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

```
import random
import math
def f(x):
    return -x*x + 4*x
def hill_climb(func, start, step=0.1, max_iters=1000, lower=-100, upper=100):
    x = start
    fx = func(x)
    for i in range(max_iters):
        # consider neighbors left and right
        left = max(lower, x - step)
        right = min(upper, x + step)
        f_left = func(left)
        f_right = func(right)
        if f_left > fx and f_left >= f_right:
            x, fx = left, f_left
        elif f_right > fx and f_right >= f_left:
            x, fx = right, f_right
        else:
            break
    return x, fx
def hill_climbing_with_restarts(func, domain=(-10, 10), step=0.01, restarts=10):
    best_x, best_fx = None, -float('inf')
    low, high = domain
    for r in range(restarts):
        start = random.uniform(low, high)
        x, fx = hill_climb(func, start, step=step, max_iters=10000, lower=low, upper=high)
        if fx > best_fx:
            best_x, best_fx = x, fx
    return best_x, best_fx
if __name__ == "__main__":
    x0 = 0.0
    x_found, f_found = hill_climb(f, start=x0, step=0.01, max_iters=10000, lower=-10, upper=10)
    print("Single-run hill-climb:")
    print(f"  start={x0} -> x* = {x_found:.6f}, f(x*) = {f_found:.6f}")
    xr, fr = hill_climbing_with_restarts(f, domain=(-10, 10), step=0.001, restarts=20)
    print("With random restarts:")
    print(f"  x* = {xr:.6f}, f(x*) = {fr:.6f}")
    print("Analytical maximum: x = 2, f(2) = 4")
```

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

```
def dfs_recursive(graph, start, goal, visited=None, path=None):
    if visited is None:   visited = set()
    if path is None:     path = []
    visited.add(start)
    path.append(start)
    if start == goal:       return path[:]
    for neighbor in graph.get(start, []):
        if neighbor not in visited:
            res = dfs_recursive(graph, neighbor, goal, visited, path)
            if res is not None:         return res
    path.pop()    return None
def dfs_iterative(graph, start, goal):
    stack = [(start, 0, [start])]
    visited = set()
    while stack:
        node, idx, path = stack.pop()
        if node == goal:   return path
        if node not in visited:
            visited.add(node)
            neighbors = graph.get(node, [])
            for nb in reversed(neighbors):
                if nb not in visited:
                    stack.append((nb, 0, path + [nb]))    return None
if __name__ == "__main__":
    graph = {1: [2, 3],2: [4, 5], 3: [6, 7],4: [],5: [8],
        6: [],7: [8],8: []    }
   start_node = 1
    goal_node = 8
    print("Graph adjacency list:")
    for k in sorted(graph.keys()):
        print(f"  {k}: {graph[k]}")
    print("\nDFS (recursive) from 1 to 8:")
    path_rec = dfs_recursive(graph, start_node, goal_node)
    if path_rec:
        print("  Path found:", " -> ".join(map(str, path_rec)))
    else:        print("  No path found.")
    print("\nDFS (iterative) from 1 to 8:")
    path_it = dfs_iterative(graph, start_node, goal_node)
    if path_it:
        print("  Path found:", " -> ".join(map(str, path_it)))
    else:
        print("  No path found.")
```

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

```
import calendar
def print_month_calendar(year: int, month: int):
    print(calendar.month(year, month))
if __name__ == "__main__":
    y = int(input("Year (e.g. 2025): "))
    m = int(input("Month (1-12): "))
    print_month_calendar(y, m)
```

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].

```
from collections import defaultdict, deque
class Graph:
    def __init__(self):
        self.adj = defaultdict(list)
    def add_edge(self, u, v):
        self.adj[u].append(v)
def dfs_recursive(graph, start, goal, visited=None, path=None):
    if visited is None: visited = set()
    if path is None: path = []
    visited.add(start)
    path.append(start)
    if start == goal:        return path.copy()
    for nbr in graph.adj[start]:
        if nbr not in visited:
            res = dfs_recursive(graph, nbr, goal, visited, path)
            if res:          return res
    path.pop()    return None
def dfs_iterative(graph, start, goal):
    visited = set()    stack = [(start, [start])]
    while stack:        node, path = stack.pop()
        if node == goal:         return path
        if node not in visited:            visited.add(node)
            for nbr in reversed(graph.adj[node]):
                if nbr not in visited:
                    stack.append((nbr, path + [nbr]))    return None
if __name__ == "__main__":
    g = Graph()    edges = [(1,2),(1,3),(2,4),(2,5),(3,6),(5,7),(6,7)]
    for u,v in edges: g.add_edge(u,v)
    start, goal = map(int, input("Start Goal: ").split())
    print("Recursive DFS path:", dfs_recursive(g, start, goal))
    print("Iterative DFS path:", dfs_iterative(g, start, goal))
```

Slip 2

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

```
import string
def remove_punct(s: str) -> str:
    return s.translate(str.maketrans('', '', string.punctuation))
if __name__ == "__main__":
    s = input("Enter text: ")
    print(remove_punct(s))
```

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]

```
import random
def hangman(words=None, max_wrong=6):
    if words is None:
        words = ["python","computer","hangman","programming","algorithm"]
    word = random.choice(words).lower()
    guessed = set()
    wrong = 0
    display = ["_" if c.isalpha() else c for c in word]
    while wrong < max_wrong and "_" in display:
        print("\nWord:"," ".join(display))
        print(f"Wrong guesses: {wrong}/{max_wrong}", "Guessed:", " ".join(sorted(guessed)))
        ch = input("Enter a letter: ").lower().strip()
        if not ch or not ch.isalpha() or len(ch)!=1:
            print("Enter single alphabet.")
            continue
        if ch in guessed:
            print("Already guessed.")
            continue
        guessed.add(ch)
        if ch in word:
            for i,c in enumerate(word):
                if c==ch: display[i]=ch
        else:
            wrong += 1
    if "_" not in display:
        print("\nYou won! Word:", word)
    else:
        print("\nYou lost! Word:", word)
if __name__ == "__main__":
    hangman()
```

