(Slip 1)

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

Ans-

import math

def objective\_function(x):

# Example mathematical function: f(x) = -x^2 + 4x

return -(x \*\* 2) + 4 \* x

def hill\_climbing(starting\_point, step\_size, iterations):

current\_x = starting\_point

for \_ in range(iterations):

current\_value = objective\_function(current\_x)

# Move to the neighboring point with a higher value

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)

if \_\_name\_\_ == "\_\_main\_\_":

starting\_point = 0.0

step\_size = 0.1

iterations = 100

max\_x, max\_value = hill\_climbing(starting\_point, step\_size, iterations)

print(f"Maximum value found at x = {max\_x}, f(x) = {max\_value}")

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

**Ans-**

**from collections import defaultdict**

**class Graph:**

**def \_\_init\_\_(self):**

**self.graph = defaultdict(list)**

**def add\_edge(self, u, v):**

**self.graph[u].append(v)**

**self.graph[v].append(u) # Assuming an undirected graph**

**def dfs(graph, start, goal):**

**visited = set()**

**stack = [start]**

**while stack:**

**current\_node = stack.pop()**

**if current\_node == goal:**

**print(f"Goal node {goal} found!")**

**return True**

**if current\_node not in visited:**

**print(f"Visited node {current\_node}")**

**visited.add(current\_node)**

**stack.extend(reversed(graph[current\_node]))**

**print("Goal node not found.")**

**return False**

**if \_\_name\_\_ == "\_\_main\_\_":**

**# Create the graph**

**graph = Graph()**

**edges = [(1, 2), (1, 3), (2, 4), (2, 5), (3, 6), (3, 7), (4, 8), (5, 8), (6, 8), (7, 8)]**

**for edge in edges:**

**graph.add\_edge(\*edge)**

**# DFS from initial node 1 to goal node 8**

**initial\_node = 1**

**goal\_node = 8**

**dfs(graph.graph, initial\_node, goal\_node)**

**(Slip-2)**

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

Ans-

**import calendar**

**def generate\_calendar(year, month):**

**cal = calendar.monthcalendar(year, month)**

**# Display the header**

**print(calendar.month\_name[month], year)**

**print("Mo Tu We Th Fr Sa Su")**

**# Display the calendar**

**for week in cal:**

**for day in week:**

**if day == 0:**

**print(" ", end=" ")**

**else:**

**print(f"{day:2} ", end=" ")**

**print()**

**if \_\_name\_\_ == "\_\_main\_\_":**

**# Input: Year and Month**

**year = int(input("Enter the year: "))**

**month = int(input("Enter the month (1-12): "))**

**# Check if the input month is valid**

**if 1 <= month <= 12:**

**generate\_calendar(year, month)**

**else:**

**print("Invalid month. Please enter a month between 1 and 12.")**

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

Ans-

**from collections import defaultdict**

**class Graph:**

**def \_\_init\_\_(self):**

**self.graph = defaultdict(list)**

**def add\_edge(self, u, v):**

**self.graph[u].append(v)**

**self.graph[v].append(u) # Assuming an undirected graph**

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

**if visited is None:**

**visited = set()**

**visited.add(start)**

**print(f"Visited node {start}")**

**if start == goal:**

**print(f"Goal node {goal} found!")**

**return True**

**for neighbor in graph[start]:**

**if neighbor not in visited:**

**if dfs(graph, neighbor, goal, visited):**

**return True**

**return False**

**if \_\_name\_\_ == "\_\_main\_\_":**

**# Create the graph**

**graph = Graph()**

**edges = [(1, 2), (1, 3), (2, 4), (2, 5), (3, 6), (3, 7)]**

**for edge in edges:**

**graph.add\_edge(\*edge)**

**# DFS from initial node 1 to goal node 7**

**initial\_node = 1**

**goal\_node = 7**

**dfs(graph.graph, initial\_node, goal\_node)**

**(SLIP-3)**

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

**Ans-**

**import string**

**def remove\_punctuations(input\_string):**

**# Create a translation table to map each punctuation character to None**

**translator = str.maketrans("", "", string.punctuation)**

**# Use translate to remove punctuations**

**result = input\_string.translate(translator)**

**return result**

**if \_\_name\_\_ == "\_\_main\_\_":**

**# Input string with punctuations**

**input\_string = "Hello, World! This is an example string with punctuations!!!"**

**# Remove punctuations**

**result\_string = remove\_punctuations(input\_string)**

**# Display the result**

**print("Original String:", input\_string)**

**print("String without Punctuations:", result\_string)**

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

