,21

Write a python program to remove punctuations from the given string?

import string

```
def remove_punctuation(input_string):
    # Create a translation table to map each punctuation character to None
    translator = str.maketrans('', '', string.punctuation)
    # Use the translation table to remove punctuations from the input string
    result_string = input_string.translate(translator)
    return result_string
# Input: String with punctuations
input_string = input("Enter a string with punctuations: ")
# Remove punctuations and print the result
result = remove_punctuation(input_string)
print("String after removing punctuations:", result)
```

Slip 16,13

Write a Program to Implement Tower of Hanoi using Python

```
def tower_of_hanoi(n, source, target, auxiliary):
    if n > 0:
        # Move n-1 disks from source to auxiliary peg
        tower_of_hanoi(n - 1, source, auxiliary, target)
        # Move the nth disk from source to target peg
        print(f"Move disk {n} from {source} to {target}")
        # Move the n-1 disks from auxiliary to target peg
        tower_of_hanoi(n - 1, auxiliary, target, source)
def main():
    num_disks = int(input("Enter the number of disks: "))
    tower_of_hanoi(num_disks, 'A', 'C', 'B')

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

slip 4,19

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

```
import random
def choose_word():
    words = ["python", "hangman", "programming", "computer", "developer", "gaming"]
    return random.choice(words)
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():
    print("Welcome to Hangman!")
    secret_word = choose_word()
    guessed_letters = []
    attempts = 6
    while attempts > 0:
        print("\nAttempts left:", attempts)
        print(display_word(secret_word, guessed_letters))
        guess = input("Enter a letter: ").lower()
        if len(guess) == 1 and guess.isalpha():
            if guess in guessed_letters:
                print("You already guessed that letter. Try again.")
            elif guess in secret_word:
                guessed_letters.append(guess)
                print("Good guess!")
            else:
                attempts -= 1
                print("Incorrect guess. Try again.")
        else:
            print("Please enter a single alphabet.")
        if all(letter in guessed_letters for letter in secret_word):
            print("\nCongratulations! You guessed the word:", secret_word)
            break
    if attempts == 0:
        print("\nSorry, you ran out of attempts. The correct word was:", secret_word)
if __name__ == "__main__":
    hangman()
```

slip 4,5,6

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]

```
from collections import deque
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']}
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)
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

Write a python program to implement Lemmatization using NLTK

```
import nltk
from nltk.stem import WordNetLemmatizer
from nltk.tokenize import word_tokenize
nltk.download('punkt')
nltk.download('wordnet')
def lemmatize_text(input_text):
   lemmatizer = WordNetLemmatizer()
   words = word_tokenize(input_text)
   lemmatized_words = [lemmatizer.lemmatize(word) for word in words]
   # Join the lemmatized words back into a sentence
   lemmatized_text = ' '.join(lemmatized_words)
   return lemmatized_text
# Input: Text to be lemmatized
input_text = input("Enter text for lemmatization: ")
# Perform lemmatization and print the result
result = lemmatize_text(input_text)
print("Lemmatized text:", result)
```

slip 6,18

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
nltk.download('stopwords')
nltk.download('punkt')
def remove_stopwords(text):
    stop_words = set(stopwords.words('english'))
    words = word_tokenize(text)
    filtered_words = [word for word in words if word.lower() not in stop_words]
    return ' '.join(filtered_words)
def main():
    file_path = 'path/to/your/text/file.txt'  # Replace with the actual file path
    with open(file_path, 'r') as file:
        passage = file.read()
    processed_passage = remove_stopwords(passage)
    print("\nOriginal Passage:\n", passage)
    print("\nPassage after Stopword Removal:\n", processed_passage)
if __name__ == "__main__":
    main()
```

Slip 7,10,22

Write a Python program to implement Simple Chatbot.

