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

def hill\_climbing(function, initial\_x, step\_size, max\_iterations):

"""

Implements the hill climbing algorithm to find the maximum of a function.

Args:

function: The function to optimize.

initial\_x: The initial value of x.

step\_size: The step size for each iteration.

max\_iterations: The maximum number of iterations.

Returns:

The maximum value of the function and the corresponding x value.

"""

x = initial\_x

for \_ in range(max\_iterations):

neighbor1 = x + step\_size

neighbor2 = x - step\_size

if function(neighbor1) > function(x):

x = neighbor1

elif function(neighbor2) > function(x):

x = neighbor2

else:

break

return function(x), x

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

def f(x):

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

# Parameters

initial\_x = random.uniform(-10, 10)

step\_size = 0.1

max\_iterations = 100

# Run hill climbing

max\_value, max\_x = hill\_climbing(f, initial\_x, step\_size, max\_iterations)

print("Maximum value:", max\_value)

print("x value:", max\_x)

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]

# Depth First Search implementation

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

# If visited is None, initialize an empty set

if visited is None:

visited = set()

# Mark the current node as visited

visited.add(start)

print(f"Visiting node: {start}")

# If we have reached the goal, return True

if start == goal:

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

return True

# Explore the neighbors

for neighbor in graph[start]:

if neighbor not in visited:

# Recursively perform DFS on the neighbor

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

return True

# If no path leads to the goal, return False

return False

# Define the graph as an adjacency list

graph = {

1: [2, 3],

2: [4, 5],

3: [6, 7],

4: [],

5: [],

6: [],

7: [8],

8: []

}

# Starting node and goal node

start\_node = 1

goal\_node = 8

# Call DFS function

if dfs(graph, start\_node, goal\_node):

print(f"Path to goal node {goal\_node} exists!")

else:

print(f"No path exists from node {start\_node} to node {goal\_node}.")

Q.3 Write a python program to generate Calendar for the given month and year?

#calander

import calendar

year=int(input("enter a year"))

month=int(input("enter a month"))

print(calendar.month(year,month))

Q.4)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].

# Depth First Search implementation

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

# If visited is None, initialize an empty set

if visited is None:

visited = set()

# Mark the current node as visited

visited.add(start)

print(f"Visiting node: {start}")

# If we have reached the goal, return True

if start == goal:

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

return True

# Explore the neighbors

for neighbor in graph[start]:

if neighbor not in visited:

# Recursively perform DFS on the neighbor

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

return True

# If no path leads to the goal, return False

return False

# Define the graph as an adjacency list

graph = {

1: [2, 3],

2: [4, 5],

3: [6],

4: [],

5: [7],

6: [],

7: []

}

# Starting node and goal node

start\_node = 1

goal\_node = 7

# Call DFS function

if dfs(graph, start\_node, goal\_node):

print(f"Path to goal node {goal\_node} exists!")

else:

print(f"No path exists from node {start\_node} to node {goal\_node}.")

Q.5Write a python program to remove punctuations from the given string?

import string

def remove\_punctuation(input\_string):

# Create a translation table that removes punctuation

translation\_table = str.maketrans('', '', string.punctuation)

# Use the translate method to remove punctuation

return input\_string.translate(translation\_table)

# Example usage

input\_string = "Hello, world! How's it going?"

output\_string = remove\_punctuation(input\_string)

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

print("String without Punctuation:", output\_string)

Q.6 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]

# DFS Implementation in Python

# Representing the graph as an adjacency list

graph = {

1: [2, 3],

2: [1, 4, 5],

3: [1, 6],

4: [2],

5: [2, 7],

6: [3],

7: [5]

}

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

if path is None:

path = []

if visited is None:

visited = set()

# Mark the start node as visited and add it to the current path

visited.add(start)

path.append(start)

# If the current node is the goal node, return the path

if start == goal:

return path

# Recurse for each neighbor of the current node

for neighbor in graph[start]:

if neighbor not in visited:

result = dfs(graph, neighbor, goal, path, visited)

if result: # If a path to the goal was found

return result

# Backtrack: Remove the node from the path if no goal is found in this path

path.pop()

return None # Return None if no path is found

# Initial node and goal node

start\_node = 2

goal\_node = 7

# Call the DFS function

path = dfs(graph, start\_node, goal\_node)

# Output the result

if path:

print(f"Path from node {start\_node} to node {goal\_node}: {path}")

else:

print(f"No path found from node {start\_node} to node {goal\_node}")

Q.7 )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

# List of possible words for the game

word\_list = ["python", "programming", "hangman", "developer", "algorithm", "software", "computer"]

# Function to display the current status of the guessed word

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

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

# Function to play the Hangman game

def hangman\_game():

# Randomly select a word from the list

word = random.choice(word\_list)

# Set of letters guessed by the user

guessed\_letters = set()

# Number of attempts before the player loses

attempts = 6

# Game loop

while attempts > 0:

# Display the current status of the word

print(f"Word: {display\_word(word, guessed\_letters)}")

print(f"Remaining attempts: {attempts}")

# Get user input for a letter

guess = input("Guess a letter: ").lower()

# Ensure the input is a single alphabetic character

if len(guess) != 1 or not guess.isalpha():

print("Please enter a valid single letter.")

continue

# If the letter has already been guessed

if guess in guessed\_letters:

print("You already guessed that letter. Try again.")

continue

# Add the guessed letter to the set of guessed letters

guessed\_letters.add(guess)

# If the guessed letter is in the word, show a success message

if guess in word:

print(f"Good guess! '{guess}' is in the word.")

else:

print(f"Oops! '{guess}' is not in the word.")

attempts -= 1

# Check if the player has guessed all the letters in the word

if all(letter in guessed\_letters for letter in word):

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

break

else:

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

# Main function to start the game

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

print("Welcome to Hangman!")

hangman\_game()

Q.8 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

# Representing the graph as an adjacency list

graph = {

1: [2, 3],

2: [1, 4, 5],

3: [1, 6],

4: [2],

5: [2, 7],

6: [3],

7: [5, 8],

8: [7]

}

def bfs(graph, start, goal):

# Queue for BFS, initialized with the start node

queue = deque([[start]]) # each element in queue is a path

visited = set() # set to track visited nodes

# BFS loop

while queue:

# Get the current path

path = queue.popleft()

node = path[-1] # last node in the path

# If the goal node is found, return the path

if node == goal:

return path

# If the node has not been visited

if node not in visited:

visited.add(node)

# Add all the neighbors of the current node to the queue

for neighbor in graph[node]:

new\_path = list(path) # Create a new path to avoid modifying the old one

new\_path.append(neighbor)

queue.append(new\_path)

return None # Return None if no path is found

# Initial node and goal node

start\_node = 1

goal\_node = 8

# Call the BFS function

path = bfs(graph, start\_node, goal\_node)

# Output the result

if path:

print(f"Path from node {start\_node} to node {goal\_node}: {path}")

else:

print(f"No path found from node {start\_node} to node {goal\_node}")

Q.9. Write a python program to implement Lemmatization using NLTK

import nltk

from nltk.stem import WordNetLemmatizer

from nltk.corpus import wordnet

# Ensure required NLTK data is downloaded

nltk.download('wordnet')

nltk.download('omw-1.4')

# Initialize the WordNetLemmatizer

lemmatizer = WordNetLemmatizer()

# Function to perform lemmatization

def lemmatize\_word(word, pos\_tag='n'):

"""

Lemmatizes a word with respect to its part of speech (POS) tag.

The POS tag can be 'n' for noun, 'v' for verb, 'a' for adjective, etc.

:param word: Word to be lemmatized

:param pos\_tag: Part of speech tag ('n', 'v', 'a', etc.)

