# Slip 1

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

Ans:-

import random

def hill\_climbing(function, initial\_value, step\_size, iterations): current\_value = initial\_value

for \_ in range(iterations):

neighbors = [current\_value - step\_size, current\_value, current\_value + step\_size] neighbors = [val for val in neighbors if val >= 0] # Ensure x is non-negative current\_value = max(neighbors, key=function, default=current\_value)

return current\_value, function(current\_value) # Define the function

my\_function = lambda x: -x\*\*2 + 4\*x

# Set initial parameters

initial\_x = random.uniform(0, 10) step\_size = 0.1

iterations = 100

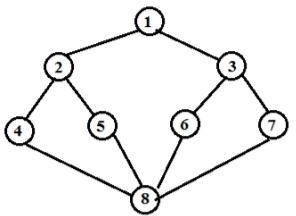
# Run hill climbing algorithm

result\_x, result\_value = hill\_climbing(my\_function, initial\_x, step\_size, iterations)

# Print results

print("Maximum found at x =", result\_x) print("Maximum value =", result\_value)

**Q2:-Write a Python program to implement Depth First Search algorithm. Refer the following graph as an Input for the program. [Initial node=1,Goal node=8]**



**Ans:-**

graph = {

'1': ['2', '3'],

'2': ['1', '4', '5'],

'3': ['1', '6','7'],

'4': ['2','8'],

'5': ['2', '8'],

'6': ['3', '8'],

'7': ['3', '8'],

'8': ['4', '5','6,'7']

}

# DFS traversal function def dfs(graph, start, visited):

if start not in visited: print(start, end=' ') visited.add(start)

for neighbor in graph[start]: dfs(graph, neighbor, visited)

# Main function to initiate DFS traversal def main():

start\_node = '1' # You can change the starting node here print("Depth-First Search Traversal:")

visited = set()

dfs(graph, start\_node, visited)

if name == ' main ': main()

# Slip2

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

**Ans:-**

import calendar

def generate\_calendar(year, month):

cal = calendar.monthcalendar(year, month) month\_name = calendar.month\_name[month]

print(f"Calendar for {month\_name} {year}:") # Print weekday names

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

# Print each week for week in cal:

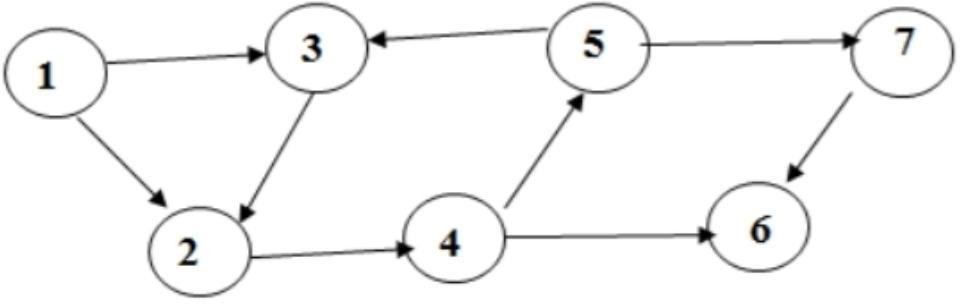
week\_str = ' '.join(str(day) if day != 0 else ' ' for day in week) print(week\_str)

# Input: Year and Month

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

month = int(input("Enter the month (1-12): ")) generate\_calendar(year, month)

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



**Ans:-**

graph = {

'1': ['2', '3'],

'2': ['4'],

'3': ['2'],

'4': ['5','6'],

'5': ['3','7'],

'7': ['6']

}

# DFS traversal function def dfs(graph, start, visited):

if start not in visited: print(start, end=' ') visited.add(start)

for neighbor in graph[start]: dfs(graph, neighbor, visited)

# Main function to initiate DFS traversal def main():

start\_node = '1' # You can change the starting node here print("Depth-First Search Traversal:")

visited = set()

dfs(graph, start\_node, visited)

if name == ' main ': main()

# Slip 3

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

import string

def remove\_punctuation(input\_string):

# Obtain the set of punctuation characters punctuation\_set = set(string.punctuation)

# Remove punctuation from the input string

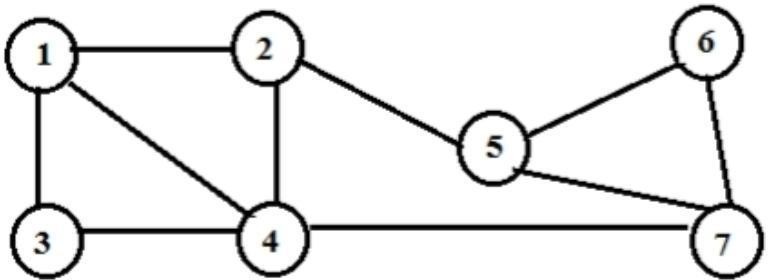
result\_string = ''.join(char for char in input\_string if char not in punctuation\_set) return result\_string

# Example usage

input\_string = "Hello, World! This is an example string with punctuations!!!" result = remove\_punctuation(input\_string)

print("Original String:", input\_string) print("String without Punctuation:", result)

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



**Ans:-**

graph = {

'1': ['2', '3','4'],

'2': ['1', '4', '5'],

'3': ['1', '4'],

'4': ['1','2','3'],

'5': ['2', '6','7,],

'6': ['5', '7'],

'7': ['4', '5','6']

}

# DFS traversal function def dfs(graph, start, visited):

if start not in visited: print(start, end=' ') visited.add(start)

for neighbor in graph[start]: dfs(graph, neighbor, visited)

# Main function to initiate DFS traversal def main():

start\_node = '2' # You can change the starting node here print("Depth-First Search Traversal:")

visited = set()

dfs(graph, start\_node, visited)

if name == ' main ': main()

# Slip 4

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

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

**Ans:-**

import random

def choose\_word():

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

def display\_word(word, guessed):

return ' '.join(letter if letter in guessed else '\_' for letter in word)

def hangman():

word, guessed, attempts = choose\_word(), set(), 6 print("Welcome to Hangman!")

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

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

if guess not in word:

attempts -= 1

print(f"Wrong guess! Attempts left: {attempts}") elif all(letter in guessed for letter in word):

print(f"Congratulations! You've guessed the word: {word}") break

else:

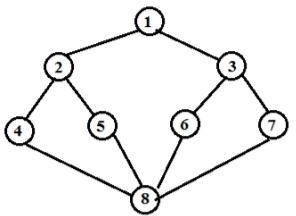
print("Please enter a single alphabet.")

if attempts == 0:

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

if name == " main ": hangman()

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



**Ans:-**

from collections import deque

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

'1': ['2', '3'],

'2': ['1', '4', '5'],

'3': ['1', '6','7'],

'4': ['2','8'],

'5': ['2', '8'],

'6': ['3', '8'],

'7': ['3', '8'],

'8': ['4', '5','6,'7']

}

# BFS traversal function def bfs(graph, start):

visited = set() # To keep track of visited nodes

queue = deque() # Create a queue for BFS

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

while queue:

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

for neighbor in graph[node]: if neighbor not in visited:

visited.add(neighbor) queue.append(neighbor)