```
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 7,8,16

Write a python program implement tic-tac-toe using alpha beeta pruning

```
def print_board(board):
  for row in board:
    print(" | ".join(row))
    print("-" * 9)
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
def check_draw(board):
  return all(cell != " " for row in board for cell in row)
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:
    player = "O" if player == "X" else "X"
    print(f"Player {player} wins!")
  else:
    print("It's a draw!")
if __name__ == "__main__":
  main()
```

Slip 8

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(string):
    upper_count = 0
    lower_count = 0
    for char in string:
        if char.isupper():
            upper_count += 1
        elif char.islower():
            lower_count += 1
    return upper_count, lower_count
def main():
    input_string = input("Enter a string: ")
    upper, lower = count_upper_lower(input_string)
    print("\nNumber of Uppercase Alphabets:", upper)
    print("Number of Lowercase Alphabets:", lower)
if __name__ == "__main__":
    main()
```

Slip 11

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

```
def mean_end_analysis(start, goal):
    operations = []
    for i in range(len(start)):
        if start[i] != goal[i]:
            operations.append(f"Change character at position {i} to {goal[i]}")
    return operations
def main():
    start_string = input("Enter the start string: ")
    goal_string = input("Enter the goal string: ")
    operations = mean_end_analysis(start_string, goal_string)
    if not operations:
        print("No operations needed. The strings are already the same.")
    else:
        print("Operations to transform the string:")
        for operation in operations:
            print(operation)
if __name__ == "__main__":
    main()
```

Slip 14,24

Write a python program to sort the sentence in alphabetical order?

```
def sort_sentence(sentence):
    words = sentence.split()
    sorted_sentence = ' '.join(sorted(words))
    return sorted_sentence
def main():
    input_sentence = input("Enter a sentence: ")
    sorted_sentence = sort_sentence(input_sentence)
    print("Sorted sentence:", sorted_sentence)
if __name__ == "__main__":  main()
```

Slip 9,17,19

Write python program to solve 8 puzzle problem using A* algorithm

```
from queue import PriorityQueue
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.heuristic = self.calculate_heuristic()
    def __lt__(self, other):
        return (self.depth + self.heuristic) < (other.depth + other.heuristic)
    def __eq__(self, other):
        return self.state == other.state
    def calculate_heuristic(self):
        # Simple heuristic: count the number of misplaced tiles
        return sum(1 for i in range(3) for j in range(3) if self.state[i][j] != i * 3 + j + 1)
    def is_goal(self):
        return self.state == [[1, 2, 3], [4, 5, 6], [7, 8, 0]]
    def generate_successors(self):
        successors = []
        zero_row, zero_col = next((i, j) for i, row in enumerate(self.state) for j, val in enumerate(row) if
val == 0)
        for move in [(0, 1), (1, 0), (0, -1), (-1, 0)]:
            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.copy() for row in self.state]
                new_state[zero_row][zero_col], new_state[new_row][new_col] =
new_state[new_row][new_col], 0
                successors.append(PuzzleNode(new_state, self, move, self.depth + 1))
        return successors
def a_star(initial_state):
    initial_node = PuzzleNode(initial_state)
    frontier = PriorityQueue()
    explored = set()
    frontier.put(initial_node)
    while not frontier.empty():
        current_node = frontier.get()
        if current_node.is_goal():
            return current_node
        explored.add(tuple(map(tuple, current_node.state)))
        successors = current_node.generate_successors()
        for successor in successors:
            if tuple(map(tuple, successor.state)) not in explored:
                frontier.put(successor)
    return None
def print_solution(solution_node):
    path = []
    while solution_node:
        path.append((solution_node.state, solution_node.move))
        solution_node = solution_node.parent
```

```
    path.reverse()
    for state, move in path:
        print_state(state)
        if move:
            print(f"Move: {move}")
        print()
def print_state(state):
    for row in state:
        print(row)
    print()
def main():
    initial_state = [[2, 8, 3], [1, 6, 4], [7, 0, 5]]
    solution_node = a_star(initial_state)
    if solution_node:
        print("Solution found!")
        print_solution(solution_node)
    else:
        print("No solution found.")
if __name__ == "__main__":
    main()
```

Slip 9,11

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_x, capacity_y, target):
    jug_x = 0
    jug_y = 0
    while jug_x != target and jug_y != target:
        print("Jug X: {jug_x}L, Jug Y: {jug_y}L")
        if jug_x == 0:
            jug_x = capacity_x
            print("Fill Jug X")
        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("Jug X: {jug_x}L, Jug Y: {jug_y}L")
    print("Solution Found!")
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

Write a Python program to simulate 4-Queens problem.