:return: Lemmatized word

"""

return lemmatizer.lemmatize(word, pos=pos\_tag)

# Example words to demonstrate lemmatization

words = ["running", "better", "geese", "watched", "mice", "flies", "happier"]

# Lemmatizing with different POS tags

lemmatized\_words = []

for word in words:

# For noun and verb lemmatization

noun\_lemma = lemmatize\_word(word, pos\_tag='n')

verb\_lemma = lemmatize\_word(word, pos\_tag='v')

lemmatized\_words.append({

"original": word,

"noun\_lemma": noun\_lemma,

"verb\_lemma": verb\_lemma

})

# Output the results

for result in lemmatized\_words:

print(f"Original word: {result['original']}")

print(f"Lemmatized as Noun: {result['noun\_lemma']}")

print(f"Lemmatized as Verb: {result['verb\_lemma']}")

print("-" \* 30)

Q.10 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 represented as an adjacency list

graph = {

1: [2, 3],

2: [1, 4, 5],

3: [1, 6],

4: [2],

5: [2, 7],

6: [3],

7: [5, 8],

8: [7]

}

# Function to perform Breadth-First Search (BFS)

def bfs(graph, start, goal):

# Queue to store the paths for BFS

queue = deque([[start]]) # Initial path contains only the start node

visited = set() # Set to track visited nodes

while queue:

# Dequeue the first path in the queue

path = queue.popleft()

node = path[-1] # Get the last node in the current path

# If the goal node is found, return the path

if node == goal:

return path

# Mark the node as visited

if node not in visited:

visited.add(node)

# Add all the neighbors to the queue

for neighbor in graph[node]:

# Create a new path by appending the neighbor

new\_path = list(path) # Make a copy of the current path

new\_path.append(neighbor)

queue.append(new\_path)

return None # Return None if no path is found

# Initial node and goal node

start\_node = 1

goal\_node = 8

# Call the BFS function

path = bfs(graph, start\_node, goal\_node)

# Output the result

if path:

print(f"Path from node {start\_node} to node {goal\_node}: {path}")

else:

print(f"No path found from node {start\_node} to node {goal\_node}")

Q.11. 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

# Ensure that the necessary NLTK resources are downloaded

nltk.download('punkt')

nltk.download('stopwords')

# Function to remove stop words from a text

def remove\_stop\_words(text):

# Tokenize the text into words

words = word\_tokenize(text)

# Get the set of stop words

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

# Filter out the stop words from the tokenized words

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

# Join the filtered words back into a string

filtered\_text = ' '.join(filtered\_words)

return filtered\_text

# Function to read a text file and remove stop words

def process\_text\_file(file\_path):

try:

# Open the file and read the content

with open(file\_path, 'r') as file:

text = file.read()

# Remove stop words from the text

filtered\_text = remove\_stop\_words(text)

return filtered\_text

except FileNotFoundError:

return "The file was not found."

# Provide the path to your text file

file\_path = 'input\_text.txt' # Change this to your file's path

# Process the text file and print the result

filtered\_text = process\_text\_file(file\_path)

if filtered\_text != "The file was not found.":

print("Filtered Text (without stop words):")

print(filtered\_text)

else:

print(filtered\_text)

Q.12. 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 represented as an adjacency list

graph = {

1: [2, 3],

2: [1, 4, 5],

3: [1, 6],

4: [2],

5: [2, 7],

6: [3],

7: [5, 8],

8: [7]

}

# Function to perform Breadth-First Search (BFS)

def bfs(graph, start, goal):

# Queue to store the paths for BFS

queue = deque([[start]]) # Initialize with the start node

visited = set() # Set to track visited nodes

while queue:

# Dequeue the first path in the queue

path = queue.popleft()

node = path[-1] # Get the last node in the current path

# If the goal node is found, return the path

if node == goal:

return path

# Mark the node as visited

if node not in visited:

visited.add(node)

# Add all the neighbors to the queue

for neighbor in graph[node]:

# Create a new path by appending the neighbor

new\_path = list(path) # Copy the current path

new\_path.append(neighbor)

queue.append(new\_path)

return None # Return None if no path is found

# Initial node and goal node

start\_node = 1

goal\_node = 8

# Call the BFS function to find the path

path = bfs(graph, start\_node, goal\_node)

# Output the result

if path:

print(f"Path from node {start\_node} to node {goal\_node}: {path}")

else:

print(f"No path found from node {start\_node} to node {goal\_node}")

Q.13. Write a python program implement tic-tac-toe using alpha beeta pruning

import math

def evaluate(board):

# Check for win conditions

for row in range(3):

if board[row][0] == board[row][1] == board[row][2]:

if board[row][0] == 'X':

return 10

elif board[row][0] == 'O':

return -10

for col in range(3):

if board[0][col] == board[1][col] == board[2][col]:

if board[0][col] == 'X':

return 10

elif board[0][col] == 'O':

return -10

if board[0][0] == board[1][1] == board[2][2]:

if board[0][0] == 'X':

return 10

elif board[0][0] == 'O':

return -10

if board[0][2] == board[1][1] == board[2][0]:

if board[0][2] == 'X':

return 10

elif board[0][2] == 'O':

return -10

# Check for draw

for i in range(3):

for j in range(3):

if board[i][j] == '\_':

return 0

return 0

def minimax(board, depth, isMax, alpha, beta):

score = evaluate(board)

if score == 10:

return score

if score == -10:

return score

if isMax:

best = -math.inf

for i in range(3):

for j in range(3):

if board[i][j] == '\_':

board[i][j] = 'X'

best = max(best, minimax(board, depth + 1, not isMax, alpha, beta))

board[i][j] = '\_'

alpha = max(alpha, best)

if beta <= alpha:

break

return best

else:

best = math.inf

for i in range(3):

for j in range(3):

if board[i][j] == '\_':

board[i][j] = 'O'

best = min(best, minimax(board, depth + 1, not isMax, alpha, beta))

board[i][j] = '\_'

beta = min(beta, best)

if beta <= alpha:

break

return best

def findBestMove(board):

bestVal = -math.inf

bestMove = (-1, -1)

for i in range(3):

for j in range(3):

if board[i][j] == '\_':

board[i][j] = 'X'

moveVal = minimax(board, 0, False, -math.inf, math.inf)

board[i][j] = '\_'

if moveVal > bestVal:

bestMove = (i, j)

bestVal = moveVal

return bestMove

def printBoard(board):

for i in range(3):

for j in range(3):

print(board[i][j], end=" ")

print()

# Driver code

board = [

['\_', '\_', '\_'],

['\_', '\_', '\_'],

['\_', '\_', '\_']

]

player = 'X'

computer = 'O'

while True:

printBoard(board)

if player == 'X':

row, col = map(int, input("Enter row and column numbers: ").split())

board[row - 1][col - 1] = player

else:

row, col = findBestMove(board)

board[row][col] = computer

if evaluate(board) == 10:

printBoard(board)

print("X wins")

break

elif evaluate(board) == -10:

printBoard(board)

print("O wins")

break

elif evaluate(board) == 0 and not any('\_' in row for row in board):

printBoard(board)

print("Draw")

break

player, computer = computer, player

Q.14. Write a Python program to implement Simple Chatbot.

import random

import time

# Predefined set of questions and responses

responses = {

"hello": ["Hi there!", "Hello! How can I help you?", "Hey! What can I do for you today?"],

"how are you": ["I'm doing great, thanks for asking!", "I'm fine, how about you?", "I'm just a bot, but I'm doing well!"],

"what is your name": ["I am your friendly chatbot!", "I don't have a specific name, but you can call me Bot!", "I am Chatbot, nice to meet you!"],

"bye": ["Goodbye! Have a great day!", "See you later!", "Bye! Take care!"],

"default": ["Sorry, I didn't understand that.", "Can you please rephrase?", "I'm not sure how to respond to that."]

}

# Function to simulate bot typing

def bot\_typing():

typing\_phases = ['...', '...', '...', '...', 'typing...']

for phase in typing\_phases:

print(f"Bot {phase}", end="\r")

time.sleep(0.5)

# Function to get bot's response

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

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

# If the user input matches a predefined question

for key in responses:

if key in user\_input:

bot\_typing() # Simulate typing

return random.choice(responses[key])

# Default response if no match is found

bot\_typing()

return random.choice(responses["default"])

# Main function to start the chatbot

def start\_chat():

print("Chatbot: Hello! I am here to chat. Type 'bye' to exit.")

while True:

user\_input = input("You: ")

if 'bye' in user\_input.lower():

print("Chatbot: Goodbye! Take care!")

break

# Get the bot's response

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

# Print bot's response

print(f"Chatbot: {response}")

# Start the chatbot

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

start\_chat()

Q.15. Write a Python program to accept a string. Find and print the number of upper case alphabets and lower case alphabets.

# Function to count uppercase and lowercase letters

def count\_case\_letters(s):

upper\_count = 0

lower\_count = 0

for char in s:

if char.isupper(): # Check if the character is uppercase

upper\_count += 1

elif char.islower(): # Check if the character is lowercase

lower\_count += 1

return upper\_count, lower\_count

# Main function

def main():

# Accepting the input string

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

# Counting uppercase and lowercase letters

upper, lower = count\_case\_letters(user\_input)

# Printing the result

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

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

# Calling the main function

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

main()

Q.16. Write a Python program to solve tic-tac-toe problem.