**Ans-**

**class Graph:**

**def \_\_init\_\_(self):**

**self.graph = {}**

**def add\_edge(self, u, v):**

**if u in self.graph:**

**self.graph[u].append(v)**

**else:**

**self.graph[u] = [v]**

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

**if visited is None:**

**visited = set()**

**visited.add(start)**

**print(f"Visited node {start}")**

**if start == goal:**

**print(f"Goal node {goal} found!")**

**return True**

**if start in graph:**

**for neighbor in graph[start]:**

**if neighbor not in visited:**

**if dfs(graph, neighbor, goal, visited):**

**return True**

**return False**

**if \_\_name\_\_ == "\_\_main\_\_":**

**# Create the graph**

**graph = Graph()**

**edges = [(1, 2), (1, 3), (2, 4), (2, 5), (3, 6), (3, 7)]**

**for edge in edges:**

**graph.add\_edge(\*edge)**

**# DFS from initial node 2 to goal node 7**

**initial\_node = 2**

**goal\_node = 7**

**dfs(graph.graph, initial\_node, goal\_node)**

**(Slip-4)**

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

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

Ans-

**import random**

**def choose\_word():**

**words = ["python", "hangman", "computer", "programming", "developer", "coding"]**

**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**

**def hangman():**

**print("Welcome to Hangman!")**

**# Choose a random word**

**secret\_word = choose\_word()**

**# Initialize variables**

**guessed\_letters = []**

**attempts = 6**

**while attempts > 0:**

**# Display the current state of the word**

**current\_display = display\_word(secret\_word, guessed\_letters)**

**print(f"\nWord: {current\_display}")**

**# Get user input**

**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 secret\_word:**

**print("Correct guess!")**

**else:**

**print("Incorrect guess.")**

**attempts -= 1**

**guessed\_letters.append(guess)**

**else:**

**print("Invalid input. Please enter a single alphabet.")**

**continue**

**# Check if the user has guessed all letters**

**if set(secret\_word) <= set(guessed\_letters):**

**print(f"\nCongratulations! You guessed the word: {secret\_word}")**

**break**

**# Display remaining attempts**

**print(f"Attempts remaining: {attempts}")**

**if attempts == 0:**

**print(f"\nSorry, you ran out of attempts. The word was: {secret\_word}")**

**if \_\_name\_\_ == "\_\_main\_\_":**

**hangman()**

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

**Ans-**

**from collections import defaultdict, deque**

**class Graph:**

**def \_\_init\_\_(self):**

**self.graph = defaultdict(list)**

**def add\_edge(self, u, v):**

**self.graph[u].append(v)**

**self.graph[v].append(u) # Assuming an undirected graph**

**def bfs(graph, start, goal):**

**visited = set()**

**queue = deque([start])**

**while queue:**

**current\_node = queue.popleft()**

**if current\_node == goal:**

**print(f"Goal node {goal} found!")**

**return True**

**if current\_node not in visited:**

**print(f"Visited node {current\_node}")**

**visited.add(current\_node)**

**queue.extend(graph[current\_node])**

**print("Goal node not found.")**

**return False**

**if \_\_name\_\_ == "\_\_main\_\_":**

**# Create the graph**

**graph = Graph()**

**edges = [(1, 2), (1, 3), (2, 4), (2, 5), (3, 6), (3, 7), (4, 8), (5, 8), (6, 8), (7, 8)]**

**for edge in edges:**

**graph.add\_edge(\*edge)**

**# BFS from initial node 1 to goal node 8**

**initial\_node = 1**

**goal\_node = 8**

**bfs(graph.graph, initial\_node, goal\_node)**

**(Slip-5)**

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