Slip 3

Q.1) Write a program to implement Hangman game using python. [10 Marks] 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 hangman():
    words = ["python", "computer", "program", "hangman", "science"]
    word = random.choice(words)     # choose a random word
    word = word.lower()
    guessed = []
    attempts = 6
    print("====== Hangman Game ======")
    print("Guess the word!")
    display = ["_" for _ in word]
    while attempts > 0 and "_" in display:
        print("\nWord: ", " ".join(display))
        print("Attempts left:", attempts)
        print("Guessed letters:", guessed)
        ch = input("Enter a single alphabet: ").lower()
        if len(ch) != 1 or not ch.isalpha():
            print("Invalid input! Enter only one alphabet.")
            continue
        if ch in guessed:
            print("You already guessed that letter!")
            continue
        guessed.append(ch)
        if ch in word:
            print("Correct guess!")
            for i in range(len(word)):
                if word[i] == ch:
                    display[i] = ch
        else:
            print("Wrong guess!")
            attempts -= 1
    if "_" not in display:
        print("\nCongratulations! You guessed the word:", word)
    else:
        print("\nGame Over! You failed to guess the word.")
        print("The correct word was:", word)
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]

```
from collections import deque
def bfs(graph, start, goal):
    visited = set()
    queue = deque([start])
   print("BFS Traversal Order:")
    while queue:
        node = queue.popleft()
        if node not in visited:
            print(node, end=" ")
            visited.add(node)
            if node == goal:
                print("\nGoal Node", goal, "found!"          return
            for neighbour in graph.get(node, []):
                if neighbour not in visited:
                    queue.append(neighbour)
    print("\nGoal Node not found in the graph.")
graph = {1: [2, 3],2: [4, 5],3: [6],4: [],
    5: [7],6: [8],7: [],8: []   }
bfs(graph, 1, 8)
```

Slip 4

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

```
def means_end_analysis(start, goal):
    print("Start String :", start)    print("Goal String  :", goal)
    print("\nSteps:\n")
    i = 0           steps = 1
    start = list(start)
    while i < len(start) and i < len(goal):
        if start[i] != goal[i]:
            print(f"Step {steps}: Substitute '{start[i]}' with '{goal[i]}' at position {i}")
            start[i] = goal[i]
            steps += 1          i += 1
    while len(start) < len(goal):
        print(f"Step {steps}: Insert '{goal[i]}' at position {i}")
        start.append(goal[i])
        i += 1          steps += 1
    while len(start) > len(goal):
        print(f"Step {steps}: Delete '{start[-1]}' from end")
        start.pop()       steps += 1
    print("\nFinal String:", ''.join(start))
start_str = "bat"       goal_str = "boat"
means_end_analysis(start_str, goal_str)
```

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]

```
from collections import deque
graph = {
    1: [2, 3],
    2: [4, 5],
    3: [6],
    4: [],
    5: [7],
    6: [8],
    7: [],
    8: []
}
def bfs(graph, start, goal):
    visited = set()          # to track visited nodes
    queue = deque([[start]])  # queue stores paths instead of just nodes
    while queue:
        path = queue.popleft()      # remove first path
        node = path[-1]             # last node of the path
        if node == goal:
            return path   # Goal found → return the path
        if node not in visited:
            visited.add(node)

            for neighbour in graph[node]:
                new_path = list(path)
                new_path.append(neighbour)
                queue.append(new_path)
    return None
start_node = 1
goal_node = 8
result = bfs(graph, start_node, goal_node)

if result:
    print("BFS Traversal Path from", start_node, "to", goal_node, "is:", result)
else:
    print("Goal not reachable from the start node.")
```

Slip 5

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

```
import nltk
from nltk.corpus import stopwords
from nltk.tokenize import word_tokenize
nltk.download('punkt')
nltk.download('stopwords')
file_path = "input.txt"
with open(file_path, "r") as file:
    text = file.read()
words = word_tokenize(text)
stop_words = set(stopwords.words('english'))
filtered_words = [word for word in words if word.lower() not in stop_words]
print("Original Text:")
print(text)
print("\nText After Removing Stop Words:")
print(" ".join(filtered_words))
```

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].

```
from collections import deque
graph = {1: [2, 3],2: [4, 5],3: [6],4: [],5: [7],6: [8],7: [],8: []}
def bfs(graph, start, goal):
    visited = set()
    queue = deque([[start]])
    while queue:
        path = queue.popleft()
        node = path[-1]
        if node == goal:
            return path
if node not in visited:
            visited.add(node)
            for neighbour in graph[node]:
                new_path = list(path)
                new_path.append(neighbour)
                queue.append(new_path)
    return None
start_node = 1
goal_node = 8
result = bfs(graph, start_node, goal_node)
print("Breadth First Search from", start_node, "to", goal_node)
if result:
    print("Goal Found! Path =", result)
else: print("Goal NOT reachable from the start node.")
```