# Function to print the Tic-Tac-Toe board

def print\_board(board):

print(f"{board[0]} | {board[1]} | {board[2]}")

print("--+---+--")

print(f"{board[3]} | {board[4]} | {board[5]}")

print("--+---+--")

print(f"{board[6]} | {board[7]} | {board[8]}")

# Function to check for a win

def check\_win(board, player):

win\_conditions = [(0, 1, 2), (3, 4, 5), (6, 7, 8), # Rows

(0, 3, 6), (1, 4, 7), (2, 5, 8), # Columns

(0, 4, 8), (2, 4, 6)] # Diagonals

for condition in win\_conditions:

if board[condition[0]] == board[condition[1]] == board[condition[2]] == player:

return True

return False

# Function to check if the board is full

def is\_full(board):

return all(spot != ' ' for spot in board)

# Main function to play the game

def play\_game():

board = [' '] \* 9 # Initial empty board

current\_player = 'X' # Player X starts

while True:

print\_board(board)

move = int(input(f"Player {current\_player}, enter your move (1-9): ")) - 1

# Check if the move is valid

if board[move] != ' ':

print("Invalid move! The spot is already taken. Try again.")

continue

# Make the move

board[move] = current\_player

# Check for a win

if check\_win(board, current\_player):

print\_board(board)

print(f"Player {current\_player} wins!")

break

# Check for a tie

if is\_full(board):

print\_board(board)

print("It's a tie!")

break

# Switch players

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

# Run the game

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

play\_game()

Q.17. Write python program to solve 8 puzzle problem using A\* algorithm.

from collections import deque

def is\_solvable(puzzle):

inversion\_count = 0

empty\_cell\_row = 0

for i in range(len(puzzle)):

for j in range(i+1, len(puzzle)):

if puzzle[i] > puzzle[j] and puzzle[j] != 0:

inversion\_count += 1

if puzzle[i] == 0:

empty\_cell\_row = i

return inversion\_count % 2 == 0 if empty\_cell\_row % 2 == 0 else inversion\_count % 2 == 1

def find\_empty\_tile(state):

for i in range(3):

for j in range(3):

if state[i][j] == 0:

return i, j

def generate\_child\_states(state, empty\_tile\_row, empty\_tile\_col):

new\_states = []

if empty\_tile\_row > 0:

new\_state = [row.copy() for row in state]

new\_state[empty\_tile\_row][empty\_tile\_col], new\_state[empty\_tile\_row-1][empty\_tile\_col] = new\_state[empty\_tile\_row-1][empty\_tile\_col], new\_state[empty\_tile\_row][empty\_tile\_col]

new\_states.append(new\_state)

if empty\_tile\_row < 2:

new\_state = [row.copy() for row in state]

new\_state[empty\_tile\_row][empty\_tile\_col], new\_state[empty\_tile\_row+1][empty\_tile\_col] = new\_state[empty\_tile\_row+1][empty\_tile\_col], new\_state[empty\_tile\_row][empty\_tile\_col]

new\_states.append(new\_state)

if empty\_tile\_col > 0:

new\_state = [row.copy() for row in state]

new\_state[empty\_tile\_row][empty\_tile\_col], new\_state[empty\_tile\_row][empty\_tile\_col-1] = new\_state[empty\_tile\_row][empty\_tile\_col-1], new\_state[empty\_tile\_row][empty\_tile\_col]

new\_states.append(new\_state)

if empty\_tile\_col < 2:

new\_state = [row.copy() for row in state]

new\_state[empty\_tile\_row][empty\_tile\_col], new\_state[empty\_tile\_row][empty\_tile\_col+1] = new\_state[empty\_tile\_row][empty\_tile\_col+1], new\_state[empty\_tile\_row][empty\_tile\_col]

new\_states.append(new\_state)

return new\_states

def manhattan\_distance(state):

distance = 0

for i in range(3):

for j in range(3):

if state[i][j] != 0:

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

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

return distance

def 1 a\_star\_search(initial\_state, goal\_state):

if not is\_solvable(initial\_state):

return "Puzzle is unsolvable"

open\_set = deque()

closed\_set = set()

open\_set.append((initial\_state, 0, manhattan\_distance(initial\_state)))

while open\_set:

current\_state, current\_g, current\_h = open\_set.popleft()

current\_state\_tuple = tuple(tuple(row) for row in current\_state)

if current\_state\_tuple in closed\_set:

continue

closed\_set.add(current\_state\_tuple)

if current\_state == goal\_state:

return current\_state

empty\_tile\_row, empty\_tile\_col = find\_empty\_tile(current\_state)

for child\_state in generate\_child\_states(current\_state, empty\_tile\_row, empty\_tile\_col):

child\_state\_tuple = tuple(tuple(row) for row in child\_state)

if child\_state\_tuple not in closed\_set:

child\_g = current\_g + 1

child\_h = manhattan\_distance(child\_state)

child\_f = child\_g + child\_h

open\_set.append((child\_state, child\_g, child\_h))

open\_set = deque(sorted(open\_set, key=lambda x: x[2]))

return "No solution found"

# Example usage

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

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

result = a\_star\_search(initial\_state, goal\_state)

if result:

print("Solution found:")

for row in result:

print(row)

else:

print(result)

Q.18 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.

from collections import deque

# Function to find the solution to the water jug problem

def water\_jug\_problem(capacity1, capacity2, target):

# Queue to store the states (x, y) where x is the amount of water in jug 1 and y is the amount in jug 2

queue = deque([(0, 0)]) # Initial state (0, 0), both jugs are empty

visited = set() # Set to track visited states

visited.add((0, 0)) # Add the initial state to visited

# List of possible actions (operations)

operations = [

("Fill Jug1", lambda x, y: (capacity1, y)), # Fill Jug1

("Fill Jug2", lambda x, y: (x, capacity2)), # Fill Jug2

("Empty Jug1", lambda x, y: (0, y)), # Empty Jug1

("Empty Jug2", lambda x, y: (x, 0)), # Empty Jug2

("Pour Jug1 to Jug2", lambda x, y: (x - min(x, capacity2 - y), y + min(x, capacity2 - y))), # Pour Jug1 to Jug2

("Pour Jug2 to Jug1", lambda x, y: (x + min(y, capacity1 - x), y - min(y, capacity1 - x))) # Pour Jug2 to Jug1

]

while queue:

current\_state = queue.popleft()

x, y = current\_state

# If we reach the goal state, return the solution

if y == target:

print(f"Solution found: Jug 1 = {x} gallons, Jug 2 = {y} gallons")

return True

# Explore all possible actions

for op\_name, operation in operations:

new\_state = operation(x, y)

# If the new state has not been visited, add it to the queue and mark it as visited

if new\_state not in visited:

visited.add(new\_state)

queue.append(new\_state)

print(f"State: Jug 1 = {new\_state[0]} gallons, Jug 2 = {new\_state[1]} gallons (via {op\_name})")

print("No solution found")

return False

# Driver function to start the solution

def main():

capacity1 = 5 # Capacity of Jug 1 (5 gallons)

capacity2 = 7 # Capacity of Jug 2 (7 gallons)

target = 4 # Goal is to measure 4 gallons in Jug 2

# Solve the water jug problem

water\_jug\_problem(capacity1, capacity2, target)

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

main()

Q.19. Write Python program to implement crypt arithmetic problem TWO+TWO=FOUR

import itertools

# Function to solve the cryptarithmetic problem TWO + TWO = FOUR

def solve\_cryptarithmetic():

# List of distinct letters in the equation TWO + TWO = FOUR

letters = ['T', 'W', 'O', 'F', 'U', 'R']

# Generate all possible permutations of digits for these 6 letters

for perm in itertools.permutations(range(10), 6):

# Create a dictionary that maps letters to digits

mapping = dict(zip(letters, perm))

# Extract the digits for each letter

T = mapping['T']

W = mapping['W']

O = mapping['O']

F = mapping['F']

U = mapping['U']

R = mapping['R']

# Form the numbers TWO and FOUR

TWO = 100 \* T + 10 \* W + O

FOUR = 1000 \* F + 100 \* O + 10 \* U + R

# Check if the equation TWO + TWO = FOUR holds

if TWO + TWO == FOUR:

print(f"Solution found:")

print(f"TWO = {TWO}, FOUR = {FOUR}")

print(f"Mapping: T={T}, W={W}, O={O}, F={F}, U={U}, R={R}")

return

print("No solution found")

# Driver function to execute the solution

def main():

solve\_cryptarithmetic()

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

main()

Q.20. Write a Python program to implement Simple Chatbot.

import random

# Predefined responses for various inputs

responses = {

"hi": ["Hello!", "Hi there!", "Hey! How can I help you?"],

"how are you": ["I'm doing great, thank you!", "I'm just a bot, but I'm doing fine.", "I'm good, how about you?"],

"your name": ["I am a chatbot created by you.", "You can call me Chatbot.", "I don't have a name, but I am happy to help you!"],

"bye": ["Goodbye!", "See you later!", "Take care!"],

"default": ["Sorry, I don't understand that.", "Can you please rephrase?", "I'm not sure what you mean."]