# Main function to initiate BFS traversal def main():

start\_node = '1' # You can change the starting node here print("Breadth-First Search Traversal:")

bfs(graph, start\_node)

if name == ' main ': main()

# Slip 5

* 1. **Write a python program to implement Lemmatization using NLTK Ans:-**

import nltk

from nltk.corpus import wordnet

from nltk.stem import WordNetLemmatizer from nltk.tokenize import word\_tokenize

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

def lemmatize\_text(text):

lemmatizer = WordNetLemmatizer() tokens = word\_tokenize(text)

lemmatized\_tokens = [lemmatizer.lemmatize(token, get\_pos\_tag(token)) for token in tokens]

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

def get\_pos\_tag(word):

tag = nltk.pos\_tag([word])[0][1][0].upper()

tag\_dict = {"J": wordnet.ADJ, "N": wordnet.NOUN, "V": wordnet.VERB, "R": wordnet.ADV}

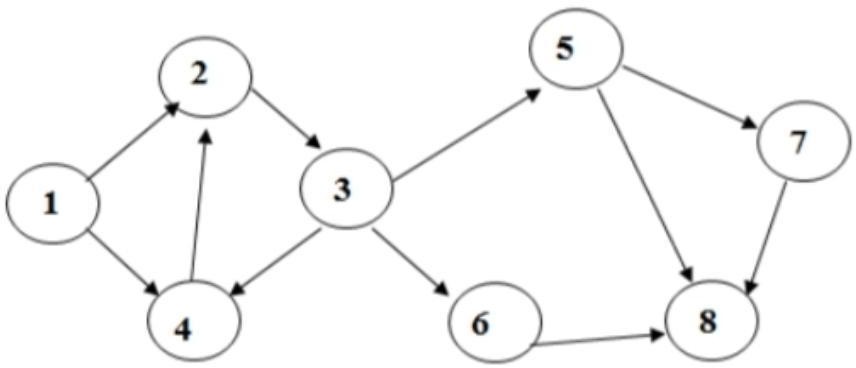
return tag\_dict.get(tag, wordnet.NOUN)

# Example usage

input\_text = "The cats are running and playing in the garden" lemmatized\_text = lemmatize\_text(input\_text)

print("Original Text:", input\_text) print("Lemmatized Text:", lemmatized\_text)

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



**Ans:**

from collections import deque

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

'1': ['2', '4'],

'2': ['3'],

'3': ['5', '6'],

'4': ['2'],

'5': ['7', '8'],

'6': ['8'],

'7': ['8']

}

# BFS traversal function def bfs(graph, start):

visited = set() # To keep track of visited nodes queue = deque() # Create a queue for BFS

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

while queue:

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

for neighbor in graph[node]: if neighbor not in visited:

visited.add(neighbor) queue.append(neighbor)

# Main function to initiate BFS traversal def main():

start\_node = '1' # You can change the starting node here print("Breadth-First Search Traversal:")

bfs(graph, start\_node)

if name == ' main ': main()

# Slip 6

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

**Ans:-**

import nltk

from nltk.corpus import stopwords

from nltk.tokenize import word\_tokenize

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

def remove\_stop\_words(file\_path): with open(file\_path, 'r') as file:

passage = file.read()

stop\_words = set(stopwords.words('english')) tokens = word\_tokenize(passage)

filtered\_tokens = [word for word in tokens if word.lower() not in stop\_words] filtered\_text = ' '.join(filtered\_tokens)

return filtered\_text # Example usage

file\_path = 'path/to/your/textfile.txt' # Replace with the actual path to your text file

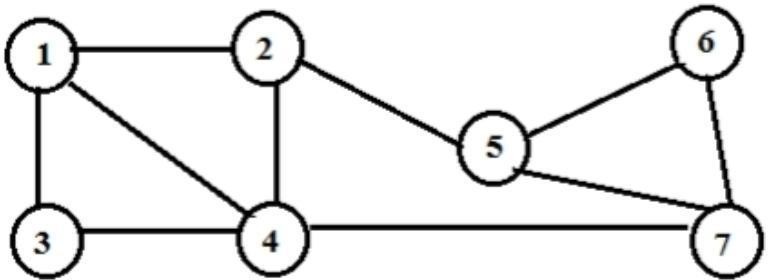
result\_text = remove\_stop\_words(file\_path)

print("Original Passage:") with open(file\_path, 'r') as file:

print(file.read())

print("\nPassage after removing stop words:") print(result\_text)

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



**Ans:-**

from collections import deque

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

'1': ['2', '3','4'],

'2': ['1', '4', '5'],

'3': ['1', '4'],

'4': ['1','2','3'],

'5': ['2', '6','7,],

'6': ['5', '7'],

'7': ['4', '5','6']

}

# BFS traversal function def bfs(graph, start):

visited = set() # To keep track of visited nodes queue = deque() # Create a queue for BFS

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

while queue:

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

for neighbor in graph[node]: if neighbor not in visited:

visited.add(neighbor) queue.append(neighbor)

# Main function to initiate BFS traversal def main():

start\_node = 'A' # You can change the starting node here print("Breadth-First Search Traversal:")

bfs(graph, start\_node)

if name == ' main ': main()

# Slip 7

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

**Ans:-**

def print\_board(board): for row in board:

print(" | ".join(row))

print("-" \* 9)

# Function to check if a player has won def check\_win(board, player):

for i in range(3):

if all(board[i][j] == player for j in range(3)): # Check rows return True

if all(board[j][i] == player for j in range(3)): # Check columns return True

if all(board[i][i] == player for i in range(3)) or all(board[i][2 - i] == player for i in range(3)): # Check diagonals

return True return False

# Function to check if the board is full (a draw) def check\_draw(board):

return all(cell != " " for row in board for cell in row)

# Main function to play the Tic-Tac-Toe game def main():

board = [[" " for \_ in range(3)] for \_ in range(3)] player = "X"

win = False

print("Tic-Tac-Toe Game:") print\_board(board)

while not win and not check\_draw(board):

print(f"Player {player}, enter your move (row and column):") row, col = map(int, input().split())

if 1 <= row <= 3 and 1 <= col <= 3 and board[row - 1][col - 1] == " ": board[row - 1][col - 1] = player

win = check\_win(board, player)

player = "O" if player == "X" else "X" print\_board(board)

else:

print("Invalid move. Try again.")

if win:

print(f"Player {player} wins!") else:

print("It's a draw!")

if name == " main ": main()

* 1. **Write a Python program to implement Simple Chatbot. Ans:-**

responses = {

"hi": "Hello there! How can I help you today?", "hello": "Hi! How can I assist you?",

"hey": "Hey! What can I do for you?",

"how are you": "I'm just a computer program, but I'm here to help you.", "bye": "Goodbye! Have a great day.",

"exit": "Goodbye! If you have more questions, feel free to come back."