**Ans-**

**import nltk**

**from nltk.stem import WordNetLemmatizer**

**from nltk.tokenize import word\_tokenize**

**nltk.download('punkt')**

**nltk.download('wordnet')**

**def lemmatize\_text(text):**

**lemmatizer = WordNetLemmatizer()**

**tokens = word\_tokenize(text)**

**lemmatized\_tokens = [lemmatizer.lemmatize(token) for token in tokens]**

**lemmatized\_text = ' '.join(lemmatized\_tokens)**

**return lemmatized\_text**

**if \_\_name\_\_ == "\_\_main\_\_":**

**input\_text = "Lemmatization is an important process for natural language processing."**

**lemmatized\_text = lemmatize\_text(input\_text)**

**print("Original Text:")**

**print(input\_text)**

**print("\nLemmatized Text:")**

**print(lemmatized\_text)**

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

**Ans-**

**from collections import defaultdict, deque**

**class Graph:**

**def \_\_init\_\_(self):**

**self.graph = defaultdict(list)**

**def add\_edge(self, u, v):**

**self.graph[u].append(v)**

**self.graph[v].append(u) # Assuming an undirected graph**

**def bfs(graph, start, goal):**

**visited = set()**

**queue = deque([(start, [start])])**

**while queue:**

**current\_node, path = queue.popleft()**

**if current\_node == goal:**

**print(f"Goal node {goal} found! Path: {path}")**

**return True**

**if current\_node not in visited:**

**visited.add(current\_node)**

**for neighbor in graph[current\_node]:**

**if neighbor not in visited:**

**queue.append((neighbor, path + [neighbor]))**

**print("Goal node not found.")**

**return False**

**if \_\_name\_\_ == "\_\_main\_\_":**

**# Create the graph**

**graph = Graph()**

**edges = [(1, 2), (1, 3), (2, 4), (2, 5), (3, 6), (3, 7), (4, 8), (5, 8), (6, 8), (7, 8)]**

**for edge in edges:**

**graph.add\_edge(\*edge)**

**# BFS from initial node 1 to goal node 8**

**initial\_node = 1**

**goal\_node = 8**

**bfs(graph.graph, initial\_node, goal\_node)**

**(SLIP-6)**

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

**Ans-**

**import nltk**

**nltk.download('stopwords')**

**import nltk**

**from nltk.corpus import stopwords**

**from nltk.tokenize import word\_tokenize**

**nltk.download('punkt') # Download the punkt tokenizer data**

**def remove\_stop\_words(input\_text):**

**stop\_words = set(stopwords.words('english'))**

**words = word\_tokenize(input\_text)**

**filtered\_words = [word.lower() for word in words if word.lower() not in stop\_words]**

**return ' '.join(filtered\_words)**

**if \_\_name\_\_ == "\_\_main\_\_":**

**# Read the passage from the text file**

**file\_path = 'your\_text\_file.txt' # Replace with the path to your text file**

**with open(file\_path, 'r', encoding='utf-8') as file:**

**passage = file.read()**

**# Remove stop words**

**processed\_passage = remove\_stop\_words(passage)**

**# Print the original and processed passages**

**print("Original Passage:")**

**print(passage)**

**print("\nPassage after removing stop words:")**

**print(processed\_passage)**

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

Ans-

**from collections import defaultdict, deque**

**class Graph:**

**def \_\_init\_\_(self):**

**self.graph = defaultdict(list)**

**def add\_edge(self, u, v):**

**self.graph[u].append(v)**

**self.graph[v].append(u) # Assuming an undirected graph**

**def bfs(graph, start, goal):**

**visited = set()**

**queue = deque([(start, [start])])**

**while queue:**

**current\_node, path = queue.popleft()**

**if current\_node == goal:**

**print(f"Goal node {goal} found! Path: {path}")**

**return True**

**if current\_node not in visited:**

**visited.add(current\_node)**

**for neighbor in graph[current\_node]:**

**if neighbor not in visited:**

**queue.append((neighbor, path + [neighbor]))**

**print("Goal node not found.")**

**return False**

**if \_\_name\_\_ == "\_\_main\_\_":**

**# Create the graph**

**graph = Graph()**

**edges = [(1, 2), (1, 3), (2, 4), (2, 5), (3, 6), (3, 7), (4, 8), (5, 8), (6, 8), (7, 8)]**

**for edge in edges:**

**graph.add\_edge(\*edge)**

**# BFS from initial node 1 to goal node 8**

**initial\_node = 1**

**goal\_node = 8**

**bfs(graph.graph, initial\_node, goal\_node)**

**(Slip-7)**

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

**Ans-**

**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 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\_full(board):**

