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)

```
import numpy as np

def objective_function(x):
    return -x**2 + 4*x

def hill_climbing(initial_x, step_size, iterations):
    current_x = initial_x

    for _ in range(iterations):
        current_value = objective_function(current_x)
        next_x = current_x + step_size
        next_value = objective_function(next_x)

        if next_value > current_value:
            current_x = next_x

    return current_x, objective_function(current_x)

# Example usage
initial_x = 0.0
step_size = 0.1
iterations = 50

max_x, max_value = hill_climbing(initial_x, step_size, iterations)

print(f"Maximum value found at x = {max_x}, f(x) = {max_value}")
```

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]

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

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

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

    if start == goal:
```

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

if __name__ == "__main__":
    initial_node = 1
    goal_node = 8

    dfs(graph, initial_node, goal_node)
```

Slip 2 :
Q1 : Write a python program to generate Calendar for the given month and year?.

```
import calendar

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

    print(f"Calendar for {calendar.month_name[month]} {year}:\n")
    print("Mo Tu We Th Fr Sa Su")

    for week in cal:
        week_str = " ".join(str(day) if day != 0 else "  " for day in week)
        print(week_str)

year = 2023
month = 4  # April

generate_calendar(year, month)
```

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=7].

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

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

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

    if start == goal:
        print("DFS Path:", path)
    else:
```

```
        for neighbor in graph.get(start, []):
            if neighbor not in visited:
                dfs(graph, neighbor, goal, visited, path)

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

    dfs(graph, initial_node, goal_node)
```

Slip 3 :
Q1 : Write a python program to remove punctuations from the given string?

```
import string

def remove_punctuation(input_string):
        translator = str.maketrans("", "", string.punctuation)

        cleaned_string = input_string.translate(translator)

    return cleaned_string

input_string = "Hello, World! This is an example string."

cleaned_string = remove_punctuation(input_string)
print(f"Original String: {input_string}")
print(f"String without Punctuation: {cleaned_string}")
```

Q2 : 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]

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

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

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

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

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

    dfs(graph, initial_node, goal_node)
```

Slip 04 :
Q1 : 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

```python
import random

def choose_word():
    # List of words for the game
    words = ["python", "hangman", "programming", "challenge", "computer", "science"]

    # Choose a random word from the list
    return random.choice(words)

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

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

    # Choose a random word
    secret_word = choose_word()

    # Initialize variables
    guessed_letters = []
    attempts = 6

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

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

        # Check if the guessed letter is in the word
        if guess.isalpha() and len(guess) == 1:
            if guess in guessed_letters:
                print("You already guessed that letter. Try again.")
            elif guess in secret_word:
```

```
            print("Good guess!")
            guessed_letters.append(guess)
        else:
            print("Incorrect guess. Try again.")
            attempts -= 1
    else:
        print("Invalid input. Please enter a single alphabet.")

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

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

# Run the Hangman game
hangman()
```

Q2 :
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]

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

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

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

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

if __name__ == "__main__":
    initial_node = 1
    goal_node = 8

    dfs(graph, initial_node, goal_node)
```

Slip 5:

1) 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(text):
    words = word_tokenize(text)


    lemmatizer = WordNetLemmatizer()

    lemmatized_words = [lemmatizer.lemmatize(word) for word in words]

    lemmatized_text = ' '.join(lemmatized_words)

    return lemmatized_text

input_text = "The cats are running and the mice are hiding."

lemmatized_result = lemmatize_text(input_text)
print(f"Original Text: {input_text}")
print(f"Lemmatized Text: {lemmatized_result}")
```

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]

```
from collections import deque

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

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

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

        if node not in visited:
            neighbors = graph.get(node, [])
            for neighbor in neighbors:
                new_path = list(path)
                new_path.append(neighbor)
                queue.append(new_path)

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