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

```
import math
def print_board(board):
    print()
    for i in range(3):
        row = board[3*i:3*i+3]
        print(" " + " | ".join(row))
        if i < 2:
            print("---+---+---")
    print()

def available_moves(board):
    return [i for i, v in enumerate(board) if v == ' ']

def is_full(board):
    return all(v != ' ' for v in board)

def check_winner(board):
    # Returns 'X' or 'O' if somebody won, or None if no winner yet
    wins = [(0,1,2),(3,4,5),(6,7,8),
            (0,3,6),(1,4,7),(2,5,8),
            (0,4,8),(2,4,6)]
    for a,b,c in wins:
        if board[a] == board[b] == board[c] != ' ':
            return board[a]
    return None
def score_for_winner(winner, depth):
    # Prefer faster wins -> include depth: bigger positive if win earlier for maximizing.
    if winner == ai_player:
        return 10 - depth
    elif winner == human_player:
        return depth - 10
    else:
        return 0
def minimax_ab(board, depth, alpha, beta, maximizing):
    winner = check_winner(board)
    if winner or is_full(board):
        return score_for_winner(winner, depth), None
    if maximizing:
        best_value = -math.inf
        best_move = None
        for move in available_moves(board):
            board[move] = ai_player
```

```
            val, _ = minimax_ab(board, depth+1, alpha, beta, False)
            board[move] = ' '
            if val > best_value:
                best_value = val
                best_move = move
            alpha = max(alpha, best_value)
            if beta <= alpha:
                break  # beta cut-off
        return best_value, best_move
    else:
        best_value = math.inf
        best_move = None
        for move in available_moves(board):
            board[move] = human_player
            val, _ = minimax_ab(board, depth+1, alpha, beta, True)
            board[move] = ' '
            if val < best_value:
                best_value = val
                best_move = move
            beta = min(beta, best_value)
            if beta <= alpha:
                break  # alpha cut-off
        return best_value, best_move
def human_turn(board):
    while True:
        try:
            user = input("Enter your move (1-9) or 'q' to quit: ").strip()
            if user.lower() == 'q':
                print("Quitting game.")
                exit(0)
            pos = int(user) - 1
            if pos < 0 or pos > 8:
                print("Invalid position. Choose 1..9.")
            elif board[pos] != ' ':
                print("Cell already taken. Choose another.")
            else:
                board[pos] = human_player
                return
        except ValueError:
            print("Please enter a number 1..9 or 'q'.")
def ai_turn(board):
    print("AI is thinking...")
    _, move = minimax_ab(board, depth=0, alpha=-math.inf, beta=math.inf, maximizing=True)
    if move is None:
```

```
        moves = available_moves(board)
        move = moves[0] if moves else None
    board[move] = ai_player
    print(f"AI plays at position {move+1}.")
def play_game():
    board = [' '] * 9
    print("Tic-Tac-Toe (positions are 1..9):")
    print(" 1 | 2 | 3")
    print("---+---+---")
    print(" 4 | 5 | 6")
    print("---+---+---")
    print(" 7 | 8 | 9")    print()
    print(f"You are '{human_player}'. AI is '{ai_player}'.")
    current = starting_player  # 'human' or 'ai'
    while True:
        print_board(board)
        winner = check_winner(board)
        if winner:
            if winner == human_player:
                print("Congratulations — you win!")
            else:
                print("AI wins. Better luck next time.")
            break
        if is_full(board):
            print("It's a draw!")
            break
        if current == 'human':
            human_turn(board)
            current = 'ai'
        else:
            ai_turn(board)
            current = 'human'
if __name__ == "__main__":
    human_player = 'X'
    ai_player = 'O'
    starting_player = 'human'
    try: choice = input("Play first? (y/n, default y): ").strip().lower()
        if choice == 'n':
            starting_player = 'ai'
        elif choice == 'y':
            starting_player = 'human'
    except Exception:   pass

    play_game()
```

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

```
# Simple Chatbot Program

print("Simple Chatbot")
print("Type 'bye' to exit.")
print("---------------------------------")

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

    if user == "bye":
        print("Chatbot: Goodbye! Have a great day!")
        break

    elif "hello" in user or "hi" in user:
        print("Chatbot: Hello! How can I help you?")

    elif "how are you" in user:
        print("Chatbot: I am fine! Thanks for asking.")

    elif "name" in user:
        print("Chatbot: I am a simple Python Chatbot.")

    elif "time" in user:
        from datetime import datetime
        now = datetime.now().strftime("%H:%M:%S")
        print("Chatbot: Current time is", now)

    elif "thank" in user:
        print("Chatbot: You’re welcome!")

    else:
        print("Chatbot: Sorry, I don't understand that.")
```

Slip 7

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

```
text = input("Enter a string: ")
upper_count = 0
lower_count = 0
for ch in text:
    if ch.isupper():
        upper_count += 1
    elif ch.islower():
        lower_count += 1
print("Number of uppercase letters:", upper_count)
print("Number of lowercase letters:", lower_count)
```

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

```
board = [' ' for _ in range(9)]
def print_board():    print()
    print(board[0], "|", board[1], "|", board[2])
    print("--+---+--")
    print(board[3], "|", board[4], "|", board[5])
    print("--+---+--")
    print(board[6], "|", board[7], "|", board[8])    print()
def check_winner(player):
    win_conditions = [   (0,1,2), (3,4,5), (6,7,8),
        (0,3,6), (1,4,7), (2,5,8),  (0,4,8), (2,4,6) ]
    for a,b,c in win_conditions:
        if board[a] == board[b] == board[c] == player:
            return True
    return False
def is_full():
    return ' ' not in board
current_player = 'X'
while True:
    print_board()
    move = int(input(f"Player {current_player}, enter position (1-9): ")) - 1
    if board[move] != ' ':
        print("Invalid move! Try again.")
        continue
    board[move] = current_player
    if check_winner(current_player):
        print_board()
        print(f"Player {current_player} wins!")
        break
    if is_full():print_board()
        print("It's a draw!")          break
    current_player = 'O' if current_player == 'X' else 'X'
```