**return all(board[i][j] != ' ' for i in range(3) for j in range(3))**

**def is\_terminal(board):**

**return is\_winner(board, 'X') or is\_winner(board, 'O') or is\_full(board)**

**def evaluate(board):**

**if is\_winner(board, 'X'):**

**return 1**

**elif is\_winner(board, 'O'):**

**return -1**

**else:**

**return 0**

**def minimax(board, depth, alpha, beta, maximizing\_player):**

**if is\_terminal(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] = 'X'**

**eval = minimax(board, depth + 1, alpha, beta, False)**

**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] = 'O'**

**eval = minimax(board, depth + 1, alpha, beta, True)**

**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 = None**

**for i in range(3):**

**for j in range(3):**

**if board[i][j] == ' ':**

**board[i][j] = 'X'**

**move\_val = minimax(board, 0, -math.inf, math.inf, False)**

**board[i][j] = ' '**

**if move\_val > best\_val:**

**best\_move = (i, j)**

**best\_val = move\_val**

**return best\_move**

**def main():**

**board = [[' ' for \_ in range(3)] for \_ in range(3)]**

**while not is\_terminal(board):**

**print\_board(board)**

**player\_move = None**

**while player\_move not in [(i, j) for i in range(3) for j in range(3)] or board[player\_move[0]][player\_move[1]] != ' ':**