}

# Chatbot function

def chatbot(user\_input):

user\_input = user\_input.lower() # Convert the input to lowercase for case-insensitive matching

response = responses.get(user\_input, "I'm not sure how to respond to that. Please choose from the predefined inputs. 'hi', 'hello', 'hey', 'how are you', 'bye', 'exit'")

return response

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

user\_input = input("You: ")

if user\_input.lower() == "bye" or user\_input.lower() == "exit": print("Simple Chatbot: Goodbye!")

break

response = chatbot(user\_input) print("Simple Chatbot:", response)

# Slip 8

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

**Ans:-**

def count\_upper\_lower(input\_string):

upper\_count = sum(1 for char in input\_string if char.isupper()) lower\_count = sum(1 for char in input\_string if char.islower()) return upper\_count, lower\_count

# Example usage

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

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

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

* 1. **Write a Python program to solve tic-tac-toe problem. Ans:-**

def print\_board(board): for row in board:

print(" | ".join(row))

print("-" \* 9)

# Function to check if a player has won def check\_win(board, player):

for i in range(3):

if all(board[i][j] == player for j in range(3)): # Check rows return True

if all(board[j][i] == player for j in range(3)): # Check columns return True

if all(board[i][i] == player for i in range(3)) or all(board[i][2 - i] == player for i in range(3)): # Check diagonals

return True return False

# Function to check if the board is full (a draw) def check\_draw(board):

return all(cell != " " for row in board for cell in row)

# Main function to play the Tic-Tac-Toe game def main():

board = [[" " for \_ in range(3)] for \_ in range(3)] player = "X"

win = False

print("Tic-Tac-Toe Game:") print\_board(board)

while not win and not check\_draw(board):

print(f"Player {player}, enter your move (row and column):") row, col = map(int, input().split())

if 1 <= row <= 3 and 1 <= col <= 3 and board[row - 1][col - 1] == " ": board[row - 1][col - 1] = player

win = check\_win(board, player) player = "O" if player == "X" else "X" print\_board(board)

else:

print("Invalid move. Try again.")

if win:

print(f"Player {player} wins!") else:

print("It's a draw!")

if name == " main ": main()

# Slip 9

* 1. **Write python program to solve 8 puzzle problem using A algorithm [10 marks]**

**Ans:-**

import heapq class PuzzleNode:

def init (self, state, parent=None, move=None, cost=0, heuristic=0):

self.state, self.parent, self.move, self.cost, self.heuristic = state, parent, move, cost, heuristic

def lt (self, other):

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

def manhattan\_distance(state):

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

def neighbors(node):

zero\_i = node.state.index(0) moves = [1, -1, 3, -3]

return [PuzzleNode(list(node.state[:zero\_i] + [node.state[zero\_i + m]] + node.state[zero\_i + m + 1:]), node, m, node.cost + 1, manhattan\_distance(node.state)) for m in moves if 0 <= zero\_i + m < 9 and (m == 1 or m == -1 or m == 3 or m == -3)]

def print\_solution(node): moves = []

while node.parent:

moves.append("Move right" if node.move == 1 else "Move left" if node.move == -1 else "Move down" if node.move == 3 else "Move up")

node = node.parent moves.reverse()

print("Solution found!\n" + '\n'.join(moves))

def solve\_puzzle(initial\_state):

goal, frontier, explored = [1, 2, 3, 4, 5, 6, 7, 8, 0], [PuzzleNode(initial\_state, None, None, 0, manhattan\_distance(initial\_state))], set()

while frontier:

current\_node = heapq.heappop(frontier)

if current\_node.state == goal: print\_solution(current\_node) return

explored.add(tuple(current\_node.state))

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

print("No solution found.") if name == " main ":

# Example usage:

solve\_puzzle([1, 2, 3, 4, 5, 6, 0, 7, 8]) # Replace with your initial state

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

**Ans:-**

def water\_jug\_problem(capacity\_x, capacity\_y, target): jug\_x = 0

jug\_y = 0

while jug\_x != target and jug\_y != target: print(f"Jug X: {jug\_x}L, Jug Y: {jug\_y}L")

# Fill jug X if it is empty if jug\_x == 0:

jug\_x = capacity\_x print("Fill Jug X")

# Transfer water from jug X to jug Y if jug X is not empty elif jug\_x > 0 and jug\_y < capacity\_y:

transfer = min(jug\_x, capacity\_y - jug\_y) jug\_x -= transfer

jug\_y += transfer

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

# Empty jug Y if it is full elif jug\_y == capacity\_y:

jug\_y = 0 print("Empty Jug Y")

print(f"Jug X: {jug\_x}L, Jug Y: {jug\_y}L") print("Solution Found!")

# Main function to initiate the problem def main():

capacity\_x = 5 # Capacity of jug X capacity\_y = 7 # Capacity of jug Y

target = 4 # Amount of water to measure

print("Solving Water Jug Problem:") water\_jug\_problem(capacity\_x, capacity\_y, target)

if name == ' main ': main()

# Slip 10

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

**Ans:-**

from itertools import permutations

def is\_valid\_assignment(mapping, word): return int(''.join(mapping[ch] for ch in word))

def solve\_cryptarithmetic\_puzzle(): puzzle = ["TWO", "TWO", "FOUR"]

unique\_chars = set(''.join(puzzle)) if len(unique\_chars) > 10:

print("Invalid puzzle: More than 10 unique characters.") return

for perm in permutations("0123456789", len(unique\_chars)): mapping = dict(zip(unique\_chars, perm))

if mapping[puzzle[0][0]] != '0' and is\_valid\_assignment(mapping, puzzle[0]) + is\_valid\_assignment(mapping, puzzle[1]) == is\_valid\_assignment(mapping, puzzle[2]):

print("Solution found:") for word in puzzle:

print(f"{word}: {is\_valid\_assignment(mapping, word)}") return

print("No solution found.") if name == " main ":

solve\_cryptarithmetic\_puzzle()

* 1. **Write a Python program to implement Simple Chatbot. Ans:-**

responses = {

"hi": "Hello there! How can I help you today?", "hello": "Hi! How can I assist you?",

"hey": "Hey! What can I do for you?",

"how are you": "I'm just a computer program, but I'm here to help you.", "bye": "Goodbye! Have a great day.",

"exit": "Goodbye! If you have more questions, feel free to come back."

}

# Chatbot function

def chatbot(user\_input):

user\_input = user\_input.lower() # Convert the input to lowercase for case-insensitive matching

response = responses.get(user\_input, "I'm not sure how to respond to that. Please choose from the predefined inputs. 'hi', 'hello', 'hey', 'how are you', 'bye', 'exit'")

return response

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

user\_input = input("You: ")

if user\_input.lower() == "bye" or user\_input.lower() == "exit": print("Simple Chatbot: Goodbye!")

break

response = chatbot(user\_input) print("Simple Chatbot:", response)

# Slip 11

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

**Ans:-**

def mean\_end\_analysis(initial, target): if len(initial) != len(target):

print("Strings must have the same length.") return

operations = []

for i in range(len(initial)): if initial[i] != target[i]:

operations.append(f"Change '{initial[i]}' to '{target[i]}' at position {i + 1}")

if not operations:

print("Strings are already the same.") else:

print("Transformation Steps:") for operation in operations:

print(operation)

if name == " main ":

initial\_string = input("Enter the initial string: ").lower() target\_string = input("Enter the target string: ").lower()

mean\_end\_analysis(initial\_string, target\_string)

* 1. **Write a Python program to solve water jug problem. Two jugs with capacity 4 gallon and 3 gallon are given with unlimited water supply respectively. The target is to achieve 2 gallon of water in second jug.**

**Ans:-**

def water\_jug\_problem(capacity\_x, capacity\_y, target): jug\_x = 0

jug\_y = 0

while jug\_x != target and jug\_y != target: print(f"Jug X: {jug\_x}L, Jug Y: {jug\_y}L")

# Fill jug X if it is empty if jug\_x == 0:

jug\_x = capacity\_x print("Fill Jug X")

# Transfer water from jug X to jug Y if jug X is not empty elif jug\_x > 0 and jug\_y < capacity\_y:

transfer = min(jug\_x, capacity\_y - jug\_y) jug\_x -= transfer

jug\_y += transfer

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

# Empty jug Y if it is full elif jug\_y == capacity\_y:

jug\_y = 0 print("Empty Jug Y")

print(f"Jug X: {jug\_x}L, Jug Y: {jug\_y}L") print("Solution Found!")