```
def is_safe(board, row, col):
   # Check if there is a queen in the same column
   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 print_solution(board):
   for row in range(len(board)):
      line = ""
      for col in range(len(board)):
         line += "Q" if board[row] == col else "."
      print(line)    print()
def solve_queens(board, row):
   if row == len(board):
      print_solution(board)
      return
   for col in range(len(board)):
      if is_safe(board, row, col):
         board[row] = col
         solve_queens(board, row + 1)
         board[row] = -1  # Backtrack
def main():
   board_size = 4
   initial_board = [-1] * board_size  # -1 represents an empty cell
   solve_queens(initial_board, 0)
if __name__ == "__main__":
   main()
```

Slip 20,25

Build a bot which provides all the information related to you in college

```
def college_bot():
   college_info = {
      'name': 'Your Name',
      'major': 'Computer Science',
      'year': 'Senior',
      'interests': 'Programming, AI, Robotics',
      'clubs': 'Programming Club, Robotics Club',
      'projects': 'Chatbot project, AI-based recommendation system',    }
   print("College Information Bot:")
   print("You can ask about your name, major, year, interests, clubs, and projects.")
   while True:
      user_query = input("Ask me something (type 'exit' to end): ").lower()
      if user_query == 'exit':
         print("Exiting College Information Bot. Goodbye!")
         break
      response = college_info.get(user_query, "I don't have information about that.")
      print("Bot: ", response)
      print()        if __name__ == "__main__":      college_bot()
```

Slip 13

Write a Python program to simulate 8-Queens problem.

```
def is_safe(board, row, col):
    # Check if there is a queen in the same column or diagonals
    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 print_board(board):
    for row in board:
        line = ['Q' if col == 1 else '.' for col in row]
        print(' '.join(line))
    print()
def solve_queens(board, row):
    if row == len(board):
        # All queens are placed successfully
        print_board(board)
        return
    for col in range(len(board)):
        if is_safe(board, row, col):
            board[row] = col
            solve_queens(board, row + 1)
            board[row] = -1  # Backtrack
def main():
    board_size = 8
    initial_board = [-1] * board_size  # -1 represents an empty cell
    solve_queens(initial_board, 0)
if __name__ == "__main__":
    main()
```

Slip 14

Write a Python program to simulate n-Queens problem.

```
def is_safe(board, row, col):
    # Check if there is a queen in the same column or diagonals
    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 print_board(board):
    for row in board:
        line = ['Q' if col == 1 else '.' for col in row]
        print(' '.join(line))
    print()
def solve_n_queens(board, row, n):
    if row == n:
        # All queens are placed successfully
```

```
        print_board(board)
        return
    for col in range(n):
        if is_safe(board, row, col):
            board[row] = col
            solve_n_queens(board, row + 1, n)
            board[row] = -1  # Backtrack
def n_queens(n):
    initial_board = [-1] * n  # -1 represents an empty cell
    solve_n_queens(initial_board, 0, n)
def main():
    n = int(input("Enter the size of the chessboard (n): "))
    n_queens(n)
if __name__ == "__main__":
    main()
```

Slip 15

Write a Program to Implement Monkey Banana Problem using Python

```
import random
def initialize_grid(rows, cols):
    grid = [[' ' for _ in range(cols)] for _ in range(rows)]
    return grid
def place_objects(grid, monkey_row, monkey_col, banana_row, banana_col):
    grid[monkey_row][monkey_col] = 'M'
    grid[banana_row][banana_col] = 'B'
def print_grid(grid):
    for row in grid:
        print(' '.join(row))
    print()
def move_monkey(grid, direction, monkey_row, monkey_col):
    grid[monkey_row][monkey_col] = ' '
    if direction == 'up' and monkey_row > 0:
        monkey_row -= 1
    elif direction == 'down' and monkey_row < len(grid) - 1:
        monkey_row += 1
    elif direction == 'left' and monkey_col > 0:
        monkey_col -= 1
    elif direction == 'right' and monkey_col < len(grid[0]) - 1:
        monkey_col += 1
    grid[monkey_row][monkey_col] = 'M'
    return monkey_row, monkey_col
def is_banana_reached(monkey_row, monkey_col, banana_row, banana_col):
    return monkey_row == banana_row and monkey_col == banana_col
def main():
    rows = 5
    cols = 5
    monkey_row = random.randint(0, rows - 1)
    monkey_col = random.randint(0, cols - 1)
    banana_row = random.randint(0, rows - 1)
    banana_col = random.randint(0, cols - 1)
```