```
        visited.add(node)

if __name__ == "__main__":
    initial_node = 1
    goal_node = 8

    bfs(graph, initial_node, goal_node)
```

Slip : 06 :

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

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

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

def remove_stop_words(input_text):
    # Tokenize the text into words
    words = word_tokenize(input_text)

    # Get the English stop words
    stop_words = set(stopwords.words('english'))

    # Remove stop words from the list of words
    filtered_words = [word for word in words if word.lower() not in stop_words]

    # Join the filtered words into a string
    filtered_text = ' '.join(filtered_words)

    return filtered_text

# Example usage
file_path = 'sample_text.txt'  # Replace with the path to your text file

try:
    with open(file_path, 'r', encoding='utf-8') as file:
        passage = file.read()
        print(f"Original Passage:\n{passage}")

        filtered_passage = remove_stop_words(passage)
        print(f"\nPassage after removing stop words:\n{filtered_passage}")

except FileNotFoundError:
    print(f"Error: File '{file_path}' not found.")
except Exception as e:
    print(f"Error: {e}")
```

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].

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

```
    5: [6, 7],
    6: [7],
    4: [7],
    3: [4]
}

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

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

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

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

    dfs(graph, initial_node, goal_node)
```

Slip : 07

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

```
import math

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

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

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

def game_over(board):
    # Check if the game is over
    return is_winner(board, 'X') or is_winner(board, 'O') or is_full(board)
```

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

def minimax(board, depth, maximizing_player, alpha, beta):
  if game_over(board):
    return evaluate(board)

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

def find_best_move(board):
  best_val = -math.inf
  best_move = (-1, -1)

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

  return best_move

def play_tic_tac_toe():
```

```
    board = [[" " for _ in range(3)] for _ in range(3)]
    current_player = 'X'

    while not game_over(board):
        print_board(board)

        if current_player == 'X':
            row = int(input("Enter row (0, 1, or 2): "))
            col = int(input("Enter column (0, 1, or 2): "))
        else:
            row, col = find_best_move(board)

        if 0 <= row < 3 and 0 <= col < 3 and board[row][col] == " ":
            board[row][col] = current_player
            current_player = 'O' if current_player == 'X' else 'X'
        else:
            print("Invalid move. Try again.")

    print_board(board)
    winner = evaluate(board)
    if winner == -1:
        print("Player X wins!")
    elif winner == 1:
        print("Player O wins!")
    else:
        print("It's a draw!")

# Start the game
play_tic_tac_toe()
```

---

2) Write a Python program to implement Simple Chatbot.

```
import random

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

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

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

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

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

Slip 8 :

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

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

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

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

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

---

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

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

def check_winner(board, player):
    # Check rows, columns, and diagonals for a win
    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(board[i][j] != ' ' for i in range(3) for j in range(3))

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

    while True:
        print_board(board)

        row = int(input(f"Player {current_player}, enter row (0, 1, or 2): "))
        col = int(input(f"Player {current_player}, enter column (0, 1, or 2): "))

        if board[row][col] == ' ':
            board[row][col] = current_player
```

```
            if check_winner(board, current_player):
                print_board(board)
                print(f"Player {current_player} wins!")
                break
            elif is_board_full(board):
                print_board(board)
                print("It's a draw!")
                break

            # Switch player
            current_player = 'O' if current_player == 'X' else 'X'
        else:
            print("Cell already occupied. Try again.")

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

Slip 9 :