# Main function to initiate the problem def main():

capacity\_x = 4 # Capacity of jug X capacity\_y = 3 # Capacity of jug Y

target = 2 # Amount of water to measure

print("Solving Water Jug Problem:") water\_jug\_problem(capacity\_x, capacity\_y, target)

if name == ' main ': main()

# Slip 12

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

import calendar

def generate\_calendar(year, month):

cal = calendar.monthcalendar(year, month) month\_name = calendar.month\_name[month]

print(f"Calendar for {month\_name} {year}:") # Print weekday names

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

# Print each week for week in cal:

week\_str = ' '.join(str(day) if day != 0 else ' ' for day in week) print(week\_str)

# Input: Year and Month

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

month = int(input("Enter the month (1-12): ")) generate\_calendar(year, month)

* 1. **Write a Python program to simulate 4-Queens problem. Ans:-**

def print\_chessboard(chessboard): for row in chessboard:

print(" ".join(row))

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

def is\_safe(chessboard, row, col, n): # Check row on the left side

for i in range(col):

if chessboard[row][i] == 'Q': return False

# Check upper diagonal on the left side

for i, j in zip(range(row, -1, -1), range(col, -1, -1)): if chessboard[i][j] == 'Q':

return False

# Check lower diagonal on the left side

for i, j in zip(range(row, n, 1), range(col, -1, -1)): if chessboard[i][j] == 'Q':

return False return True

# Recursive function to solve the Four Queens problem def solve\_four\_queens(chessboard, col, n):

if col >= n:

return True # All queens are placed

for i in range(n):

if is\_safe(chessboard, i, col, n): chessboard[i][col] = 'Q' # Place a queen

# Recur to place the rest of the queens

if solve\_four\_queens(chessboard, col + 1, n): return True

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

return False # No solution exists

# Main function to solve the Four Queens problem def main():

n = 4 # Size of the chessboard (8x8)

chessboard = [['.' for \_ in range(n)] for \_ in range(n)]

if solve\_four\_queens(chessboard, 0, n): print("Solution to the Four Queens Problem:") print\_chessboard(chessboard)

else:

print("No solution found.")

if name == ' main ': main()

# Slip 13

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

import math

def evaluate(board):

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

def is\_terminal(board):

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

def get\_available\_moves(board):

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

def mini\_max(board, depth, maximizing\_player): if is\_terminal(board):

return evaluate(board)

return max(mini\_max(make\_move(board, move, 'X'), depth + 1, False) if maximizing\_player else mini\_max(make\_move(board, move, 'O'), depth + 1, True) for move in get\_available\_moves(board))

def find\_best\_move(board):

return max(get\_available\_moves(board), key=lambda move: mini\_max(make\_move(board, move, 'X'), 0, False))

def make\_move(board, move, player):

i, j = move

new\_board = [row.copy() for row in board] new\_board[i][j] = player

return new\_board

def print\_board(board): for row in board:

print(" ".join(cell for cell in row)) print()

def play\_game():

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

print("Initial Board:") print\_board(board)

for \_ in range(4): # Play four moves for demonstration

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

board = make\_move(board, player\_move, 'O')

print("Updated Board after your move:") print\_board(board)

if is\_terminal(board): print("Game over!") break

print("Computer's move:") computer\_move = find\_best\_move(board)

board = make\_move(board, computer\_move, 'X')

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

if is\_terminal(board): print("Game over!") break

if name == " main ":

play\_game()

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

def print\_chessboard(chessboard): for row in chessboard:

print(" ".join(row))

# Function to check if it's safe to place a queen at the given position def is\_safe(chessboard, row, col, n):

# Check row on the left side for i in range(col):

if chessboard[row][i] == 'Q': return False

# Check upper diagonal on the left side

for i, j in zip(range(row, -1, -1), range(col, -1, -1)): if chessboard[i][j] == 'Q':

return False

# Check lower diagonal on the left side

for i, j in zip(range(row, n, 1), range(col, -1, -1)): if chessboard[i][j] == 'Q':

return False return True

# Recursive function to solve the Eight Queens problem def solve\_eight\_queens(chessboard, col, n):

if col >= n:

return True # All queens are placed

for i in range(n):

if is\_safe(chessboard, i, col, n): chessboard[i][col] = 'Q' # Place a queen

# Recur to place the rest of the queens

if solve\_eight\_queens(chessboard, col + 1, n):

return True

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

return False # No solution exists

# Main function to solve the Eight Queens problem def main():

n = 8 # Size of the chessboard (8x8)

chessboard = [['.' for \_ in range(n)] for \_ in range(n)]

if solve\_eight\_queens(chessboard, 0, n): print("Solution to the Eight Queens Problem:") print\_chessboard(chessboard)

else:

print("No solution found.")

if name == ' main ': main()

# Slip 14

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

def sort\_sentence(sentence): words = sentence.split() sorted\_words = sorted(words)

sorted\_sentence = ' '.join(sorted\_words) return sorted\_sentence

if name == " main ":

input\_sentence = input("Enter a sentence: ") result = sort\_sentence(input\_sentence) print("Sorted Sentence:", result)

* 1. **Write a Python program to simulate n-Queens problem.**

**Ans:-**

def print\_chessboard(chessboard): for row in chessboard:

print(" ".join(row))

# Function to check if it's safe to place a queen at the given position def is\_safe(chessboard, row, col, n):

# Check the column for i in range(row):

if chessboard[i][col] == 'Q': return False

# Check the upper left diagonal

for i, j in zip(range(row, -1, -1), range(col, -1, -1)): if chessboard[i][j] == 'Q':

return False

# Check the upper right diagonal

for i, j in zip(range(row, -1, -1), range(col, n)): if chessboard[i][j] == 'Q':

return False return True

# Recursive function to solve the N-Queens problem using forward checking def solve\_nqueens(chessboard, row, n):

if row >= n:

return True # All queens are placed

for col in range(n):

if is\_safe(chessboard, row, col, n): chessboard[row][col] = 'Q' # Place a queen

if solve\_nqueens(chessboard, row + 1, n): return True

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

return False # No solution exists

# Main function to perform N-Queens puzzle with forward checking def main():

n = 8 # Size of the chessboard (8x8)

chessboard = [['.' for \_ in range(n)] for \_ in range(n)]

print("N-Queens Puzzle using Forward Checking:") if solve\_nqueens(chessboard, 0, n):

print("\nSolution to the N-Queens Puzzle:") print\_chessboard(chessboard)

else:

print("No solution found.")