**player\_move = tuple(map(int, input("Enter your move (row and column, separated by space): ").split()))**

**board[player\_move[0]][player\_move[1]] = 'O'**

**if is\_terminal(board):**

**break**

**computer\_move = find\_best\_move(board)**

**print(f"Computer's move: {computer\_move}")**

**board[computer\_move[0]][computer\_move[1]] = 'X'**

**print\_board(board)**

**if is\_winner(board, 'X'):**

**print("Computer wins!")**

**elif is\_winner(board, 'O'):**

**print("You win!")**

**else:**

**print("It's a draw!")**

**if \_\_name\_\_ == "\_\_main\_\_":**

**main()**

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

**Ans-**

**import random**

**def get\_response(user\_input):**

**user\_input = user\_input.lower()**

**if "hello" in user\_input or "hi" in user\_input:**

**return "Hello! How can I help you today?"**

**elif "your name" in user\_input:**

**return "I'm a simple chatbot. You can call me ChatBot."**

**elif "how are you" in user\_input:**

**return "I'm just a computer program, but thanks for asking!"**

**elif "bye" in user\_input or "goodbye" in user\_input:**

**return "Goodbye! Have a great day!"**

**else:**

**return "I'm sorry, I didn't understand that. Can you please rephrase?"**

**def main():**

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

**while True:**

**user\_input = input("You: ")**

**if user\_input.lower() == 'bye':**

**print("Simple Chatbot: Goodbye! Have a great day!")**

**break**

**response = get\_response(user\_input)**

**print("Simple Chatbot:", response)**

**if \_\_name\_\_ == "\_\_main\_\_":**

**main()**

**(Slip-8)**

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

**Ans-**

**def count\_upper\_lower(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**

**if \_\_name\_\_ == "\_\_main\_\_":**

**user\_input = input("Enter a string: ")**

**upper, lower = count\_upper\_lower(user\_input)**

**print(f"Number of uppercase alphabets: {upper}")**

**print(f"Number of lowercase alphabets: {lower}")**

**W**

**Example**

**Enter a string: Hello World!**

**Number of uppercase alphabets: 2**

**Number of lowercase alphabets: 8**

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

**Ans-**

**def print\_board(board):**

**for row in board:**

**print(" ".join(row))**

**print()**

**def check\_winner(board):**

**# Check rows, columns, and diagonals**

**for i in range(3):**

**if all(board[i][j] == 'X' for j in range(3)) or all(board[j][i] == 'X' for j in range(3)):**

**return 'X'**

**if all(board[i][j] == 'O' for j in range(3)) or all(board[j][i] == 'O' for j in range(3)):**

**return 'O'**

**if all(board[i][i] == 'X' for i in range(3)) or all(board[i][2 - i] == 'X' for i in range(3)):**

**return 'X'**

**if all(board[i][i] == 'O' for i in range(3)) or all(board[i][2 - i] == 'O' for i in range(3)):**

**return 'O'**

**return None**

**def check\_draw(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 the row (0, 1, or 2): "))**

**col = int(input(f"Player {current\_player}, enter the column (0, 1, or 2): "))**

**if 0 <= row < 3 and 0 <= col < 3 and board[row][col] == ' ':**

**board[row][col] = current\_player**

**winner = check\_winner(board)**

**if winner:**

**print\_board(board)**

**print(f"Player {winner} wins!")**

**break**

**elif check\_draw(board):**

**print\_board(board)**

**print("It's a draw!")**

**break**

**else:**

**current\_player = 'O' if current\_player == 'X' else 'X'**

**else:**

**print("Invalid move. Try again.")**

**if \_\_name\_\_ == "\_\_main\_\_":**

**tic\_tac\_toe()**

**(Slip-9)**

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

**Ans-**

**from heapq import heappop, heappush**

**class PuzzleNode:**

**def \_\_init\_\_(self, state, parent=None, move=None, cost=0, heuristic=0):**

**self.state = state**

**self.parent = parent**

**self.move = move**

**self.cost = cost**

**self.heuristic = heuristic**

**def \_\_lt\_\_(self, other):**

**return (self.cost + self.heuristic) < (other.cost + other.heuristic)**

**def print\_board(board):**

**for row in board:**

**print(" ".join(map(str, row)))**

**print()**

**def get\_goal\_state():**

**return [[1, 2, 3], [4, 5, 6], [7, 8, 0]]**

**def get\_manhattan\_distance(board):**

**distance = 0**

**for i in range(3):**

**for j in range(3):**

**if board[i][j] != 0:**

**goal\_row, goal\_col = divmod(board[i][j] - 1, 3)**

**distance += abs(i - goal\_row) + abs(j - goal\_col)**

**return distance**

**def get\_neighbors(node):**

**neighbors = []**

**row, col = None, None**

**for i in range(3):**

**for j in range(3):**

**if node.state[i][j] == 0:**

**row, col = i, j**

**moves = [(0, 1), (1, 0), (0, -1), (-1, 0)]**

**for dr, dc in moves:**

**new\_row, new\_col = row + dr, col + dc**

**if 0 <= new\_row < 3 and 0 <= new\_col < 3:**

**new\_state = [row[:] for row in node.state]**

**new\_state[row][col], new\_state[new\_row][new\_col] = new\_state[new\_row][new\_col], new\_state[row][col]**

**neighbors.append(PuzzleNode(new\_state, node, move=(new\_row, new\_col), cost=node.cost + 1, heuristic=get\_manhattan\_distance(new\_state)))**

**return neighbors**

**def solve\_8\_puzzle(initial\_state):**

**initial\_node = PuzzleNode(initial\_state, None, move=None, cost=0, heuristic=get\_manhattan\_distance(initial\_state))**

**goal\_state = get\_goal\_state()**

**open\_set = [initial\_node]**

**closed\_set = set()**

**while open\_set:**

**current\_node = heappop(open\_set)**

**if current\_node.state == goal\_state:**

**path = []**

**while current\_node:**

**path.append((current\_node.move[0], current\_node.move[1]))**

**current\_node = current\_node.parent**

**return path[::-1]**

**closed\_set.add(tuple(map(tuple, current\_node.state)))**

**neighbors = get\_neighbors(current\_node)**

**for neighbor in neighbors:**

**if tuple(map(tuple, neighbor.state)) not in closed\_set:**

**heappush(open\_set, neighbor)**

**return None**

**if \_\_name\_\_ == "\_\_main\_\_":**

**# Example initial state**

**initial\_state = [[1, 2, 3], [4, 5, 6], [0, 7, 8]]**

**print("Initial State:")**

**print\_board(initial\_state)**

**solution = solve\_8\_puzzle(initial\_state)**

**if solution:**

**print("Solution:")**

**for move in solution:**

**print(f"Move {move}:")**

**print\_board(initial\_state)**

**initial\_state[move[0]][move[1]], initial\_state[move[0] - 1][move[1]] = initial\_state[move[0] - 1][move[1]], initial\_state[move[0]][move[1]]**