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

```
import heapq
import itertools

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

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

    def calculate_cost(self):
        return self.depth + self.heuristic()

    def heuristic(self):
        # Manhattan distance heuristic
        distance = 0
        for i in range(3):
            for j in range(3):
                value = self.state[i][j]
                if value != 0:
                    goal_row, goal_col = divmod(value - 1, 3)
                    distance += abs(i - goal_row) + abs(j - goal_col)
        return distance

    def get_neighbors(self):
        neighbors = []
        zero_row, zero_col = self.find_zero()
        moves = [(0, 1), (0, -1), (1, 0), (-1, 0)]

        for move in moves:
            new_row, new_col = zero_row + move[0], zero_col + move[1]
            if 0 <= new_row < 3 and 0 <= new_col < 3:
                new_state = [list(row) for row in self.state]
```

```python
            new_state[zero_row][zero_col], new_state[new_row][new_col] = \
                new_state[new_row][new_col], new_state[zero_row][zero_col]
            neighbors.append(PuzzleNode(new_state, self, move, self.depth + 1))

        return neighbors

    def find_zero(self):
        for i in range(3):
            for j in range(3):
                if self.state[i][j] == 0:
                    return i, j

    def print_path(self):
        if self.parent is not None:
            self.parent.print_path()
            print(f"Move {self.move}:\n{self}")
        else:
            print("Initial State:")
            print(self)

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


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

    if not is_solvable(initial_state):
        print("The given puzzle is not solvable.")
        return

    heap = [initial_node]
    heapq.heapify(heap)
    visited = set()

    while heap:
        current_node = heapq.heappop(heap)

        if current_node.state == goal_state:
            print("Solution found!")
            current_node.print_path()
            return

        visited.add(tuple(map(tuple, current_node.state)))

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

    print("No solution found.")


def is_solvable(puzzle):
    inversions = 0
    flatten_puzzle = list(itertools.chain.from_iterable(puzzle))
```

```
    for i in range(8):
        for j in range(i + 1, 9):
            if flatten_puzzle[i] > flatten_puzzle[j] and flatten_puzzle[i] != 0 and flatten_puzzle[j] != 0:
                inversions += 1

    return inversions % 2 == 0


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

---

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

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

    while jug2 != target:
        print(f"Jug 1: {jug1} gallons, Jug 2: {jug2} gallons")

        # Fill jug 2 if it's empty
        if jug2 == 0:
            jug2 = capacity_jug2
        # Pour water from jug 2 to jug 1
        elif jug1 + jug2 <= capacity_jug1:
            jug1 += jug2
            jug2 = 0
        # Pour water from jug 2 to jug 1 until jug 1 is full
        else:
            jug2 -= capacity_jug1 - jug1
            jug1 = capacity_jug1

    print(f"Jug 1: {jug1} gallons, Jug 2: {jug2} gallons\nTarget of {target} gallons achieved!")

if __name__ == "__main__":
    water_jug_problem(5, 7, 4)
```

Slip 10 :

1) Write Python program to implement crypt arithmetic problem TWO+TWO=FOUR

```
from itertools import permutations

def solve_cryptarithmetic():
    for perm in permutations("0123456789", 6):
        # Assign digits to letters
        mapping = {'T': perm[0], 'W': perm[1], 'O': perm[2], 'F': perm[3], 'U': perm[4], 'R': perm[5]}

        # Check if leading digits are not zero
        if mapping['T'] == '0' or mapping['W'] == '0' or mapping['F'] == '0':
            continue

        # Convert letters to integers
        T = int(mapping['T'])
        W = int(mapping['W'])
        O = int(mapping['O'])
```

```
    F = int(mapping['F'])
    U = int(mapping['U'])
    R = int(mapping['R'])

    # Check if the equation is satisfied
    if T*100 + W*10 + O + T*100 + W*10 + O == F*1000 + O*100 + U*10 + R:
        print(f"T = {T}, W = {W}, O = {O}, F = {F}, U = {U}, R = {R}")
        print(f"{T}{W}{O}")
        print(f"+{T}{W}{O}")
        print(f"--------")
        print(f"{F}{O}{U}{R}")

# Solve the cryptarithmetic problem
solve_cryptarithmetic()
```

2) Write a Python program to implement Simple Chatbot :

```
import random

def simple_chatbot():
    responses = {
        "hello": ["Hi there!", "Hello!", "Hey!"],
        "how are you": ["I'm good, thanks!", "I'm doing well.", "All good!"],
        "what's your name": ["I'm just a simple chatbot.", "I don't have a name.", "Call me ChatBot."],
        "bye": ["Goodbye!", "See you later!", "Bye!"],
        "default": ["I'm not sure how to respond.", "Could you say that again?", "Sorry, I didn't get that."]
    }

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

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

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

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