if name == ' main ': main()

# Slip 15

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

import queue class State:

def init (self, monkey\_row, monkey\_col, has\_banana):

self.monkey\_row = monkey\_row self.monkey\_col = monkey\_col self.has\_banana = has\_banana

def is\_valid(state, rows, cols):

return 0 <= state.monkey\_row < rows and 0 <= state.monkey\_col < cols

def is\_goal(state, banana\_row, banana\_col):

return state.monkey\_row == banana\_row and state.monkey\_col == banana\_col and state.has\_banana

def move(state, action):

new\_state = State(state.monkey\_row, state.monkey\_col, state.has\_banana)

if action == 'UP': new\_state.monkey\_row -= 1

elif action == 'DOWN': new\_state.monkey\_row += 1

elif action == 'LEFT': new\_state.monkey\_col -= 1

elif action == 'RIGHT': new\_state.monkey\_col += 1

elif action == 'GRAB': new\_state.has\_banana = True

return new\_state

def bfs(start\_state, banana\_row, banana\_col, rows, cols): frontier = queue.Queue()

frontier.put((start\_state, []))

while not frontier.empty(): current\_state, path = frontier.get()

if is\_goal(current\_state, banana\_row, banana\_col): return path

for action in ['UP', 'DOWN', 'LEFT', 'RIGHT', 'GRAB']:

new\_state = move(current\_state, action)

if is\_valid(new\_state, rows, cols): new\_path = path + [action] frontier.put((new\_state, new\_path))

return None

def print\_solution(path): if path is None:

print("No solution found.") else:

print("Solution:") print(" -> ".join(path))

if name == " main ": rows = 4

cols = 4

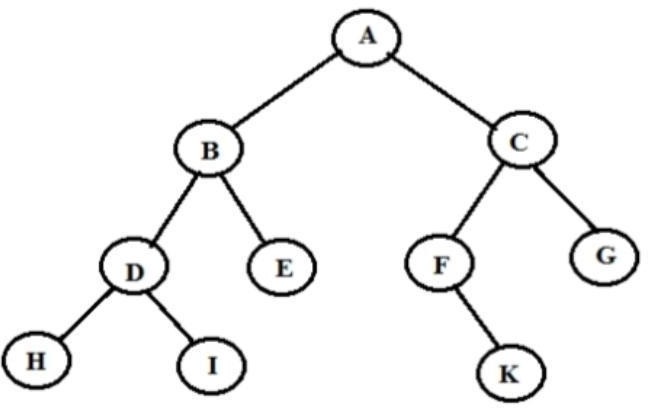
monkey\_start = (3, 0)

banana\_location = (0, 3)

start\_state = State(monkey\_start[0], monkey\_start[1], False)

solution\_path = bfs(start\_state, banana\_location[0], banana\_location[1], rows, cols) print\_solution(solution\_path)

* 1. **Write a program to implement Iterative Deepening DFS algorithm. [Goal Node=G]**



**Ans:-**

class Node:

def init (self, state, children=None): self.state = state

self.children = children if children else []

def depth\_limited\_dfs(node, goal\_state, depth\_limit, current\_depth=0): if current\_depth > depth\_limit:

return None

if node.state == goal\_state: return [node.state]

for child in node.children:

path = depth\_limited\_dfs(child, goal\_state, depth\_limit, current\_depth + 1) if path is not None:

return [node.state] + path

return None

def iterative\_deepening\_dfs(root, goal\_state): depth\_limit = 0

while True:

result = depth\_limited\_dfs(root, goal\_state, depth\_limit) if result is not None:

return result depth\_limit += 1

if name == " main ": # Example usage:

# Creating a simple tree structure for demonstration

root = Node("A", [Node("B", [Node("D", [Node("G")])]), Node("C", [Node("E"), Node("F", [Node("H", [Node("I")])])])])

goal\_node = "G"

solution\_path = iterative\_deepening\_dfs(root, goal\_node)

if solution\_path:

print("Solution Path:", " -> ".join(solution\_path)) else:

print("No solution found.")

# Slip 16

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

def tower\_of\_hanoi(n, source\_peg, target\_peg, auxiliary\_peg): if n == 1:

print(f"Move disk 1 from {source\_peg} to {target\_peg}") return

tower\_of\_hanoi(n - 1, source\_peg, auxiliary\_peg, target\_peg) print(f"Move disk {n} from {source\_peg} to {target\_peg}") tower\_of\_hanoi(n - 1, auxiliary\_peg, target\_peg, source\_peg)

if name == " main ":

number\_of\_disks = int(input("Enter the number of disks: "))

tower\_of\_hanoi(number\_of\_disks, 'A', 'C', 'B')

* 1. **Write a Python program to solve tic-tac-toe problem. Ans:-**

def print\_board(board): for row in board:

print(" | ".join(row))

print("-" \* 9)

# Function to check if a player has won def check\_win(board, player):

for i in range(3):

if all(board[i][j] == player for j in range(3)): # Check rows return True

if all(board[j][i] == player for j in range(3)): # Check columns return True

if all(board[i][i] == player for i in range(3)) or all(board[i][2 - i] == player for i in range(3)): # Check diagonals

return True return False

# Function to check if the board is full (a draw) def check\_draw(board):

return all(cell != " " for row in board for cell in row)

# Main function to play the Tic-Tac-Toe game def main():

board = [[" " for \_ in range(3)] for \_ in range(3)] player = "X"

win = False

print("Tic-Tac-Toe Game:") print\_board(board)

while not win and not check\_draw(board):

print(f"Player {player}, enter your move (row and column):") row, col = map(int, input().split())

if 1 <= row <= 3 and 1 <= col <= 3 and board[row - 1][col - 1] == " ": board[row - 1][col - 1] = player

win = check\_win(board, player) player = "O" if player == "X" else "X" print\_board(board)

else:

print("Invalid move. Try again.")

if win:

print(f"Player {player} wins!") else:

print("It's a draw!")

if name == " main ": main()