**else:**

**print("No solution found.")**

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

**Ans-**

**def water\_jug\_dfs(jug1, jug2, target, path, visited):**

**if (jug1, jug2) == target:**

**return path**

**visited.add((jug1, jug2))**

**# Fill jug1**

**if jug1 < 5 and (5, jug2) not in visited:**

**fill\_jug1\_path = water\_jug\_dfs(5, jug2, target, path + [(5, jug2)], visited.copy())**

**if fill\_jug1\_path:**

**return fill\_jug1\_path**

**# Fill jug2**

**if jug2 < 7 and (jug1, 7) not in visited:**

**fill\_jug2\_path = water\_jug\_dfs(jug1, 7, target, path + [(jug1, 7)], visited.copy())**

**if fill\_jug2\_path:**

**return fill\_jug2\_path**

**# Pour water from jug1 to jug2**

**pour\_jug1\_to\_jug2 = min(jug1, 7 - jug2)**

**if pour\_jug1\_to\_jug2 > 0 and (jug1 - pour\_jug1\_to\_jug2, jug2 + pour\_jug1\_to\_jug2) not in visited:**

**pour\_path = water\_jug\_dfs(jug1 - pour\_jug1\_to\_jug2, jug2 + pour\_jug1\_to\_jug2, target, path + [(jug1 - pour\_jug1\_to\_jug2, jug2 + pour\_jug1\_to\_jug2)], visited.copy())**

**if pour\_path:**

**return pour\_path**

**# Pour water from jug2 to jug1**

**pour\_jug2\_to\_jug1 = min(jug2, 5 - jug1)**

**if pour\_jug2\_to\_jug1 > 0 and (jug1 + pour\_jug2\_to\_jug1, jug2 - pour\_jug2\_to\_jug1) not in visited:**

**pour\_path = water\_jug\_dfs(jug1 + pour\_jug2\_to\_jug1, jug2 - pour\_jug2\_to\_jug1, target, path + [(jug1 + pour\_jug2\_to\_jug1, jug2 - pour\_jug2\_to\_jug1)], visited.copy())**

**if pour\_path:**

**return pour\_path**

**return None**

**def solve\_water\_jug\_problem(capacity\_jug1, capacity\_jug2, target):**

**initial\_state = (0, 0)**

**target\_state = (0, target)**

**path = water\_jug\_dfs(\*initial\_state, target\_state, [initial\_state], set())**

**if path:**

**print("Solution found:")**

**for state in path:**

**print(f"Jug 1: {state[0]} gallons, Jug 2: {state[1]} gallons")**

**else:**

**print("No solution found.")**

**if \_\_name\_\_ == "\_\_main\_\_":**

**jug1\_capacity = 5**

**jug2\_capacity = 7**

**target\_gallons = 4**

**solve\_water\_jug\_problem(jug1\_capacity, jug2\_capacity, target\_gallons)**

**(Slip-10)**

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

**Ans-**

**T W O**

**+ T W O**

**--------**

**F O U R**

**from itertools import permutations**

**def is\_valid(puzzle):**

**t, w, o, f, u, r = puzzle**

**if t == 0 or f == 0:**

**return False**

**two = 100\*t + 10\*w + o**

**four = 1000\*f + 100\*o + 10\*u + r**

**return two + two == four**

**def solve\_cryptarithmetic():**

**for perm in permutations(range(10), 6):**

**puzzle = {'T': perm[0], 'W': perm[1], 'O': perm[2], 'F': perm[3], 'U': perm[4], 'R': perm[5]}**

**if is\_valid(puzzle):**

**print("Solution found:")**

**print(f" T W O\n+ T W O\n--------\n F O U R\n")**

**print(f" {puzzle['T']} {puzzle['W']} {puzzle['O']}")**

**print(f"+ {puzzle['T']} {puzzle['W']} {puzzle['O']}")**

**print("--------")**

**print(f" {puzzle['F']} {puzzle['O']} {puzzle['U']} {puzzle['R']}")**

**return**

**print("No solution found.")**

**if \_\_name\_\_ == "\_\_main\_\_":**

**solve\_cryptarithmetic()**

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

**Ans-**

**import random**

**def get\_response(user\_input):**

**user\_input = user\_input.lower()**

**if "hello" in user\_input or "hi" in user\_input:**

**return "Hello! How can I help you today?"**

**elif "your name" in user\_input:**

**return "I'm a simple chatbot. You can call me ChatBot."**

**elif "how are you" in user\_input:**

**return "I'm just a computer program, but thanks for asking!"**

**elif "bye" in user\_input or "goodbye" in user\_input:**

**return "Goodbye! Have a great day!"**

**else:**

**return "I'm sorry, I didn't understand that. Can you please rephrase?"**

**def main():**

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

**while True:**

**user\_input = input("You: ")**

**if user\_input.lower() == 'bye':**

**print("Simple Chatbot: Goodbye! Have a great day!")**

**break**

**response = get\_response(user\_input)**

**print("Simple Chatbot:", response)**

**if \_\_name\_\_ == "\_\_main\_\_":**

**main()**