}

# Function to get a response from the chatbot

def get\_response(user\_input):

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

# Check for specific responses

for key in responses:

if key in user\_input:

return random.choice(responses[key]) # Return a random response from the list

# If no specific match, return a default response

return random.choice(responses["default"])

# Main function to interact with the chatbot

def main():

print("Chatbot: Hello! I'm your friendly chatbot. Type 'bye' to end the conversation.")

while True:

user\_input = input("You: ") # Take input from the user

if user\_input.lower() == 'bye': # End the conversation if the user types 'bye'

print("Chatbot:", get\_response(user\_input))

break

response = get\_response(user\_input)

print("Chatbot:", response)

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

main()

Q.21. Write a python program using mean end analysis algorithmproblem of transforming a string of lowercase letters into another string.

def mean\_end\_analysis(start, goal):

# Initialize the steps (actions) to an empty list

steps = []

# While the start string is not equal to the goal string

while start != goal:

# If the start string and goal string are of different lengths

if len(start) < len(goal):

# We can add a character (insert operation)

# Insert the character that is needed in the goal string

start += goal[len(start)] # Add the next required character from goal to start

steps.append(f"Insert '{goal[len(start)-1]}' at position {len(start)-1}")

elif len(start) > len(goal):

# We can delete a character (delete operation)

start = start[:-1] # Remove the last character

steps.append(f"Delete '{start[-1]}' from position {len(start)}")

else:

# Strings are of the same length, perform a substitution

for i in range(len(start)):

if start[i] != goal[i]:

# Replace the character in start with the character from goal

steps.append(f"Substitute '{start[i]}' at position {i} with '{goal[i]}'")

start = start[:i] + goal[i] + start[i+1:] # Perform substitution

break # After one transformation, go to the next iteration

return steps

# Driver function to test the solution

def main():

start = "abcde" # Example start string

goal = "abzxy" # Example goal string

print(f"Start String: {start}")

print(f"Goal String: {goal}")

# Call the Mean-End Analysis function

steps = mean\_end\_analysis(start, goal)

# Output the steps (actions) performed to transform the start string to the goal string

print("\nTransformation Steps:")

for step in steps:

print(step)

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

main()

Q.22. 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.

from collections import deque

# Define the capacity of both jugs

jug1\_capacity = 4

jug2\_capacity = 3

target = 2 # Target is to get 2 gallons in Jug 2

# List to track visited states (to avoid revisiting)

visited = set()

# Function to solve the water jug problem using BFS

def water\_jug\_bfs(jug1\_capacity, jug2\_capacity, target):

# Queue to store the current state (amount of water in Jug 1, amount of water in Jug 2)

queue = deque([(0, 0)]) # Start with both jugs empty

visited.add((0, 0)) # Mark initial state as visited

# List of operations and their corresponding state transitions

while queue:

jug1, jug2 = queue.popleft()

# Check if we have reached the target in Jug 2

if jug2 == target:

print(f"Solution found: Jug 1 = {jug1}, Jug 2 = {jug2}")

return True

# Generate all possible states from the current state

# Fill Jug 1

if (jug1\_capacity, jug2) not in visited:

visited.add((jug1\_capacity, jug2))

queue.append((jug1\_capacity, jug2))

print(f"Fill Jug 1: Jug 1 = {jug1\_capacity}, Jug 2 = {jug2}")

# Fill Jug 2

if (jug1, jug2\_capacity) not in visited:

visited.add((jug1, jug2\_capacity))

queue.append((jug1, jug2\_capacity))

print(f"Fill Jug 2: Jug 1 = {jug1}, Jug 2 = {jug2\_capacity}")

# Empty Jug 1

if (0, jug2) not in visited:

visited.add((0, jug2))

queue.append((0, jug2))

print(f"Empty Jug 1: Jug 1 = {0}, Jug 2 = {jug2}")

# Empty Jug 2

if (jug1, 0) not in visited:

visited.add((jug1, 0))

queue.append((jug1, 0))

print(f"Empty Jug 2: Jug 1 = {jug1}, Jug 2 = {0}")

# Pour water from Jug 1 to Jug 2

transfer = min(jug1, jug2\_capacity - jug2) # Water that can be transferred

if (jug1 - transfer, jug2 + transfer) not in visited:

visited.add((jug1 - transfer, jug2 + transfer))

queue.append((jug1 - transfer, jug2 + transfer))

print(f"Pour Jug 1 to Jug 2: Jug 1 = {jug1 - transfer}, Jug 2 = {jug2 + transfer}")

# Pour water from Jug 2 to Jug 1

transfer = min(jug2, jug1\_capacity - jug1) # Water that can be transferred

if (jug1 + transfer, jug2 - transfer) not in visited:

visited.add((jug1 + transfer, jug2 - transfer))

queue.append((jug1 + transfer, jug2 - transfer))

print(f"Pour Jug 2 to Jug 1: Jug 1 = {jug1 + transfer}, Jug 2 = {jug2 - transfer}")

# If no solution is found

print("No solution found")

return False

# Driver function

def main():

print("Starting Water Jug Problem solver...")

water\_jug\_bfs(jug1\_capacity, jug2\_capacity, target)

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

main()

Q.23. Write a python program to generate Calendar for the given month and year?

#calander

import calendar

year=int(input("enter a year"))

month=int(input("enter a month"))

print(calendar.month(year,month))

Q.24 Write a Python program to simulate 4-Queens problem.

def is\_safe(board, row, col):

# Check this row on left side

for i in range(col):

if board[row][i] == 1:

return False

# Check upper diagonal on left side

for i, j in zip(range(row, -1, -1), range(col, -1, -1)):

if board[i][j] == 1:

return False

# Check lower diagonal on left side

for i, j in zip(range(row, 4, 1), range(col, -1, -1)):

if board[i][j] == 1:

return False

return True

def solve\_n\_queens\_util(board, col):

if col >= 4:

return True

for i in range(4):

if is\_safe(board, i, col):

board[i][col] = 1

if solve\_n\_queens\_util(board, col + 1):

return True

board[i][col] = 0

return False

def solve\_n\_queens():

board = [[0 for \_ in range(4)] for \_ in range(4)]

if not solve\_n\_queens\_util(board, 0):

print("Solution does not exist")

return False

print\_solution(board)

return True

def print\_solution(board):

for i in range(4):

for j in range(4):

print(board[i][j], end=' ')

print()

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

solve\_n\_queens()

Q.25 Write a Python program to implement Mini-Max Algorithm.

# Constants representing the players

PLAYER\_X = 'X'

PLAYER\_O = 'O'

EMPTY = ' '

# Check if the current player has won

def is\_winner(board, player):

# Check rows, columns, and diagonals

for i in range(3):

if board[i][0] == board[i][1] == board[i][2] == player: # Check row

return True

if board[0][i] == board[1][i] == board[2][i] == player: # Check column

return True

if board[0][0] == board[1][1] == board[2][2] == player: # Check diagonal

return True

if board[0][2] == board[1][1] == board[2][0] == player: # Check other diagonal

return True

return False

# Check if the board is full (draw condition)

def is\_board\_full(board):

for row in board:

if EMPTY in row:

return False

return True

# Get the available moves (empty cells)

def get\_available\_moves(board):

moves = []

for i in range(3):

for j in range(3):

if board[i][j] == EMPTY:

moves.append((i, j))

return moves

# Minimax function to evaluate the best move

def minimax(board, depth, is\_maximizing\_player):

# Base cases: check for terminal states

if is\_winner(board, PLAYER\_X):

return 1 # Maximizing player wins

if is\_winner(board, PLAYER\_O):

return -1 # Minimizing player wins

if is\_board\_full(board):

return 0 # Draw

# Maximizing player (Player X)

if is\_maximizing\_player:

best = -float('inf') # Start with the worst possible score

for move in get\_available\_moves(board):

i, j = move

board[i][j] = PLAYER\_X # Make the move

score = minimax(board, depth + 1, False) # Recursively evaluate the new state

board[i][j] = EMPTY # Undo the move

best = max(best, score) # Take the maximum score

return best

else:

# Minimizing player (Player O)

best = float('inf') # Start with the best possible score

for move in get\_available\_moves(board):

i, j = move

board[i][j] = PLAYER\_O # Make the move

score = minimax(board, depth + 1, True) # Recursively evaluate the new state

board[i][j] = EMPTY # Undo the move

best = min(best, score) # Take the minimum score

return best

# Function to find the best move for the maximizing player (Player X)

def find\_best\_move(board):

best\_val = -float('inf')

best\_move = None

for move in get\_available\_moves(board):

i, j = move

board[i][j] = PLAYER\_X # Make the move

move\_val = minimax(board, 0, False) # Get the score of this move

board[i][j] = EMPTY # Undo the move

if move\_val > best\_val:

best\_val = move\_val

best\_move = move # Update best move

return best\_move

# Function to print the board

def print\_board(board):

for row in board:

print(" | ".join(row))

print("-" \* 5)

# Driver code

def main():

# Initialize the board with empty spaces

board = [

[EMPTY, EMPTY, EMPTY],

[EMPTY, EMPTY, EMPTY],

[EMPTY, EMPTY, EMPTY]

]

# Start the game

print("Tic-Tac-Toe Game!")

print\_board(board)

while True:

# Player X's turn (Maximizing Player)

print("Player X's move:")

move = find\_best\_move(board)

i, j = move

board[i][j] = PLAYER\_X

print\_board(board)

# Check if Player X won

if is\_winner(board, PLAYER\_X):

print("Player X wins!")

break

# Check if the board is full (Draw)

if is\_board\_full(board):

print("It's a draw!")

break

# Player O's turn (Minimizing Player)

print("Player O's move:")

move = find\_best\_move(board)

i, j = move

board[i][j] = PLAYER\_O

print\_board(board)

# Check if Player O won

if is\_winner(board, PLAYER\_O):

print("Player O wins!")

break

# Check if the board is full (Draw)

if is\_board\_full(board):

print("It's a draw!")

break

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

main()

Q.26 Write a Python program to simulate 8-Queens problem.