# Slip 17

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

**Ans:-**

def hill\_climbing(function, initial\_guess, step\_size, max\_iterations): current\_solution = initial\_guess

current\_value = function(current\_solution)

for \_ in range(max\_iterations):

neighbor = current\_solution + step\_size neighbor\_value = function(neighbor)

if neighbor\_value > current\_value: current\_solution = neighbor current\_value = neighbor\_value

else:

break

return current\_solution, current\_value

# Example mathematical function (you can replace this with your own function) def example\_function(x):

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

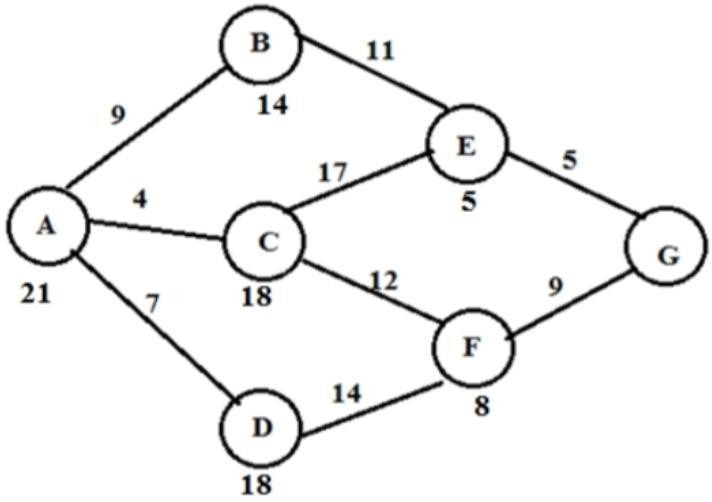
if name == " main ": # Example usage:

initial\_guess = 0 # Initial guess for the maximum step\_size = 0.1 # Step size for climbing max\_iterations = 100 # Maximum number of iterations

result\_solution, result\_value = hill\_climbing(example\_function, initial\_guess, step\_size, max\_iterations)

print(f"Maximum Solution: {result\_solution}") print(f"Maximum Value: {result\_value}")

* 1. **Write a Python program to implement A\* algorithm. Refer the following graph as an Input for the program. Start vertex is A and Goal Vertex is G]**



**Ans:-**

import heapq

# Graph represented as an adjacency list graph = {

'A': {'B': 9, 'C': 4, 'D':7},

'B': {'A': 9, 'E': 11},

'C': {'A': 4, 'E': 17, 'F': 12},

'D': {'A': 7, 'F': 14},

'E': {'B': 11, 'C': 17, 'G': 5},

'F': {'C': 12, 'D': 14,'G': 9},

'G': {'E': 5, 'F': 9}

}

# Heuristic function (replace with your own heuristic) heuristic = {

'A': 21,

'B': 14,

'C': 18,

'D': 18,

'E': 5,

'F': 8,

'G': 0

}

def astar(start, goal): priority\_queue = [(0, start)] visited = set()

while priority\_queue:

current\_cost, current\_node = heapq.heappop(priority\_queue)

if current\_node == goal: return current\_cost

if current\_node not in visited: visited.add(current\_node)

for neighbor, edge\_cost in graph[current\_node].items(): heuristic\_cost = heuristic[neighbor]

total\_cost = current\_cost + edge\_cost + heuristic\_cost heapq.heappush(priority\_queue, (total\_cost, neighbor))

return float('inf') # No path found if name == " main ":

start\_vertex = 'A'

goal\_vertex = 'G'

result\_cost = astar(start\_vertex, goal\_vertex) if result\_cost != float('inf'):

print(f"Cost from {start\_vertex} to {goal\_vertex} using A\* algorithm: {result\_cost}") else:

print(f"No path found from {start\_vertex} to {goal\_vertex}.")

# Slip 18

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

**Ans:-**

import nltk

from nltk.corpus import stopwords

from nltk.tokenize import word\_tokenize

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

def remove\_stop\_words(input\_text): stop\_words = set(stopwords.words('english')) words = word\_tokenize(input\_text)

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

if name == " main ":

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

try:

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

cleaned\_passage = remove\_stop\_words(passage) print("Original Passage:")

print(passage)

print("\nPassage after removing stop words:") print(cleaned\_passage)

except FileNotFoundError:

print(f"File not found at path: {file\_path}") except Exception as e:

print(f"An error occurred: {e}")

**Q.2) Implement a system that performs arrangement of some set of objects in a room. Assume that you have only 5 rectangular, 4 square-shaped objects. Use A approach for the placement of the objects in room for efficient space utilisation.**

**Assume suitable heuristic, and dimensions of objects and rooms. (Informed Search)**

**Ans:-**

import heapq class State:

def init (self, room\_width, room\_height, remaining\_objects, current\_state=None): self.room\_width = room\_width

self.room\_height = room\_height self.remaining\_objects = remaining\_objects

if current\_state:

self.placed\_objects = current\_state.placed\_objects.copy() self.total\_wasted\_space = current\_state.total\_wasted\_space

else:

self.placed\_objects = [] self.total\_wasted\_space = 0

def is\_goal(self):

return not self.remaining\_objects

def heuristic(self):

# Simple heuristic: Minimize wasted space return self.total\_wasted\_space

def lt (self, other):

return (self.total\_wasted\_space + self.heuristic()) < (other.total\_wasted\_space + other.heuristic())

def a\_star(room\_width, room\_height, object\_dimensions): initial\_state = State(room\_width, room\_height, object\_dimensions) priority\_queue = [initial\_state]

while priority\_queue:

current\_state = heapq.heappop(priority\_queue)

if current\_state.is\_goal(): return current\_state

for obj\_width, obj\_height in current\_state.remaining\_objects:

new\_state = State(room\_width, room\_height, current\_state.remaining\_objects, current\_state)

if room\_width - obj\_width >= 0 and room\_height - obj\_height >= 0: new\_state.placed\_objects.append((obj\_width, obj\_height)) new\_state.remaining\_objects.remove((obj\_width, obj\_height)) new\_state.total\_wasted\_space += room\_width \* room\_height - obj\_width \*

obj\_height

heapq.heappush(priority\_queue, new\_state) return None

if name == " main ": room\_width = 10

room\_height = 8

object\_dimensions = [(3, 2), (2, 2), (4, 3), (1, 1), (2, 1)]

result\_state = a\_star(room\_width, room\_height, object\_dimensions) if result\_state:

print("Optimal arrangement:") print(result\_state.placed\_objects)

print("Total wasted space:", result\_state.total\_wasted\_space) else:

print("No solution found.")

# Slip 19

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

**Ans:-**

import random

def choose\_word():

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

def display\_word(word, guessed\_letters):

return ''.join(letter if letter in guessed\_letters else '\_' for letter in word)

def hangman():

word\_to\_guess = choose\_word().lower() guessed\_letters = set()

attempts\_left = 6

print("Welcome to Hangman!") print(display\_word(word\_to\_guess, guessed\_letters))

while attempts\_left > 0:

user\_guess = input("Enter a letter: ").lower()

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

if user\_guess in guessed\_letters: print("You've already guessed that letter.") continue

guessed\_letters.add(user\_guess)

if user\_guess not in word\_to\_guess: attempts\_left -= 1

print(f"Wrong guess! Attempts left: {attempts\_left}") else:

print("Correct guess!") print(display\_word(word\_to\_guess, guessed\_letters))