```
    while monkey_row == banana_row and monkey_col == banana_col:
        banana_row = random.randint(0, rows - 1)
        banana_col = random.randint(0, cols - 1)
    grid = initialize_grid(rows, cols)
    place_objects(grid, monkey_row, monkey_col, banana_row, banana_col)
    while not is_banana_reached(monkey_row, monkey_col, banana_row, banana_col):
        print_grid(grid)
        direction = input("Enter the direction (up/down/left/right): ")
        monkey_row, monkey_col = move_monkey(grid, direction, monkey_row, monkey_col)
    print("Congratulations! Monkey reached the banana.")
if __name__ == "__main__":
    main()
```

Slip 15

Write a program to implement Iterative Deepening DFS algorithm.

[Goal Node =G]

```
class Node:
    def __init__(self, value):
        self.value = value
        self.children = []
def iterative_deepening_dfs(root, goal):
    depth = 0
    while True:
        result = depth_limited_dfs(root, goal, depth)
        if result is not None:
            return result
        depth += 1
def depth_limited_dfs(node, goal, depth):
    if depth == 0 and node.value == goal:
        return [node.value]
    elif depth > 0:
        for child in node.children:
            result = depth_limited_dfs(child, goal, depth - 1)
            if result is not None:
                return [node.value] + result
    return None
def print_path(path):
    if path is not None:
        print("Path found:", " -> ".join(map(str, path)))
    else:
        print("Path not found.")
def main():
    # Example graph
    a = Node('A')
    b = Node('B')
    c = Node('C')
    d = Node('D')
    e = Node('E')
    f = Node('F')
    g = Node('G')
```

```
    a.children = [b, c]
    b.children = [d, e]
    c.children = [f, g]
    start_node = a
    goal_node = 'G'
    path = iterative_deepening_dfs(start_node, goal_node)
    print_path(path)
if __name__ == "__main__":
    main()
```

slip 18

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
class State:
    def __init__(self, width, height):
        self.width, self.height = width, height
        self.objects = [(3, 2), (2, 3), (2, 2), (3, 1), (1, 2)]
    def is_goal(self):
        return not self.objects
    def heuristic(self):
        return self.width * self.height - sum(w * h for w, h in self.objects)
def a_star(width, height):
    initial_state = State(width, height)
    open_set, closed_set = [initial_state], set()
    while open_set:
        current_state = heapq.heappop(open_set)
        if current_state.is_goal():
            return current_state.objects
        closed_set.add(tuple(current_state.objects))
        for obj_size in current_state.objects:
            new_objects = [obj for obj in current_state.objects if obj != obj_size]
            for rotation in range(2):
                if rotation == 1:
                    obj_size = (obj_size[1], obj_size[0])
                new_state = State(width, height)
                new_state.objects = new_objects
                if tuple(new_state.objects) not in closed_set:
                    heapq.heappush(open_set, new_state)
    return None
def main():
    result = a_star(10, 5)
    if result:
        print("Objects placement:", result)
    else:
        print("No solution found.")
if __name__ == "__main__":
    main()
```

Slip ,10,21,23,24
Write a Python program for the following Cryptarithmetic problems.
SEND + MORE = MONEY ,GO + TO = OUT, CROSS+ROADS = DANGER, TWO+TWO=FOUR

```
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 solutionfound.")
```

Slip 22
Q.1) Write a Program to Implement Alpha-Beta Pruning using Python

```
def alpha_beta_pruning(node, depth, alpha, beta, maximizing_player):
   if depth == 0 or is_terminal(node):
      return evaluate(node)
   if maximizing_player:
      max_eval = float('-inf')
      for child in get_children(node):
         eval_child = alpha_beta_pruning(child, depth - 1, alpha, beta, False)
         max_eval = max(max_eval, eval_child)
         alpha = max(alpha, eval_child)
         if beta <= alpha:
            break
      return max_eval
```

```
    else:
        min_eval = float('inf')
        for child in get_children(node):
            eval_child = alpha_beta_pruning(child, depth - 1, alpha, beta, True)
            min_eval = min(min_eval, eval_child)
            beta = min(beta, eval_child)
            if beta <= alpha:
                break
        return min_eval
def is_terminal(node):
    return True
def evaluate(node):
    return 0
def get_children(node):
    return []
def main():
    root_node = "Root"
    max_depth = 3
    alpha = float('-inf')
    beta = float('inf')
    result = alpha_beta_pruning(root_node, max_depth, alpha, beta, True)
    print("Optimal value:", result)
if __name__ == "__main__":
    main()
```