# Constants for the board size

N = 8

# Function to print the chessboard

def print\_board(board):

for row in board:

print(" ".join(row))

print("\n")

# Function to check if it's safe to place a queen at position (row, col)

def is\_safe(board, row, col):

# Check the column

for i in range(row):

if board[i][col] == 'Q':

return False

# Check upper-left diagonal

for i, j in zip(range(row-1, -1, -1), range(col-1, -1, -1)):

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

return False

# Check upper-right diagonal

for i, j in zip(range(row-1, -1, -1), range(col+1, N)):

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

return False

return True

# Backtracking function to solve the 8-Queens problem

def solve\_8\_queens(board, row):

# If all queens are placed

if row == N:

print\_board(board)

return True

# Try placing a queen in each column of the current row

for col in range(N):

if is\_safe(board, row, col):

board[row][col] = 'Q' # Place the queen

# Recursively place queens in the next row

if solve\_8\_queens(board, row + 1):

return True

# If placing queen leads to a solution, return True

# Otherwise, backtrack (remove the queen)

board[row][col] = '.' # Undo the move (backtrack)

# If no place is found in this row, return False

return False

# Function to solve the 8-Queens problem

def solve():

# Create an 8x8 board initialized with '.'

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

# Call the backtracking function to solve the problem

if not solve\_8\_queens(board, 0):

print("Solution does not exist.")

else:

print("Solution found:")

# Driver function to run the program

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

solve()

Q.27. Write a python program to sort the sentence in alphabetical order?

def sort\_sentence(sentence):

# Split the sentence into words

words = sentence.split()

# Sort the words in alphabetical order

words.sort()

# Join the sorted words back into a sentence

sorted\_sentence = ' '.join(words)

return sorted\_sentence

# Input sentence

sentence = input("Enter a sentence: ")

# Get the sorted sentence

sorted\_sentence = sort\_sentence(sentence)

# Output the sorted sentence

print("Sorted sentence:", sorted\_sentence)

Q.28 Write a Python program to simulate n-Queens problem.

def is\_safe(board, row, col, n):

# Check this row on left side

for i in range(col):

if board[row][i] == 1:

return False

# Check upper diagonal on left side

for i, j in zip(range(row, -1, -1), range(col, -1, -1)):

if board[i][j] == 1:

return False

# Check lower diagonal on left side

for i, j in zip(range(row, n, 1), range(col, -1, -1)):

if board[i][j] == 1:

return False

return True

def solve\_n\_queens\_util(board, col, n):

if col >= n:

return True

for i in range(n):

if is\_safe(board, i, col, n):

board[i][col] = 1

if solve\_n\_queens\_util(board, col + 1, n):

return True

board[i][col] = 0

return False

def solve\_n\_queens(n):

board = [[0 for \_ in range(n)] for \_ in range(n)]

if not solve\_n\_queens\_util(board, 0, n):

print("Solution does not exist")

return False

print\_solution(board)

return True

def print\_solution(board):

for i in range(len(board)):

for j in range(len(board)):

print(board[i][j], end=' ')

print()

# Example usage

n = 8

solve\_n\_queens(n)

Q.29. Write a Program to Implement Monkey Banana Problem using Python

def monkey\_banana(boxes, banana\_height):

"""

Solves the monkey banana problem using a recursive approach.

Args:

boxes: A list of box heights.

banana\_height: The height of the banana bunch.

Returns:

True if the monkey can reach the bananas, False otherwise.

"""

if banana\_height <= 0:

return True # Monkey can reach the bananas

if not boxes:

return False # No more boxes to stack

# Try stacking the current box

return monkey\_banana(boxes[1:], banana\_height - boxes[0]) or \

monkey\_banana(boxes[1:], banana\_height) # Don't use the current box

# Example usage:

box\_heights = [2, 3, 1, 4]

banana\_height = 6

if monkey\_banana(box\_heights, banana\_height):

print("The monkey can reach the bananas!")

else:

print("The monkey cannot reach the bananas.")

Q.30 Write a program to implement Iterative Deepening DFS algorithm. [ Goal Node =G]

# Define the graph as an adjacency list

graph = {

'A': ['B', 'C'],

'B': ['A', 'D', 'E'],

'C': ['A', 'F'],

'D': ['B'],

'E': ['B', 'G'],

'F': ['C'],

'G': ['E']

}

# Function to perform DFS up to a given depth

def dfs(graph, node, goal, visited, depth):

# If the goal is found at this node, return the path

if node == goal:

return [node]

# If the depth limit is reached, return None

if depth == 0:

return None

# Mark the node as visited

visited.add(node)

# Explore all neighbors at the current level

for neighbor in graph[node]:

if neighbor not in visited:

path = dfs(graph, neighbor, goal, visited, depth - 1)

if path:

return [node] + path

# Backtrack by removing the node from visited set

visited.remove(node)

return None

# Function for Iterative Deepening DFS

def iddfs(graph, start, goal):

# Start depth from 0 and increase until goal is found

depth = 0

while True:

visited = set() # Reset visited set for each depth iteration

path = dfs(graph, start, goal, visited, depth)

if path:

return path

depth += 1 # Increment depth if goal is not found

# Driver code to run the IDDFS algorithm

start\_node = 'A'

goal\_node = 'G'

path = iddfs(graph, start\_node, goal\_node)

if path:

print("Path to goal node G:", ' -> '.join(path))

else:

print("Goal node not found.")

Q.31 Write a Program to Implement Tower of Hanoi using Python

# Function to solve the Tower of Hanoi problem

def tower\_of\_hanoi(n, source, destination, auxiliary):

# Base case: If only one disk is left, move it from source to destination

if n == 1:

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

return

# Move top n-1 disks from source to auxiliary, using destination as auxiliary

tower\_of\_hanoi(n - 1, source, auxiliary, destination)

# Move the nth disk from source to destination

print(f"Move disk {n} from {source} to {destination}")

# Move n-1 disks from auxiliary to destination, using source as auxiliary

tower\_of\_hanoi(n - 1, auxiliary, destination, source)

# Driver code

def main():

n = int(input("Enter the number of disks: "))

tower\_of\_hanoi(n, 'A', 'C', 'B') # A, B, C are the names of rods

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

main()

Q.32 Write a Python program to solve tic-tac-toe problem.

# Function to initialize the game board

def initialize\_board():

return [[' ' for \_ in range(3)] for \_ in range(3)]

# Function to print the board

def print\_board(board):

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

for row in board:

print("|", " | ".join(row), "|")

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

# Function to check if a player has won

def check\_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 board[0][0] == player and board[1][1] == player and board[2][2] == player:

return True

if board[0][2] == player and board[1][1] == player and board[2][0] == player:

return True

return False

# Function to check if the board is full

def is\_board\_full(board):

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

# Function to handle player's move

def make\_move(board, player):

while True:

try:

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

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

board[row][col] = player

break

else:

print("That spot is already taken. Try again.")

except (ValueError, IndexError):

print("Invalid move. Please enter row and column values between 0 and 2.")