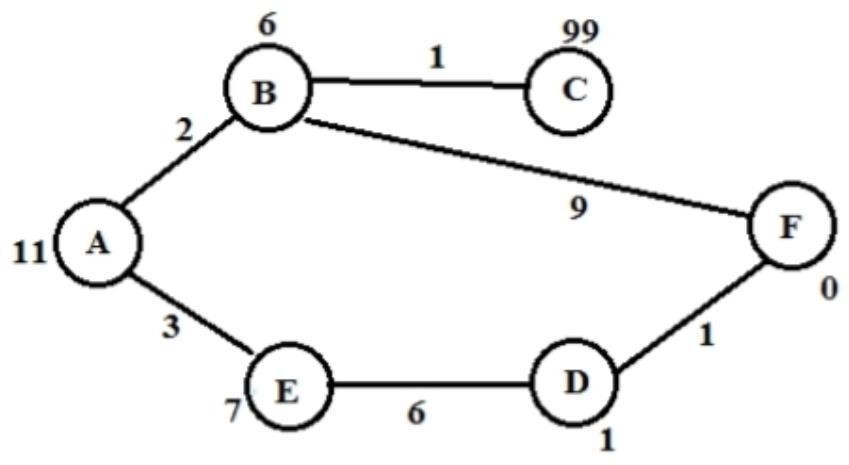
if '\_' not in display\_word(word\_to\_guess, guessed\_letters): print("Congratulations! You've guessed the word.") break

if attempts\_left == 0:

print(f"Sorry, you've run out of attempts. The word was: {word\_to\_guess}")

if name == " main ": hangman()

* 1. **Write a Python program to implement A\* algorithm. Refer the following graph as an Input for the program.**



**Ans:-**

import heapq

# Graph represented as an adjacency list graph = {

'A': {'B': 2, 'E': 3},

'B': {'A': 2, 'C': 1,'F':9}, 'C': {'B': 1},

'D': {'E': 6, 'F': 1},

'E': {'A': 3, 'D': 6},

'F': {'B': 9, 'D': 1},

}

# Heuristic function (replace with your own heuristic) heuristic = {

'A': 11,

'B': 6,

'C': 99,

'D': 1,

'E': 7,

'F': 0,

}

def astar(start, goal):

priority\_queue = [(0, start)] visited = set()

while priority\_queue:

current\_cost, current\_node = heapq.heappop(priority\_queue)

if current\_node == goal: return current\_cost

if current\_node not in visited: visited.add(current\_node)

for neighbor, edge\_cost in graph[current\_node].items(): heuristic\_cost = heuristic[neighbor]

total\_cost = current\_cost + edge\_cost + heuristic\_cost heapq.heappush(priority\_queue, (total\_cost, neighbor))

return float('inf') # No path found if name == " main ":

start\_vertex = 'A'

goal\_vertex = 'F'

result\_cost = astar(start\_vertex, goal\_vertex) if result\_cost != float('inf'):

print(f"Cost from {start\_vertex} to {goal\_vertex} using A\* algorithm: {result\_cost}") else:

print(f"No path found from {start\_vertex} to {goal\_vertex}.")

# Slip 20

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

from flask import Flask, request, jsonify app = Flask( name )

# Replace these with your own information your\_name = "Your Name"

your\_program = "Your Program" your\_year = "Your Year"

your\_interests = ["Interest 1", "Interest 2", "Interest 3"]

@app.route('/college\_bot', methods=['POST']) def college\_bot():

data = request.get\_json()

if 'action' in data:

action = data['action']

if action == 'get\_info': response = {

'name': your\_name, 'program': your\_program, 'year': your\_year, 'interests': your\_interests

}

return jsonify(response) else:

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

if name == ' main ': app.run(debug=True)

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

import math

def mini\_max(board, depth, is\_maximizing\_player): if check\_winner(board):

return evaluate(board)

if is\_maximizing\_player: max\_eval = -math.inf

for move in possible\_moves(board): board[move] = 'X'

eval = mini\_max(board, depth + 1, False) board[move] = ' ' # undo the move max\_eval = max(max\_eval, eval)

return max\_eval else:

min\_eval = math.inf

for move in possible\_moves(board): board[move] = 'O'

eval = mini\_max(board, depth + 1, True) board[move] = ' ' # undo the move min\_eval = min(min\_eval, eval)

return min\_eval

def check\_winner(board):

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

def evaluate(board):

# Evaluate the current state of the board (you need to define this based on your game)

return 0

def possible\_moves(board):

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

if name == " main ": # Example usage:

initial\_board = [' '] \* 9 # Assume a Tic-Tac-Toe board for simplicity best\_move = -1

best\_value = -math.inf

for move in possible\_moves(initial\_board): initial\_board[move] = 'X'

move\_value = mini\_max(initial\_board, 0, False) initial\_board[move] = ' ' # undo the move

if move\_value > best\_value: best\_value = move\_value best\_move = move

print(f"The best move is {best\_move} with a value of {best\_value}")

# Slip 21

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

import string

def remove\_punctuation(input\_string):

return ''.join(char for char in input\_string if char not in string.punctuation)

if name == " main ":

input\_string = "Hello, world! This is an example string."

result = remove\_punctuation(input\_string)

print("Original String:", input\_string) print("String without Punctuation:", result)

* 1. **Write a Python program for the following Cryptarithmetic problems. GO + TO**

**= OUT**

**Ans:-**

from itertools import permutations

def is\_solution(mapping):

go = mapping['G'] \* 10 + mapping['O'] to = mapping['T'] \* 10 + mapping['O']

out = mapping['O'] \* 100 + mapping['U'] \* 10 + mapping['T'] return go + to == out

def solve\_cryptarithmetic():

for p in permutations(range(10), 5):

mapping = {'G': p[0], 'O': p[1], 'T': p[2], 'U': p[3], 'N': p[4]}

if is\_solution(mapping): return mapping

return None

if name == " main ": solution = solve\_cryptarithmetic()

if solution:

print("Solution found:") print(f" G = {solution['G']}")

print(f" O = {solution['O']}")

print(f" T = {solution['T']}")

print(f" U = {solution['U']}")

print(f" N = {solution['N']}") print("\n GO")

print("+ TO") print(" ")

print(f" OUT") else:

print("No solution found.")

# Slip 22

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

import math

def alpha\_beta\_pruning(board, depth, alpha, beta, is\_maximizing\_player): if depth == 0 or game\_over(board):

return evaluate(board)

if is\_maximizing\_player: max\_eval = -math.inf

for move in possible\_moves(board): board[move] = 'X'

eval = alpha\_beta\_pruning(board, depth - 1, alpha, beta, False) board[move] = ' ' # undo the move

max\_eval = max(max\_eval, eval) alpha = max(alpha, eval)

if beta <= alpha:

break # Beta cut-off

return max\_eval else:

min\_eval = math.inf

for move in possible\_moves(board): board[move] = 'O'

eval = alpha\_beta\_pruning(board, depth - 1, alpha, beta, True) board[move] = ' ' # undo the move

min\_eval = min(min\_eval, eval) beta = min(beta, eval)

if beta <= alpha:

break # Alpha cut-off return min\_eval

def game\_over(board):

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

def evaluate(board):

# Implement your evaluation function based on the current state of the board return 0

def possible\_moves(board):

# Implement generating a list of possible moves based on the current state of the board

return []

if name == " main ": # Example usage:

initial\_board = [' '] \* 9 # Assume a Tic-Tac-Toe board for simplicity depth\_limit = 3

best\_move = -1 best\_value = -math.inf

for move in possible\_moves(initial\_board): initial\_board[move] = 'X'

move\_value = alpha\_beta\_pruning(initial\_board, depth\_limit - 1, -math.inf, math.inf, False)

initial\_board[move] = ' ' # undo the move

if move\_value > best\_value: best\_value = move\_value best\_move = move

print(f"The best move is {best\_move} with a value of {best\_value}")

* 1. **Write a Python program to implement Simple Chatbot Ans:-**

responses = {

"hi": "Hello there! How can I help you today?", "hello": "Hi! How can I assist you?",

"hey": "Hey! What can I do for you?",

"how are you": "I'm just a computer program, but I'm here to help you.", "bye": "Goodbye! Have a great day.",

"exit": "Goodbye! If you have more questions, feel free to come back."