Slip 8

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

```
from heapq import heappush, heappop
goal = "123456780"
def heuristic(state):    distance = 0
   for i, c in enumerate(state):
      if c != '0':
         x1, y1 = i // 3, i % 3
         x2, y2 = (int(c) - 1) // 3, (int(c) - 1) % 3
         distance += abs(x1 - x2) + abs(y1 - y2)
   return distance
def get_neighbors(state):
   neighbors = []
   idx = state.index('0')
   x, y = idx // 3, idx % 3
   moves = [(-1,0), (1,0), (0,-1), (0,1)]
   for dx, dy in moves:
      nx, ny = x + dx, y + dy
      if 0 <= nx < 3 and 0 <= ny < 3:
         new_idx = nx * 3 + ny
         new_state = list(state)
         new_state[idx], new_state[new_idx] = new_state[new_idx], new_state[idx]
         neighbors.append(''.join(new_state))
   return neighbors
def a_star(start):      pq = []
   heappush(pq, (heuristic(start), start))
   visited = set()    parent = {}
   while pq:
      _, state = heappop(pq)
  if state == goal:
         path = []
         while state:      path.append(state)
            state = parent.get(state)
         return path[::-1]
      if state in visited:
         Continue         visited.add(state)
      for next_state in get_neighbors(state):
         if next_state not in visited:
            heappush(pq, (heuristic(next_state), next_state))
            parent[next_state] = state
start = input("Enter 8-puzzle start state (e.g., 125340678): ")
path = a_star(start)   print("\nSolution Path:")
for p in path:    print(p[0:3])    print(p[3:6])    print(p[6:9])    print()
```

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.

```
# Water Jug Problem (5 gallon and 7 gallon) Target = 4 in second jug

from collections import deque

def water_jug():
    visited = set()
    queue = deque()

    # (jug5, jug7)
    queue.append((0, 0))

    while queue:
        x, y = queue.popleft()

        if (x, y) in visited:
            continue

        visited.add((x, y))
        print("State:", (x, y))

        if y == 4:
            print("Goal Achieved! 4 gallons in 7-gallon jug.")
            return

        # Possible operations
        next_states = [
            (5, y),      # Fill 5-gallon
            (x, 7),      # Fill 7-gallon
            (0, y),      # Empty 5-gallon
            (x, 0),      # Empty 7-gallon
        ]
        pour = min(x, 7 - y)
        next_states.append((x - pour, y + pour))
        pour = min(y, 5 - x)
        next_states.append((x + pour, y - pour))
        for state in next_states:
            if state not in visited:
                queue.append(state)

water_jug()
```

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

```
import itertools
letters = ('T', 'W', 'O', 'F', 'U', 'R')
for digits in itertools.permutations(range(10), 6):
    mapping = dict(zip(letters, digits))
    if mapping['T'] == 0 or mapping['F'] == 0:
        continue
    TWO = 100*mapping['T'] + 10*mapping['W'] + mapping['O']
    FOUR = (1000*mapping['F'] + 100*mapping['O'] +
            10*mapping['U'] + mapping['R'])
    if TWO + TWO == FOUR:
        print("Solution Found!")
        print(mapping)
        print("TWO =", TWO)
        print("FOUR =", FOUR)
        break
else:         print("No solution.")
```

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

```
print("Simple Chatbot")
print("Type 'bye' to exit.")
print("--------------------------------")
while True:
    user = input("You: ").lower()
    if user == "bye":
        print("Chatbot: Goodbye!")
        break
    elif "hello" in user or "hi" in user:
        print("Chatbot: Hello! How can I help you?")

    elif "how are you" in user:
        print("Chatbot: I am fine. Hope you are doing well!")

    elif "name" in user:
        print("Chatbot: I am a simple Python chatbot.")

    elif "time" in user:
        from datetime import datetime
        print("Chatbot:", datetime.now().strftime("%H:%M:%S"))

    elif "thank" in user:
        print("Chatbot: You're welcome!")
    else:
        print("Chatbot: Sorry, I didn't understand that.")
```

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

```
def means_end_analysis(start, goal):
 print("Start String:", start)    print("Goal String :", goal)  print("\nSteps:")
    s = list(start)    g = list(goal)    i = 0
    while i < max(len(s), len(g)):
        if i < len(s) and i < len(g) and s[i] == g[i]:
            i += 1           continue
        if i < len(s) and i < len(g):
            print(f"Replace '{s[i]}' with '{g[i]}' at position {i}")
            s[i] = g[i]
            i += 1           continue
        if i >= len(s) and i < len(g):
            print(f"Insert '{g[i]}' at position {i}")
            s.append(g[i])
            i += 1           continue
        if i < len(s) and i >= len(g):
            print(f"Delete '{s[i]}' from position {i}")
            s.pop(i)           continue
    print("\nFinal Result:", "".join(s))
start = input("Enter start string: ")
goal = input("Enter goal string : ")
means_end_analysis(start, goal)
```

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

```
from collections import deque
def water_jug():
    visited = set()
    queue = deque()    queue.append((0, 0))
    while queue:
        x, y = queue.popleft()
       if (x, y) in visited:         continue
        visited.add((x, y))
        print("State:", (x, y))
        if y == 2:        print("\nGoal Achieved! 2 gallons in 3-gallon jug.")
            return        next_states = [ (4, y), (x, 3),(0, y), (x, 0)]
        pour = min(x, 3 - y)
        next_states.append((x - pour, y + pour))
        pour = min(y, 4 - x)
        next_states.append((x + pour, y - pour))
  for state in next_states:
            if state not in visited:
                queue.append(state)
water_jug()
```