# Function to play the game

def play\_game():

board = initialize\_board()

current\_player = 'X'

# Main game loop

while True:

print\_board(board)

make\_move(board, current\_player)

# Check for winner

if check\_winner(board, current\_player):

print\_board(board)

print(f"Player {current\_player} wins!")

break

# Check for draw

if is\_board\_full(board):

print\_board(board)

print("It's a draw!")

break

# Switch players

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

# Start the game

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

play\_game()

**Q.33 Python program that demonstrates the hill climbing algorithm to find the maximum of a mathematical function.**

import random

# The objective function (mathematical function to maximize)

def objective\_function(x):

return -x\*\*2 + 5\*x + 3

# Hill Climbing algorithm to find the maximum of the function

def hill\_climbing(start\_point, step\_size=0.1, max\_iterations=1000):

current\_point = start\_point

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

for \_ in range(max\_iterations):

# Generate a neighboring solution by adjusting the current point

neighbor\_left = current\_point - step\_size

neighbor\_right = current\_point + step\_size

# Evaluate the neighboring solutions

value\_left = objective\_function(neighbor\_left)

value\_right = objective\_function(neighbor\_right)

# Check if any neighbor is better (higher value)

if value\_left > current\_value:

current\_point = neighbor\_left

current\_value = value\_left

elif value\_right > current\_value:

current\_point = neighbor\_right

current\_value = value\_right

else:

# If no neighbor improves, stop (local maximum found)

break

return current\_point, current\_value

# Driver code to run the Hill Climbing algorithm

def main():

# Start at a random point, let's say from -2 to 7

start\_point = random.uniform(-2, 7)

# Run the Hill Climbing algorithm

best\_point, best\_value = hill\_climbing(start\_point)

print(f"Starting point: {start\_point:.4f}")

print(f"Best point (maximum): {best\_point:.4f}")

print(f"Maximum value: {best\_value:.4f}")

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

main()

Q.34 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]

import heapq

# Define the graph with nodes and their neighbors (edges with costs)

graph = {

'A': [('B', 1), ('C', 4)],

'B': [('A', 1), ('C', 2), ('D', 5)],

'C': [('A', 4), ('B', 2), ('D', 1), ('G', 3)],

'D': [('B', 5), ('C', 1), ('G', 1)],

'G': [('C', 3), ('D', 1)],

}

# Heuristic function for each node (estimated cost to reach goal node 'G')

heuristics = {

'A': 6,

'B': 5,

'C': 2,

'D': 1,

'G': 0

}

# A\* algorithm implementation

def a\_star(start, goal):

# Open set to store the nodes to be evaluated, and the closed set for visited nodes

open\_set = []

heapq.heappush(open\_set, (heuristics[start], start)) # (f, node)

# G values: cost from start to the current node

g\_values = {start: 0}

# Path dictionary to store the path from start to goal

came\_from = {}

while open\_set:

# Get the node with the lowest f-value (f = g + h)

\_, current = heapq.heappop(open\_set)

# If we reached the goal, reconstruct the path

if current == goal:

path = []

while current in came\_from:

path.append(current)

current = came\_from[current]

path.append(start)

path.reverse()

return path, g\_values[goal]

# Explore neighbors

for neighbor, cost in graph[current]:

tentative\_g = g\_values[current] + cost

# If the neighbor is not visited or we found a better path

if neighbor not in g\_values or tentative\_g < g\_values[neighbor]:

g\_values[neighbor] = tentative\_g

f\_value = tentative\_g + heuristics[neighbor]

heapq.heappush(open\_set, (f\_value, neighbor))

came\_from[neighbor] = current

return None, None # In case no path is found

# Driver code

def main():

start\_node = 'A'

goal\_node = 'G'

# Run A\* algorithm

path, cost = a\_star(start\_node, goal\_node)

if path:

print(f"Path from {start\_node} to {goal\_node}: {' -> '.join(path)}")

print(f"Total cost: {cost}")

else:

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

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

main()

Q.35 Write a python program to remove stop words for a given passage from a text file using NLTK?.

import nltk

from nltk.corpus import stopwords

from nltk.tokenize import word\_tokenize

# Download necessary resources from NLTK

nltk.download('punkt')

nltk.download('stopwords')

# Function to remove stop words from the given text

def remove\_stop\_words(text):

# Tokenize the text into words

words = word\_tokenize(text)

# Get the list of stop words from NLTK (for English)

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

# Remove stop words by filtering out the words that are in the stopwords list

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

# Return the filtered text (joined back into a string)

return ' '.join(filtered\_words)

# Function to read a file and process it

def process\_file(file\_path):

# Open and read the content of the text file

with open(file\_path, 'r') as file:

text = file.read()

# Remove stop words from the text

cleaned\_text = remove\_stop\_words(text)

# Print the cleaned text or save it to another file

print("Cleaned Text (without stop words):")

print(cleaned\_text)

# Example Usage: Provide the path to your text file

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

# Change this path to the location of your text file

file\_path = 'input\_text.txt'

process\_file(file\_path)

Q.36 Implement a system that performs arrangement of some set of objects in a room. Assume that you have only 5 rectangular, 4 square-shaped objects. Use A\* approach for the placement of the objects in room for efficient space utilisation. Assume suitable heuristic, and dimensions of objects and rooms. (Informed Search)

import heapq

# Object dimensions (width, height)

rectangular\_objects = [(2, 4)] \* 5 # 5 rectangular objects of size 2x4

square\_objects = [(3, 3)] \* 4 # 4 square objects of size 3x3

# Room dimensions

room\_width = 10

room\_height = 10

# Define the room as a grid

room = [[0] \* room\_width for \_ in range(room\_height)] # 0 means unoccupied, 1 means occupied

# Heuristic: Remaining empty area in the room

def heuristic(state):

empty\_area = sum(row.count(0) for row in state)

return empty\_area

# Function to check if an object fits in a given position in the room

def can\_place\_object(state, obj\_width, obj\_height, x, y):

# Check if the object fits within the bounds of the room

if x + obj\_width > room\_width or y + obj\_height > room\_height:

return False

# Check if the object overlaps with any already placed object

for i in range(x, x + obj\_width):

for j in range(y, y + obj\_height):

if state[j][i] == 1: # Occupied space

return False

return True

# Function to place an object in the room at a given position

def place\_object(state, obj\_width, obj\_height, x, y):

new\_state = [row[:] for row in state] # Make a copy of the current state

for i in range(x, x + obj\_width):

for j in range(y, y + obj\_height

Q.36Write a program to implement Hangman game using python.

import random

# List of words to choose from

word\_list = ["python", "java", "hangman", "programming", "developer", "algorithm", "computer"]

# Function to choose a random word

def choose\_word():

return random.choice(word\_list)

# Function to display the hangman figure based on incorrect guesses

def display\_hangman(tries):

hangman\_pics = [

'''

-----

| |

|

|

|

|

''', '''

-----

| |

O |

|

|

|

''', '''

-----

| |

O |

| |

|

|

''', '''

-----

| |

O |

/| |

|

|

''', '''

-----

| |

O |

/|\\ |

|

|

''', '''

-----

| |

O |

/|\\ |

/ |

|

''', '''

-----

| |

O |

/|\\ |

/ \\ |

|

'''

]

return hangman\_pics[tries]

# Function to play the Hangman game

def hangman():

word = choose\_word() # Choose a random word from the word list

word\_length = len(word)

guessed\_word = ['\_'] \* word\_length # List to track guessed letters in the word

guessed\_letters = [] # To store the letters guessed by the player

tries = 0 # To keep track of incorrect guesses

max\_tries = 6 # The maximum number of wrong attempts

print("Welcome to Hangman!")

while tries < max\_tries:

print("\nWord to guess: ", ' '.join(guessed\_word))

print("Guessed letters: ", ', '.join(guessed\_letters))

print(display\_hangman(tries)) # Display the hangman figure

guess = input("Enter a letter: ").lower() # Prompt the player for a guess

# Check for valid input

if len(guess) != 1 or not guess.isalpha():

print("Please enter a single valid letter.")

continue

if guess in guessed\_letters:

print("You already guessed that letter. Try again.")

continue

guessed\_letters.append(guess) # Add the guessed letter to the list

# Check if the guess is correct

if guess in word:

print(f"Good guess! The letter '{guess}' is in the word.")