}

# Chatbot function

def chatbot(user\_input):

user\_input = user\_input.lower() # Convert the input to lowercase for case-insensitive matching

response = responses.get(user\_input, "I'm not sure how to respond to that. Please choose from the predefined inputs. 'hi', 'hello', 'hey', 'how are you', 'bye', 'exit'")

return response

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

user\_input = input("You: ")

if user\_input.lower() == "bye" or user\_input.lower() == "exit": print("Simple Chatbot: Goodbye!")

break

response = chatbot(user\_input) print("Simple Chatbot:", response)

# Slip 23

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

**Ans:-**

def tower\_of\_hanoi(n, source, target, auxiliary): if n == 1:

print(f"Move disk 1 from {source} to {target}") return

tower\_of\_hanoi(n - 1, source, auxiliary, target) print(f"Move disk {n} from {source} to {target}") tower\_of\_hanoi(n - 1, auxiliary, target, source)

if name == " main ":

num\_discs = int(input("Enter the number of discs: ")) tower\_of\_hanoi(num\_discs, 'A', 'C', 'B')

* 1. **Write a Python program for the following Cryptarithmetic problems SEND + MORE = MONEY**

**Ans:-**

from itertools import permutations def is\_solution(mapping):

send = mapping['S'] \* 1000 + mapping['E'] \* 100 + mapping['N'] \* 10 + mapping['D'] more = mapping['M'] \* 1000 + mapping['O'] \* 100 + mapping['R'] \* 10 + mapping['E'] money = mapping['M'] \* 10000 + mapping['O'] \* 1000 + mapping['N'] \* 100 +

mapping['E'] \* 10 + mapping['Y'] return send + more == money

def solve\_cryptarithmetic():

for p in permutations(range(10), 8):

mapping = {'S': p[0], 'E': p[1], 'N': p[2], 'D': p[3], 'M': p[4], 'O': p[5], 'R': p[6], 'Y': p[7]}

if is\_solution(mapping): return mapping

return None

if name == " main ": solution = solve\_cryptarithmetic()

if solution:

print("Solution found:")

print(f" S = {solution['S']}")

print(f" E = {solution['E']}")

print(f" N = {solution['N']}")

print(f" D = {solution['D']}")

print(f" M = {solution['M']}")

print(f" O = {solution['O']}")

print(f" R = {solution['R']}")

print(f" Y = {solution['Y']}") print("\n SEND")

print("+ MORE") print(" ")

print(f" MONEY") else:

print("No solution found.")

# Slip 24

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

def sort\_sentence(sentence): words = sentence.split() sorted\_words = sorted(words)

sorted\_sentence = ' '.join(sorted\_words) return sorted\_sentence

if name == " main ":

input\_sentence = "This is a sample sentence to sort alphabetically." sorted\_sentence = sort\_sentence(input\_sentence)

print("Original Sentence:", input\_sentence) print("Sorted Sentence:", sorted\_sentence)

* 1. **Write a Python program for the following Crypt arithmetic problems CROSS+ROADS = DANGER**

**Ans:-**

from itertools import permutations

def is\_solution(mapping):

cross = mapping['C'] \* 10000 + mapping['R'] \* 1000 + mapping['O'] \* 100 + mapping['S'] \* 10 + mapping['S']

roads = mapping['R'] \* 10000 + mapping['O'] \* 1000 + mapping['A'] \* 100 + mapping['D'] \* 10 + mapping['S']

danger = mapping['D'] \* 100000 + mapping['A'] \* 10000 + mapping['N'] \* 1000 + mapping['G'] \* 100 + mapping['E'] \* 10 + mapping['R']

return cross + roads == danger

def solve\_cryptarithmetic():

for p in permutations(range(10), 8):

mapping = {'C': p[0], 'R': p[1], 'O': p[2], 'S': p[3], 'A': p[4], 'D': p[5], 'N': p[6], 'G': p[7], 'E': p[8]}

if is\_solution(mapping): return mapping

return None

if name == " main ": solution = solve\_cryptarithmetic()

if solution:

print("Solution found:") print(f" C = {solution['C']}")

print(f" R = {solution['R']}")

print(f" O = {solution['O']}")

print(f" S = {solution['S']}")

print(f" A = {solution['A']}")

print(f" D = {solution['D']}")

print(f" N = {solution['N']}")

print(f" G = {solution['G']}")

print(f" E = {solution['E']}") print("\n CROSS") print("+ ROADS")

print(" ")

print(f" DANGER") else:

print("No solution found.")

# Slip 25

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

from flask import Flask, request, jsonify app = Flask( name )

college\_info = {

"name": "Sample College", "location": "City, Country",

"programs": ["Computer Science", "Business Administration", "Engineering"], "facilities": ["Library", "Gym", "Sports Fields"],

}

@app.route('/college\_chatbot', methods=['POST']) def college\_chatbot():

data = request.get\_json()

if 'query' in data:

query = data['query'].lower()

if 'name' in query:

response = f"The college's name is {college\_info['name']}." elif 'location' in query:

response = f"The college is located in {college\_info['location']}." elif 'programs' in query:

response = f"The college offers programs in {', '.join(college\_info['programs'])}." elif 'facilities' in query:

response = f"The college provides facilities such as {', '.join(college\_info['facilities'])}."

else:

response = "I'm sorry, I don't understand that query." return jsonify({"response": response})

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

if name == ' main ': app.run(debug=True)

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

def print\_chessboard(chessboard): for row in chessboard:

print(" ".join(row))

# Function to check if it's safe to place a queen at the given position def is\_safe(chessboard, row, col, n):

# Check row on the left side for i in range(col):

if chessboard[row][i] == 'Q': return False

# Check upper diagonal on the left side

for i, j in zip(range(row, -1, -1), range(col, -1, -1)): if chessboard[i][j] == 'Q':

return False

# Check lower diagonal on the left side

for i, j in zip(range(row, n, 1), range(col, -1, -1)): if chessboard[i][j] == 'Q':

return False return True

# Recursive function to solve the Eight Queens problem def solve\_eight\_queens(chessboard, col, n):

if col >= n:

return True # All queens are placed

for i in range(n):

if is\_safe(chessboard, i, col, n): chessboard[i][col] = 'Q' # Place a queen

# Recur to place the rest of the queens

if solve\_eight\_queens(chessboard, col + 1, n): return True

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

return False # No solution exists

# Main function to solve the Eight Queens problem def main():

n = 8 # Size of the chessboard (8x8)

chessboard = [['.' for \_ in range(n)] for \_ in range(n)]

if solve\_eight\_queens(chessboard, 0, n): print("Solution to the Eight Queens Problem:") print\_chessboard(chessboard)

else:

print("No solution found.")

if name == ' main ': main()