Slip 11

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

```
import calendar
def print_month_calendar(year: int, month: int):
    print(calendar.month(year, month))
if __name__ == "__main__":
    y = int(input("Year (e.g. 2025): "))
    m = int(input("Month (1-12): "))
    print_month_calendar(y, m)
```

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

```
def is_safe(board, row, col):
    for i in range(row):
        if board[i] == col:           return False
    i, j = row - 1, col - 1
    while i >= 0 and j >= 0:
        if board[i] == j:            return False
        i -= 1      j -= 1
    i, j = row - 1, col + 1
    while i >= 0 and j < 4:
        if board[i] == j:           return False
        i -= 1        j += 1    return True
def solve_4_queens(row, board, solutions):
    if row == 4:
        solutions.append(board.copy())
        return
    for col in range(4):
        if is_safe(board, row, col):
            board[row] = col
            solve_4_queens(row + 1, board, solutions)
def display_solutions(solutions):
    print(f"Total solutions: {len(solutions)}\n")
    for sol in solutions:
        for row in range(4):
            line = ""
            for col in range(4):
                if sol[row] == col:
                    line += " Q "
         else:                line += " . "
            print(line)        print("\n")
solutions = []
board = [-1, -1, -1, -1]
solve_4_queens(0, board, solutions)
display_solutions(solutions)
```

Slip 12

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

```
def minimax(depth, node_index, is_max, scores):
    # Base case: leaf node reached
    if depth == 3:
        return scores[node_index]
    if is_max:        return max(
            minimax(depth+1, node_index*2, False, scores),
            minimax(depth+1, node_index*2 + 1, False, scores) )
 else: return min( minimax(depth+1, node_index*2, True, scores),
            minimax(depth+1, node_index*2 + 1, True, scores) )
scores = [3, 5, 2, 9, 12, 5, 23, 23]
print("The optimal value is:", minimax(0, 0, True, scores))
```

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

```
N = 8   def print_board(board):
    for row in board:
        print(" ".join(str(x) for x in row))
    print()
def is_safe(board, row, col):
    for i in range(col):
        if board[row][i] == 1:
            return False
    i, j = row, col
    while i >= 0 and j >= 0:
        if board[i][j] == 1:
            return False
        i -= 1        j -= 1
    i, j = row, col
    while j >= 0 and i < N:
        if board[i][j] == 1:
            return False
        i += 1        j -= 1    return True
def solve(board, col):
    if col >= N:        return True
    for i in range(N):
        if is_safe(board, i, col):
            board[i][col] = 1
            if solve(board, col + 1):     return True
            board[i][col] = 0     return False
board = [[0] * N for _ in range(N)]
if solve(board, 0):
    print("Solution to 8-Queens Problem:")
    print_board(board)
else:    print("No solution exists")
```

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

```
sentence = input("Enter a sentence: ")
words = sentence.split()
words.sort()
print("Sorted words:")
for w in words:    print(w)
```

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

```
def safe(board, row, col, n):
    for i in range(row):
        if board[i] == col or \
           board[i] - i == col - row or \
           board[i] + i == col + row:
            return False
    return True
def solve(board, row, n):
    if row == n:
        print(board)
        return True
    found = False
    for col in range(n):
        if safe(board, row, col, n):
            board[row] = col
            found = solve(board, row + 1, n) or found
    return found
n = int(input("Enter number of queens: "))
board = [-1] * n
solve(board, 0, n)
```

Slip 14

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

```
from collections import deque
def get_next_states(state):
    monkey, box, on_box, banana = state
    states = []
    for loc in ['door', 'window', 'middle']:
        states.append((loc, box, False, banana))
    if monkey == box and not on_box:
        for loc in ['door', 'window', 'middle']:
            states.append((loc, loc, False, banana))
    if monkey == box:
        states.append((monkey, box, True, banana))
    if monkey == 'middle' and box == 'middle' and on_box:
        states.append((monkey, box, on_box, True))
    return states
def solve():
    start = ('door', 'window', False, False)
    goal = ('middle', 'middle', True, True)
    q = deque([start])
    visited = set([start])
    while q:        state = q.popleft()
        print("State:", state)
        if state == goal:
            print("\nMonkey got the banana!")
            return
        for ns in get_next_states(state):
            if ns not in visited:
                visited.add(ns)
                q.append(ns)       solve()
```

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

```
graph = { 'A': ['B', 'C'],'B': ['D', 'E'],'C': ['F'],'D': [],
   'E': ['G'],  'F': [],   'G': [] }
goal = 'G'       def dls(node, depth):
    print("Visiting:", node)
    if node == goal:        return True
    if depth == 0:        return False
    for child in graph.get(node, []):
        if dls(child, depth - 1):            return True
    return False       def iddfs(start, max_depth):
    for depth in range(max_depth + 1):
        print(f"\nDepth Limit = {depth}")
        if dls(start, depth):
  print("\nGoal found at depth", depth)       return
    print("Goal not found")      iddfs('A', 5)
```

Slip15

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

```
def tower_of_hanoi(n, source, auxiliary, destination):
    if n == 1:
        print(f"Move disk 1 from {source} to {destination}")
        return
    tower_of_hanoi(n - 1, source, destination, auxiliary)
    print(f"Move disk {n} from {source} to {destination}")
    tower_of_hanoi(n - 1, auxiliary, source, destination)
n = int(input("Enter number of disks: "))
tower_of_hanoi(n, 'A', 'B', 'C')
```