# Update the guessed word with the correct letter

for i in range(word\_length):

if word[i] == guess:

guessed\_word[i] = guess

else:

print(f"Oops! The letter '{guess}' is not in the word.")

tries += 1

# Check if the player has guessed the word

if ''.join(guessed\_word) == word:

print("\nCongratulations! You've guessed the word:", word)

break

else:

# The player ran out of tries

print(display\_hangman(tries))

print(f"\nGame Over! The word was: {word}")

# Run the game

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

hangman()

Q.37 Write a Python program to implement A\* algorithm. Refer the following graph as an Input for the program.

import heapq

# Define the graph as an adjacency list where each key is a node,

# and its value is a list of tuples (neighbor, cost).

graph = {

'A': [('B', 1), ('C', 4)],

'B': [('A', 1), ('C', 2), ('D', 5)],

'C': [('A', 4), ('B', 2), ('D', 1), ('E', 3)],

'D': [('B', 5), ('C', 1), ('F', 2)],

'E': [('C', 3), ('F', 2)],

'F': [('D', 2), ('E', 2), ('G', 1)],

'G': [('F', 1)]

}

# Define heuristic values for each node (h(n))

heuristics = {

'A': 6,

'B': 4,

'C': 3,

'D': 2,

'E': 1,

'F': 0,

'G': 0

}

# A\* Search Algorithm

def a\_star(start, goal):

# Priority queue to store (f, g, node) where f = g + h

open\_set = []

heapq.heappush(open\_set, (heuristics[start], 0, start)) # f, g, node

# Dictionary to store the best cost to reach a node (g(n))

g\_cost = {start: 0}

# Dictionary to track the path

came\_from = {}

while open\_set:

# Pop the node with the lowest f value

\_, g, current = heapq.heappop(open\_set)

# If we reached the goal, reconstruct the path

if current == goal:

path = []

while current in came\_from:

path.append(current)

current = came\_from[current]

path.append(start)

return path[::-1] # Return reversed path

# Explore neighbors

for neighbor, cost in graph[current]:

tentative\_g = g + cost

# If this path to the neighbor is better, update its g cost and add to open set

if neighbor not in g\_cost or tentative\_g < g\_cost[neighbor]:

g\_cost[neighbor] = tentative\_g

f = tentative\_g + heuristics[neighbor]

heapq.heappush(open\_set, (f, tentative\_g, neighbor))

came\_from[neighbor] = current

return None # If no path is found

# Test the A\* algorithm

start\_node = 'A'

goal\_node = 'G'

path = a\_star(start\_node, goal\_node)

if path:

print("Path found:", " -> ".join(path))

else:

print("No path found")

Q.38 Build a bot which provides all the information related to you in college

import random

# Define possible responses and information

college\_info = {

"name": "XYZ University",

"location": "123 University Ave, College City, Country",

"history": "XYZ University was founded in 1980 and has been providing quality education ever since. It is known for its research and academic excellence.",

"contact": "You can contact the college at +1 234 567 890 or email us at info@xyzuniversity.edu"

}

courses = {

"computer science": "Bachelor of Science in Computer Science (BSc CS) - A 4-year course that covers programming, algorithms, data structures, and more.",

"mechanical engineering": "Bachelor of Science in Mechanical Engineering (BSc ME) - A 4-year course that includes thermodynamics, mechanics, and robotics.",

"business administration": "Bachelor of Business Administration (BBA) - A 3-year course focusing on management, economics, and marketing.",

}

faculty = {

"computer science": ["Dr. Alice Smith", "Prof. John Doe", "Dr. Sarah Lee"],

"mechanical engineering": ["Prof. Mark Taylor", "Dr. Lisa Brown"],

"business administration": ["Dr. Emily Clark", "Prof. James Wilson"],

}

events = [

"Guest Lecture on Artificial Intelligence - 25th Nov 2024",

"Tech Fest - 15th Dec 2024",

"Workshop on Data Science - 20th Dec 2024"

]

# Default responses

responses = {

"greeting": ["Hello! How can I assist you today?", "Hi there! How can I help you?", "Greetings! What information do you need?"],

"thank\_you": ["You're welcome!", "Glad to help!", "Happy to assist!"],

"goodbye": ["Goodbye! Have a great day!", "See you later!", "Take care!"]

}

# Function to get college information

def get\_college\_info(query):

query = query.lower()

if "name" in query or "college" in query:

return college\_info["name"]

elif "location" in query:

return college\_info["location"]

elif "history" in query:

return college\_info["history"]

elif "contact" in query or "contact details" in query:

return college\_info["contact"]

else:

return "Sorry, I didn't understand your query about college information."

# Function to get course details

def get\_course\_info(query):

query = query.lower()

for course in courses:

if course in query:

return courses[course]

return "Sorry, I don't have information about that course."

# Function to get faculty details

def get\_faculty\_info(query):

query = query.lower()

for department in faculty:

if department in query:

return f"Faculty in {department.capitalize()} department: " + ", ".join(faculty[department])

return "Sorry, I couldn't find any faculty information for that department."

# Function to get events

def get\_events():

return "Upcoming Events:\n" + "\n".join(events)

# Bot response function

def respond\_to\_query(query):

query = query.lower()

# Respond based on type of query

if "college" in query or "information" in query:

return get\_college\_info(query)

elif "course" in query:

return get\_course\_info(query)

elif "faculty" in query or "professor" in query:

return get\_faculty\_info(query)

elif "event" in query or "upcoming events" in query:

return get\_events()

elif "thanks" in query or "thank you" in query:

return random.choice(responses["thank\_you"])

elif "bye" in query or "goodbye" in query:

return random.choice(responses["goodbye"])

else:

return random.choice(responses["greeting"])

# Main function to interact with the bot

def run\_chatbot():

print("Welcome to the College Information Bot!")

print("You can ask me about the college, courses, faculty, events, or any other general queries.")

while True:

user\_query = input("\nYou: ").lower()

# Exit condition for chatbot

if "bye" in user\_query or "exit" in user\_query:

print("Bot: " + random.choice(responses["goodbye"]))

break

# Get and print the bot response

response = respond\_to\_query(user\_query)

print("Bot: " + response)

# Run the chatbot

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

run\_chatbot()

Q.39 Write a Python program to implement Mini-Max Algorithm

import math

def evaluate(board):

# Check for win conditions

for row in range(3):

if board[row][0] == board[row][1] == board[row][2]:

if board[row][0] == 'X':

return 10

elif board[row][0] == 'O':

return -10

for col in range(3):

if board[0][col] == board[1][col] == board[2][col]:

if board[0][col] == 'X':

return 10

elif board[0][col] == 'O':

return -10

if board[0][0] == board[1][1] == board[2][2]:

if board[0][0] == 'X':

return 10

elif board[0][0] == 'O':

return -10

if board[0][2] == board[1][1] == board[2][0]:

if board[0][2] == 'X':

return 10

elif board[0][2] == 'O':

return -10

# Check for draw

for i in range(3):

for j in range(3):

if board[i][j] == '\_':

return 0

return 0

def minimax(board, depth, isMax):

score = evaluate(board)

if score == 10:

return score

if score == -10:

return score

if isMax:

best = -math.inf

for i in range(3):

for j in range(3):

if board[i][j] == '\_':

board[i][j] = 'X'

best = max(best, minimax(board, depth + 1, not isMax))

board[i][j] = '\_'

return best

else:

best = math.inf

for i in range(3):

for j in range(3):

if board[i][j] == '\_':

board[i][j] = 'O'

best = min(best, minimax(board, depth + 1, not isMax))

board[i][j] = '\_'

return best

def findBestMove(board):

bestVal = -math.inf

bestMove = (-1, -1)

for i in range(3):

for j in range(3):

if board[i][j] == '\_':

board[i][j] = 'X'

moveVal = minimax(board, 0, False)

board[i][j] = '\_'

if moveVal > bestVal:

bestMove = (i, j)

bestVal = moveVal

return bestMove

# ... (rest of the Tic-Tac-Toe game logic)

Q.40Write a python program to remove punctuations from the given string?

import string

def remove\_punctuations(input\_string):

# Create a translation table that maps each punctuation mark to None

translator = str.maketrans('', '', string.punctuation)

# Return the string after removing punctuations

return input\_string.translate(translator)

# Example usage:

input\_string = "Hello, World! This is a test: do you want to remove #punctuations?"

result = remove\_punctuations(input\_string)

print("Original string:", input\_string)

print("String without punctuations:", result)

Q.41 Write a Python program for the following Cryptarithmetic problems. [ 20 Marks ] GO + TO = OUT

from itertools import permutations

# Function to solve the cryptarithmetic equation

def solve\_cryptarithmetic():

# We know the letters involved are G, O, T, U

letters = ['G', 'O', 'T', 'U']

# Generate all possible permutations of digits 0-9 for the four letters

# There are 10 digits and we need to assign them to 4 letters

for perm in permutations(range(10), 4):

# Create a dictionary to map letters to digits

letter\_to\_digit = dict(zip(letters, perm))

# Assign the digits to variables G, O, T, U

G = letter\_to\_digit['G']

O = letter\_to\_digit['O']

T = letter\_to\_digit['T']

U = letter\_to\_digit['U']

# Convert the letters into their respective numeric values

GO = 10 \* G + O

TO = 10 \* T + O

OUT = 100 \* O + 10 \* T + U

# Check if the equation is satisfied

if GO + TO == OUT:

# If the equation holds, print the result

print(f"Solution found:")

print(f"G = {G}, O = {O}, T = {T}, U = {U}")

print(f"GO = {GO}, TO = {TO}, OUT = {OUT}")

return # Return once we find a valid solution

print("No solution found.")