Slip 25

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

```
from collections import deque
def print_puzzle(puzzle):
    for row in puzzle:
        print(row)
    print()
def is_goal_state(puzzle):
    goal_state = [[1, 2, 3], [8, 0, 4], [7, 6, 5]]
    return puzzle == goal_state
def get_blank_position(puzzle):
    for i in range(3):
        for j in range(3):
            if puzzle[i][j] == 0:
                return i, j
def get_neighbors(puzzle):
    i, j = get_blank_position(puzzle)
    moves = [(0, 1), (1, 0), (0, -1), (-1, 0)]
    neighbors = []
    for di, dj in moves:
        ni, nj = i + di, j + dj
        if 0 <= ni < 3 and 0 <= nj < 3:
            new_puzzle = [row.copy() for row in puzzle]
            new_puzzle[i][j], new_puzzle[ni][nj] = new_puzzle[ni][nj], new_puzzle[i][j]
            neighbors.append(new_puzzle)
```

```
    return neighbors
def solve_8_puzzle(initial_state):
    queue = deque([(initial_state, [])])
    while queue:
        current_state, path = queue.popleft()
        if is_goal_state(current_state):
            return path
        for neighbor in get_neighbors(current_state):
            if neighbor not in path:
                queue.append((neighbor, path + [neighbor]))
    return None
def main():
    initial_state = [
        [1, 2, 3],
        [0, 8, 4],
        [7, 6, 5]
    ]
    print("Initial Puzzle:")
    print_puzzle(initial_state)
    solution_path = solve_8_puzzle(initial_state)
    if solution_path:
        print("Solution Path:")
        for step, state in enumerate(solution_path):
            print(f"Step {step + 1}:")
            print_puzzle(state)
    else:
        print("No solution found.")
if __name__ == "__main__":
    main()
```

Slip 13,20

Write a Python program to implement Mini-Max Algorithm.

```
def print_board(board):
    for row in board:
        print(" | ".join(row))
        print("-" * 9)
def is_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 evaluate(board):
    if is_winner(board, "O"):
        return 1
    elif is_winner(board, "X"):
        return -1
    elif is_board_full(board):
```

```
        return 0
    else:
        return None  # Game is not over
def minimax(board, depth, maximizing_player):
    result = evaluate(board)
    if result is not None:
        return result
    if maximizing_player:
        max_eval = float('-inf')
        for i in range(3):
            for j in range(3):
                if board[i][j] == " ":
                    board[i][j] = "O"
                    eval_child = minimax(board, depth + 1, False)
                    max_eval = max(max_eval, eval_child)
                    board[i][j] = " "
        return max_eval
    else:
        min_eval = float('inf')
        for i in range(3):
            for j in range(3):
                if board[i][j] == " ":
                    board[i][j] = "X"
                    eval_child = minimax(board, depth + 1, True)
                    min_eval = min(min_eval, eval_child)
                    board[i][j] = " "
        return min_eval
def get_best_move(board):
    best_val = float('-inf')
    best_move = None
    for i in range(3):
        for j in range(3):
            if board[i][j] == " ":
                board[i][j] = "O"
                move_val = minimax(board, 0, False)
                board[i][j] = " "
                if move_val > best_val:
                    best_val = move_val
                    best_move = (i, j)
    return best_move
def main():
    board = [[" " for _ in range(3)] for _ in range(3)]
    print("Tic-Tac-Toe Game:")
    print_board(board)
    while True:
        print("Player X's turn:")
        row, col = map(int, input("Enter your move (row and column): ").split())
        if 1 <= row <= 3 and 1 <= col <= 3 and board[row - 1][col - 1] == " ":
            board[row - 1][col - 1] = "X"
            print_board(board)
```

```
            if is_winner(board, "X"):
                print("Player X wins!")
                break
            elif is_board_full(board):
                print("It's a draw!")
                break
            print("Player O's turn:")
            best_move = get_best_move(board)
            board[best_move[0]][best_move[1]] = "O"
            print_board(board)
            if is_winner(board, "O"):
                print("Player O wins!")
                break
            elif is_board_full(board):
                print("It's a draw!")
                break
        else:
            print("Invalid move. Try again.")
if __name__ == "__main__":
    main()
```