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

```
board = [' ' for _ in range(9)]
def print_board():    print()
    print(board[0], "|", board[1], "|", board[2])
    print("--+---+--")
    print(board[3], "|", board[4], "|", board[5])
    print("--+---+--")
    print(board[6], "|", board[7], "|", board[8])    print()
def check_winner(player):
    win_conditions = [   (0,1,2), (3,4,5), (6,7,8),
        (0,3,6), (1,4,7), (2,5,8),  (0,4,8), (2,4,6) ]
    for a,b,c in win_conditions:
        if board[a] == board[b] == board[c] == player:
            return True
    return False
def is_full():
    return ' ' not in board
current_player = 'X'
while True:
    print_board()
    move = int(input(f"Player {current_player}, enter position (1-9): ")) - 1
    if board[move] != ' ':
        print("Invalid move! Try again.")
        continue
    board[move] = current_player
    if check_winner(current_player):
        print_board()
        print(f"Player {current_player} wins!")
        break
    if is_full():print_board()
        print("It's a draw!")          break
    current_player = 'O' if current_player == 'X' else 'X'
```

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

```
import random
import math
def f(x):
    return -x*x + 4*x
def hill_climb(func, start, step=0.1, max_iters=1000, lower=-100, upper=100):
    x = start
    fx = func(x)
    for i in range(max_iters):
        # consider neighbors left and right
        left = max(lower, x - step)
        right = min(upper, x + step)
        f_left = func(left)
        f_right = func(right)
        if f_left > fx and f_left >= f_right:
            x, fx = left, f_left
        elif f_right > fx and f_right >= f_left:
            x, fx = right, f_right
        else:
            break
    return x, fx
def hill_climbing_with_restarts(func, domain=(-10, 10), step=0.01, restarts=10):
    best_x, best_fx = None, -float('inf')
    low, high = domain
    for r in range(restarts):
        start = random.uniform(low, high)
        x, fx = hill_climb(func, start, step=step, max_iters=10000, lower=low, upper=high)
        if fx > best_fx:
            best_x, best_fx = x, fx
    return best_x, best_fx
if __name__ == "__main__":
    x0 = 0.0
    x_found, f_found = hill_climb(f, start=x0, step=0.01, max_iters=10000, lower=-10, upper=10)
    print("Single-run hill-climb:")
    print(f"  start={x0} -> x* = {x_found:.6f}, f(x*) = {f_found:.6f}")
    xr, fr = hill_climbing_with_restarts(f, domain=(-10, 10), step=0.001, restarts=20)
    print("With random restarts:")
    print(f"  x* = {xr:.6f}, f(x*) = {fr:.6f}")
    print("Analytical maximum: x = 2, f(2) = 4")
```

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]

```
def a_star(graph, heuristic, start, goal):
    open_set = set([start])
    closed_set = set()
    g_cost = {start: 0}
    parent = {start: None}
    while open_set:
        current = min(open_set, key=lambda x: g_cost[x] + heuristic[x])
        if current == goal:
            path = []
            while current is not None:
                path.append(current)
                current = parent[current]
            return path[::-1]
        open_set.remove(current)
        closed_set.add(current)
        for neighbor, cost in graph[current].items():
            if neighbor in closed_set:
                continue
            tentative_g = g_cost[current] + cost
            if neighbor not in open_set:
                open_set.add(neighbor)
            elif tentative_g >= g_cost[neighbor]:
                continue
            parent[neighbor] = current
            g_cost[neighbor] = tentative_g

    return None
graph = { 'A': {'B': 2, 'C': 4},    'B': {'D': 7, 'E': 3},
    'C': {'F': 5},    'D': {'G': 1},    'E': {'G': 8},
    'F': {'G': 2},    'G': {}  }
heuristic = {    'A': 10,
    'B': 8,
    'C': 7,
    'D': 3,
    'E': 4,
    'F': 2,
    'G': 0  }
start = 'A'
goal = 'G'
path = a_star(graph, heuristic, start, goal)
print("Shortest Path using A* :", path)
```

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

```
import nltk
from nltk.corpus import stopwords
from nltk.tokenize import word_tokenize
nltk.download('punkt')
nltk.download('stopwords')
file_path = "input.txt"
with open(file_path, "r") as file:
    text = file.read()
words = word_tokenize(text)
stop_words = set(stopwords.words('english'))
filtered_words = [word for word in words if word.lower() not in stop_words]
print("Original Text:")
print(text)
print("\nText After Removing Stop Words:")
print(" ".join(filtered_words))
```

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

```
graph = { 'A': ['B', 'C'],'B': ['D', 'E'],'C': ['F'],'D': [],
    'E': ['G'],  'F': [],   'G': [] }
goal = 'G'       def dls(node, depth):
    print("Visiting:", node)
    if node == goal:        return True
    if depth == 0:        return False
    for child in graph.get(node, []):
        if dls(child, depth - 1):            return True
    return False       def iddfs(start, max_depth):
    for depth in range(max_depth + 1):
        print(f"\nDepth Limit = {depth}")
        if dls(start, depth):
  print("\nGoal found at depth", depth)       return
    print("Goal not found")      iddfs('A', 5)
```