# Call the function to solve the cryptarithmetic problem

solve\_cryptarithmetic()

Q.42 Write a Program to Implement Alpha-Beta Pruning using Python

# Alpha-Beta Pruning Implementation

# Define a sample game tree as a dictionary with nodes and their children

# Each node contains a tuple of value and its children (if any)

def alpha\_beta\_pruning(node, depth, alpha, beta, maximizing\_player):

# Terminal condition, return the value of the node if we reach the leaf node

if depth == 0 or not node['children']:

return node['value']

if maximizing\_player:

max\_eval = float('-inf')

for child in node['children']:

eval = alpha\_beta\_pruning(child, depth-1, alpha, beta, False) # Minimize

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

alpha = max(alpha, eval) # Update alpha

if beta <= alpha:

break # Beta cut-off

return max\_eval

else:

min\_eval = float('inf')

for child in node['children']:

eval = alpha\_beta\_pruning(child, depth-1, alpha, beta, True) # Maximize

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

beta = min(beta, eval) # Update beta

if beta <= alpha:

break # Alpha cut-off

return min\_eval

# Example tree structure

# The tree is represented as a dictionary. Each node has a 'value' and a list of 'children' nodes.

# Define a simple example tree (can be replaced with any game tree)

root = {

'value': None,

'children': [

{

'value': None,

'children': [

{'value': 3, 'children': []},

{'value': 12, 'children': []},

{'value': 8, 'children': []}

]

},

{

'value': None,

'children': [

{'value': 2, 'children': []},

{'value': 6, 'children': []},

{'value': 14, 'children': []}

]

},

{

'value': None,

'children': [

{'value': 5, 'children': []},

{'value': 7, 'children': []},

{'value': 4, 'children': []}

]

}

]

}

# Max depth of the tree (the number of levels we want to search)

max\_depth = 3

# Starting with alpha=-infinity and beta=+infinity

alpha = float('-inf')

beta = float('inf')

# Start with the maximizing player at the root node

best\_value = alpha\_beta\_pruning(root, max\_depth, alpha, beta, True)

# Print the result

print("Best value:", best\_value)

Q.43 Write a Python program for the following Cryptarithmetic problems SEND + MORE = MONEY

from itertools import permutations

def solve\_send\_more\_money():

# We know the letters involved are: S, E, N, D, M, O, R, Y

letters = ['S', 'E', 'N', 'D', 'M', 'O', 'R', 'Y']

# Generate all permutations of digits 0-9 for the 8 letters (using only 8 distinct digits)

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

# Create a dictionary that maps each letter to a digit

letter\_to\_digit = dict(zip(letters, perm))

# Extract digits for each letter

S = letter\_to\_digit['S']

E = letter\_to\_digit['E']

N = letter\_to\_digit['N']

D = letter\_to\_digit['D']

M = letter\_to\_digit['M']

O = letter\_to\_digit['O']

R = letter\_to\_digit['R']

Y = letter\_to\_digit['Y']

# Calculate SEND, MORE, and MONEY based on the current assignment

SEND = 1000\*S + 100\*E + 10\*N + D

MORE = 1000\*M + 100\*O + 10\*R + E

MONEY = 10000\*M + 1000\*O + 100\*N + 10\*E + Y

# Check if the equation SEND + MORE = MONEY holds

if SEND + MORE == MONEY:

# If it holds, print the solution

print(f"Solution found:")

print(f"S = {S}, E = {E}, N = {N}, D = {D}")

print(f"M = {M}, O = {O}, R = {R}, Y = {Y}")

print(f"SEND = {SEND}, MORE = {MORE}, MONEY = {MONEY}")

return # Exit once the solution is found

print("No solution found.")

# Call the function to solve the puzzle

solve\_send\_more\_money()

Q.44 Write a python program to sort the sentence in alphabetical order?

def sort\_sentence(sentence):

# Split the sentence into words

words = sentence.split()

# Sort the words alphabetically

words.sort()

# Join the sorted words back into a sentence

sorted\_sentence = ' '.join(words)

return sorted\_sentence

# Example usage:

sentence = "the quick brown fox jumps over the lazy dog"

sorted\_sentence = sort\_sentence(sentence)

print("Original Sentence: ", sentence)

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

Q.45 Write a Python program for the following Cryptorithmetic problems CROSS+ROADS = DANGER

from itertools import permutations

def solve\_cross\_roads\_danger():

# Letters involved: C, R, O, S, A, D, N, G, E

letters = ['C', 'R', 'O', 'S', 'A', 'D', 'N', 'G', 'E']

# Generate all permutations of digits 0-9 for the 9 letters (using 9 distinct digits)

for perm in permutations(range(10), 9):

# Create a dictionary to map each letter to a digit

letter\_to\_digit = dict(zip(letters, perm))

# Extract digits for each letter

C = letter\_to\_digit['C']

R = letter\_to\_digit['R']

O = letter\_to\_digit['O']

S = letter\_to\_digit['S']

A = letter\_to\_digit['A']

D = letter\_to\_digit['D']

N = letter\_to\_digit['N']

G = letter\_to\_digit['G']

E = letter\_to\_digit['E']

# Calculate CROSS, ROADS, and DANGER based on the current assignment

CROSS = 10000\*C + 1000\*R + 100\*O + 10\*S + S

ROADS = 10000\*R + 1000\*O + 100\*A + 10\*D + S

DANGER = 100000\*D + 10000\*A + 1000\*N + 100\*G + 10\*E + R

# Check if the equation CROSS + ROADS = DANGER holds

if CROSS + ROADS == DANGER:

# If it holds, print the solution

print(f"Solution found:")

print(f"C = {C}, R = {R}, O = {O}, S = {S}")

print(f"A = {A}, D = {D}, N = {N}, G = {G}, E = {E}")

print(f"CROSS = {CROSS}, ROADS = {ROADS}, DANGER = {DANGER}")

return # Exit once we find a valid solution

print("No solution found.")

# Call the function to solve the puzzle

solve\_cross\_roads\_danger()

Q.46 Write a Python program to solve 8-puzzle problem.

import heapq

# The goal state of the 8-puzzle

GOAL\_STATE = (1, 2, 3, 4, 5, 6, 7, 8, 0)

# Define directions for the movement of the blank space

DIRECTIONS = [(-1, 0), (1, 0), (0, -1), (0, 1)] # up, down, left, right

def manhattan\_distance(state):

"""Calculate the Manhattan distance for the given state."""

distance = 0

for i in range(9):

value = state[i]

if value == 0:

continue

target\_row, target\_col = divmod(value - 1, 3)

current\_row, current\_col = divmod(i, 3)

distance += abs(current\_row - target\_row) + abs(current\_col - target\_col)

return distance

def get\_neighbors(state):

"""Return a list of valid neighboring states by moving the blank space."""

blank\_index = state.index(0)

row, col = divmod(blank\_index, 3)

neighbors = []

for dr, dc in DIRECTIONS:

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

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

new\_blank\_index = new\_row \* 3 + new\_col

new\_state = list(state)

# Swap the blank space with the adjacent tile

new\_state[blank\_index], new\_state[new\_blank\_index] = new\_state[new\_blank\_index], new\_state[blank\_index]

neighbors.append(tuple(new\_state))

return neighbors

def a\_star\_search(start\_state):

"""Solve the 8-puzzle using A\* search."""

# Priority queue to hold states and their priorities (f = g + h)

open\_list = []

# Set to keep track of visited states

visited = set()

# Push the start state into the priority queue with priority (f = g + h)

g = 0 # Cost to reach the current state

h = manhattan\_distance(start\_state) # Heuristic estimate (Manhattan distance)

f = g + h

heapq.heappush(open\_list, (f, g, start\_state))

visited.add(start\_state)

# Dictionary to track the parent state for path reconstruction

parent\_map = {start\_state: None}

while open\_list:

# Pop the state with the lowest f-value from the priority queue

\_, g, current\_state = heapq.heappop(open\_list)

# If we reached the goal state, reconstruct the path

if current\_state == GOAL\_STATE:

path = []

while current\_state is not None:

path.append(current\_state)

current\_state = parent\_map[current\_state]

return path[::-1] # Return reversed path to get the sequence from start to goal

# Explore the neighbors of the current state

for neighbor in get\_neighbors(current\_state):

if neighbor not in visited:

visited.add(neighbor)

g\_new = g + 1 # Cost to reach the neighbor

h\_new = manhattan\_distance(neighbor) # Heuristic estimate for the neighbor

f\_new = g\_new + h\_new

heapq.heappush(open\_list, (f\_new, g\_new, neighbor))

parent\_map[neighbor] = current\_state

return None # Return None if no solution is found

def print\_state(state):

"""Print the state in a 3x3 grid format."""

for i in range(0, 9, 3):

print(state[i:i+3])

# Test the A\* search on a sample 8-puzzle problem

start\_state = (2, 8, 3, 1, 6, 4, 7, 0, 5) # An example initial state

print("Initial State:")

print\_state(start\_state)

solution = a\_star\_search(start\_state)

if solution:

print("\nSolution path:")

for state in solution:

print\_state(state)

print("---")

else:

print("\nNo solution found.")