Slip 18

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

```
import random
def hangman():
    words = ["python", "computer", "program", "hangman", "science"]
    word = random.choice(words)     # choose a random word
    word = word.lower()
    guessed = []
    attempts = 6
    print("====== Hangman Game ======")
    print("Guess the word!")
    display = ["_" for _ in word]
    while attempts > 0 and "_" in display:
        print("\nWord: ", " ".join(display))
        print("Attempts left:", attempts)
        print("Guessed letters:", guessed)
        ch = input("Enter a single alphabet: ").lower()
        if len(ch) != 1 or not ch.isalpha():
            print("Invalid input! Enter only one alphabet.")
            continue
        if ch in guessed:
            print("You already guessed that letter!")
            continue
        guessed.append(ch)
        if ch in word:
            print("Correct guess!")
            for i in range(len(word)):
                if word[i] == ch:
                    display[i] = ch
        else:
            print("Wrong guess!")
            attempts -= 1
    if "_" not in display:
        print("\nCongratulations! You guessed the word:", word)
    else:
        print("\nGame Over! You failed to guess the word.")
        print("The correct word was:", word)
hangman()
```

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

```
def is_safe(board, row, col):
    for i in range(row):
        if board[i] == col:
      return False
    i, j = row - 1, col - 1
    while i >= 0 and j >= 0:
        if board[i] == j:
    return False
        i -= 1
     j -= 1
    i, j = row - 1, col + 1
    while i >= 0 and j < 4:
        if board[i] == j:
  return False
        i -= 1
   j += 1
  return True
def solve_4_queens(row, board, solutions):
    if row == 4:
        solutions.append(board.copy())
        return
    for col in range(4):
        if is_safe(board, row, col):
            board[row] = col
            solve_4_queens(row + 1, board, solutions)
def display_solutions(solutions):
    print(f"Total solutions: {len(solutions)}\n")
    for sol in solutions:
        for row in range(4):
            line = ""
            for col in range(4):
                if sol[row] == col:
                    line += " Q "
         else:
        line += " . "
            print(line)
    print("\n")
solutions = []
board = [-1, -1, -1, -1]
solve_4_queens(0, board, solutions)
display_solutions(solutions)
```

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

```
import itertools
letters = ('T', 'W', 'O', 'F', 'U', 'R')
for digits in itertools.permutations(range(10), 6):
    mapping = dict(zip(letters, digits))
    if mapping['T'] == 0 or mapping['F'] == 0:
        continue
    TWO = 100*mapping['T'] + 10*mapping['W'] + mapping['O']
    FOUR = (1000*mapping['F'] + 100*mapping['O'] +
            10*mapping['U'] + mapping['R'])
    if TWO + TWO == FOUR:
        print("Solution Found!")
        print(mapping)
        print("TWO =", TWO)
        print("FOUR =", FOUR)
        break
else:         print("No solution.")
```

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

```
def minimax(depth, node_index, is_max, scores):
    # Base case: leaf node reached
    if depth == 3:
        return scores[node_index]
    if is_max:        return max(
            minimax(depth+1, node_index*2, False, scores),
            minimax(depth+1, node_index*2 + 1, False, scores) )
 else: return min( minimax(depth+1, node_index*2, True, scores),
            minimax(depth+1, node_index*2 + 1, True, scores) )
scores = [3, 5, 2, 9, 12, 5, 23, 23]
print("The optimal value is:", minimax(0, 0, True, scores))
```

Slip 20

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

```
import string
def remove_punct(s: str) -> str:
    return s.translate(str.maketrans('', '', string.punctuation))
if __name__ == "__main__":
    s = input("Enter text: ")
    print(remove_punct(s))
```

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

```
def safe(board, row, col, n):
    for i in range(row):
        if board[i] == col or \
           board[i] - i == col - row or \
           board[i] + i == col + row:
            return False
    return True
def solve(board, row, n):
    if row == n:
        print(board)
        return True
    found = False
    for col in range(n):
        if safe(board, row, col, n):
            board[row] = col
            found = solve(board, row + 1, n) or found
    return found
n = int(input("Enter number of queens: "))
board = [-1] * n
solve(board, 0, n)
```

Slip 21

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

```
def alphabeta(depth, idx, maxPlayer, values, alpha, beta):
    if depth == 3:
        return values[idx]
    if maxPlayer:
        best = -999
        for i in range(2):
            val = alphabeta(depth+1, idx*2+i, False, values, alpha, beta)
            best = max(best, val)
            alpha = max(alpha, best)
            if beta <= alpha: break        return best
    else:
        best = 999
        for i in range(2):
            val = alphabeta(depth+1, idx*2+i, True, values, alpha, beta)
            best = min(best, val)
            beta = min(beta, best)
            if beta <= alpha: break
        return best
values = [3, 5, 6, 9, 1, 2, 0, -1]
print("Optimal value:", alphabeta(0, 0, True, values, -999, 999))
```

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

```
# Simple Chatbot Program

print("Simple Chatbot")
print("Type 'bye' to exit.")
print("---------------------------------")

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

    if user == "bye":
        print("Chatbot: Goodbye! Have a great day!")
        break

    elif "hello" in user or "hi" in user:
        print("Chatbot: Hello! How can I help you?")

    elif "how are you" in user:
        print("Chatbot: I am fine! Thanks for asking.")

    elif "name" in user:
        print("Chatbot: I am a simple Python Chatbot.")

    elif "time" in user:
        from datetime import datetime
        now = datetime.now().strftime("%H:%M:%S")
        print("Chatbot: Current time is", now)

    elif "thank" in user:
        print("Chatbot: You're welcome!")

    else:
        print("Chatbot: Sorry, I don't understand that.")
```

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

```
def tower_of_hanoi(n, source, auxiliary, destination):
    if n == 1:
        print(f"Move disk 1 from {source} to {destination}")
        return
    tower_of_hanoi(n - 1, source, destination, auxiliary)
    print(f"Move disk {n} from {source} to {destination}")
    tower_of_hanoi(n - 1, auxiliary, source, destination)
n = int(input("Enter number of disks: "))
tower_of_hanoi(n, 'A', 'B', 'C')
```

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]

```
import random
def hangman(words=None, max_wrong=6):
    if words is None:
        words = ["python","computer","hangman","programming","algorithm"]
    word = random.choice(words).lower()
    guessed = set()
    wrong = 0
    display = ["_" if c.isalpha() else c for c in word]
    while wrong < max_wrong and "_" in display:
        print("\nWord:"," ".join(display))
        print(f"Wrong guesses: {wrong}/{max_wrong}", "Guessed:", " ".join(sorted(guessed)))
        ch = input("Enter a letter: ").lower().strip()
        if not ch or not ch.isalpha() or len(ch)!=1:
            print("Enter single alphabet.")
            continue
        if ch in guessed:
            print("Already guessed.")
            continue
        guessed.add(ch)
        if ch in word:
            for i,c in enumerate(word):
                if c==ch: display[i]=ch
        else:
            wrong += 1
    if "_" not in display:
        print("\nYou won! Word:", word)
    else:
        print("\nYou lost! Word:", word)
if __name__ == "__main__":
    hangman()
```

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

```
sentence = input("Enter a sentence: ")
words = sentence.split()
words.sort()
print("Sorted words:")
for w in words:
   print(w)
```

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

```
def dfs_recursive(graph, start, goal, visited=None, path=None):
   if visited is None:   visited = set()
   if path is None:     path = []
   visited.add(start)
   path.append(start)
   if start == goal:       return path[:]
   for neighbor in graph.get(start, []):
      if neighbor not in visited:
         res = dfs_recursive(graph, neighbor, goal, visited, path)
         if res is not None:         return res
   path.pop()    return None
def dfs_iterative(graph, start, goal):
   stack = [(start, 0, [start])]
   visited = set()
   while stack:
      node, idx, path = stack.pop()
      if node == goal:   return path
      if node not in visited:
         visited.add(node)
         neighbors = graph.get(node, [])
         for nb in reversed(neighbors):
            if nb not in visited:
               stack.append((nb, 0, path + [nb]))    return None
if __name__ == "__main__":
   graph = {1: [2, 3],2: [4, 5], 3: [6, 7],4: [],5: [8],
      6: [],7: [8],8: []    }
  start_node = 1
   goal_node = 8
   print("Graph adjacency list:")
   for k in sorted(graph.keys()):
      print(f"  {k}: {graph[k]}")
   print("\nDFS (recursive) from 1 to 8:")
   path_rec = dfs_recursive(graph, start_node, goal_node)
   if path_rec:
```

```
        print("  Path found:", " -> ".join(map(str, path_rec)))
    else:       print("  No path found.")
    print("\nDFS (iterative) from 1 to 8:")
    path_it = dfs_iterative(graph, start_node, goal_node)
    if path_it:
        print("  Path found:", " -> ".join(map(str, path_it)))
    else:
        print("  No path found.")
```

Slip 24

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

```
import random
import math
def f(x):
    return -x*x + 4*x
def hill_climb(func, start, step=0.1, max_iters=1000, lower=-100, upper=100):
    x = start
    fx = func(x)
    for i in range(max_iters):
        # consider neighbors left and right
        left = max(lower, x - step)
        right = min(upper, x + step)
        f_left = func(left)
        f_right = func(right)
        if f_left > fx and f_left >= f_right:
            x, fx = left, f_left
        elif f_right > fx and f_right >= f_left:
            x, fx = right, f_right
        else:
            break
    return x, fx
def hill_climbing_with_restarts(func, domain=(-10, 10), step=0.01, restarts=10):
    best_x, best_fx = None, -float('inf')
    low, high = domain
    for r in range(restarts):
        start = random.uniform(low, high)
        x, fx = hill_climb(func, start, step=step, max_iters=10000, lower=low, upper=high)
        if fx > best_fx:
            best_x, best_fx = x, fx
    return best_x, best_fx
if __name__ == "__main__":
    x0 = 0.0
```

```
    x_found, f_found = hill_climb(f, start=x0, step=0.01, max_iters=10000, lower=-10, upper=10)
    print("Single-run hill-climb:")
    print(f"  start={x0} -> x* = {x_found:.6f}, f(x*) = {f_found:.6f}")
    xr, fr = hill_climbing_with_restarts(f, domain=(-10, 10), step=0.001, restarts=20)
    print("With random restarts:")
    print(f"  x* = {xr:.6f}, f(x*) = {fr:.6f}")
    print("Analytical maximum: x = 2, f(2) = 4")
```

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

```
from collections import deque
def get_neighbors(state):
    neighbors = []
    zero = state.index(0)
    x, y = divmod(zero, 3)
    moves = [(-1,0),(1,0),(0,-1),(0,1)]
    for dx, dy in moves:
        nx, ny = x + dx, y + dy
        if 0 <= nx < 3 and 0 <= ny < 3:
            new_zero = nx * 3 + ny
            new_state = list(state)
            new_state[zero], new_state[new_zero] = new_state[new_zero], new_state[zero]
            neighbors.append(tuple(new_state))
    return neighbors
def bfs(start, goal):
    queue = deque()
    queue.append((start, [start]))
    visited = set()
    while queue:
        current, path = queue.popleft()
        if current == goal:
            return path
        visited.add(current)
for neighbor in get_neighbors(current):
            if neighbor not in visited:
                queue.append((neighbor, path + [neighbor]))
    return None
start = (1, 2, 3,  4, 0, 5, 6, 7, 8)
goal = (1, 2, 3,4, 5, 6, 7, 8, 0)
solution = bfs(start, goal)
print("\nSolution Path:")
for state in solution:
    print(state[0:3])
    print(state[3:6])
    print(state[6:9])
    print("-----")
```

Slip 25