**First 7 slip solutions**

**Slip 1:**

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

import numpy as np

def objective\_function(x):

"""

The objective function to maximize.

"""

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

def hill\_climbing(initial\_x, step\_size, num\_steps):

"""

Hill climbing algorithm to find the maximum of a function.

"""

current\_x = initial\_x

for step in range(num\_steps):

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

next\_x = current\_x + step\_size

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

if next\_value > current\_value:

current\_x = next\_x

return current\_x, objective\_function(current\_x)

# Set initial parameters

initial\_x = 0.0

step\_size = 0.1

num\_steps = 100

# Run the hill climbing algorithm

result\_x, result\_value = hill\_climbing(initial\_x, step\_size, num\_steps)

# Print the results

print(f"Maximum found at x = {result\_x}")

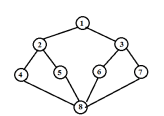
print(f"Maximum value is {result\_value}")

**Output:**

Maximum found at x = 2.0000000000000004

Maximum value is 4.0

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



def dfs(graph, start, goal):

stack = [start]

visited = []

print('The path traversed is:')

while stack:

# Pop from stack to set current element

element = stack.pop()

print(element, end=" ")

if element == goal:

print("\nGoal found!")

return True

if element not in visited:

visited.append(element)

for neighbor in graph[element]:

if neighbor not in visited:

stack.append(neighbor)

print("\nGoal not found.")

return False

# A dictionary representing the illustrated graph

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']

}

start = '1'

goal = '8'

# Call the DFS function

dfs(graph, start, goal)

**Output:**

The path traversed is:

1 3 7 8

**Slip 2:**

**Q1:** Q.1) Write a python program to generate Calendar for the given month and year?. [ 10 Marks]

**import calendar**

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

**# Create a TextCalendar instance**

**cal = calendar.TextCalendar(calendar.SUNDAY)**

**# Generate the calendar for the given month and year**

**calendar\_text = cal.formatmonth(year, month)**

**# Print the calendar**

**print(calendar\_text)**

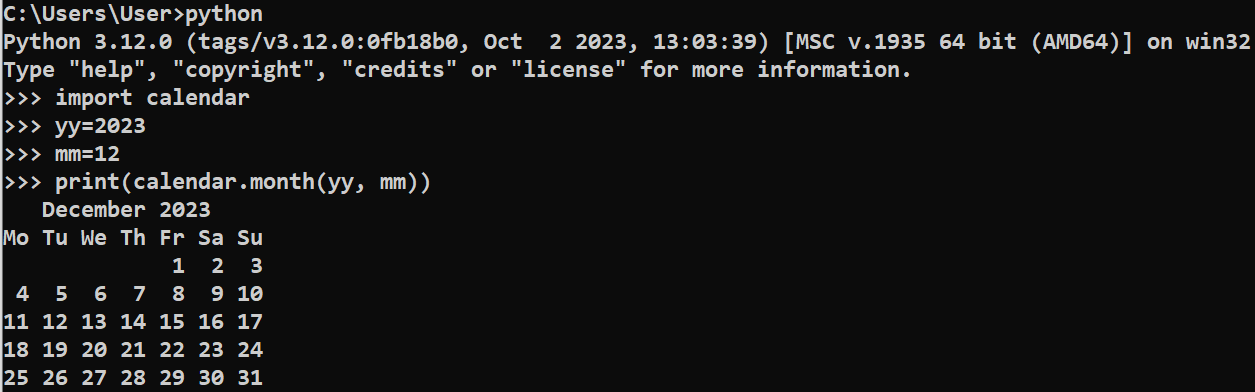
**# Get user input for the year and month**

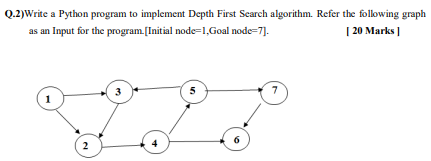
**year = int(input("Enter the year: "))**

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

**# Generate and print the calendar**

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





# DFS algorithm in Python

class Graph:

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

self.graph = {}

def add\_edge(self, start, end):

if start not in self.graph:

self.graph[start] = []

self.graph[start].append(end)

def dfs(graph, current, goal, path=[]):

path = path + [current]

if current == goal:

return path

if current not in graph:

return None

for neighbor in graph[current]:

if neighbor not in path:

new\_path = dfs(graph, neighbor, goal, path)

if new\_path:

return new\_path

return None

# Example Usage:

# Create a directed graph and add edges

g = Graph()

g.add\_edge('1', '2')

g.add\_edge('1', '3')

g.add\_edge('2', '4')

g.add\_edge('3', '2')

g.add\_edge('4', '5')

g.add\_edge('4', '6')

g.add\_edge('5', '3')

g.add\_edge('5', '7')

g.add\_edge('7', '6')

# Perform DFS from node '1' to '7'

initial\_node = '1'

goal\_node = '7'

result\_path = dfs(g.graph, initial\_node, goal\_node)

# Display the result

if result\_path:

print(f"Path from {initial\_node} to {goal\_node}: {result\_path}")

else:

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

Output:

Path from 1 to 7: ['1', '2', '4', '5', '7']

**Slip 3:**

Q.1) Write a python program to remove punctuations from the given string? .[ 10 marks ]

import string

def remove\_punctuation(input\_string):

# Create a translation table with None for all punctuation characters

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

# Use the translation table to remove punctuations from the input string

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

return result\_string

# Example Usage:

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

result = remove\_punctuation(input\_string)

print("Original String:")

print(input\_string)

print("\nString after removing punctuations:")

print(result)

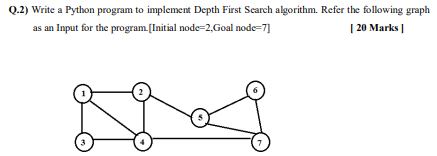
**Output:**

Original String:

Hello, world! This is an example string with punctuations!!!

String after removing punctuations:

Hello world This is an example string with punctuations



# DFS algorithm in Python

class Graph:

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

self.graph = {}

def add\_edge(self, node1, node2):

# Add edges for an undirected graph

if node1 not in self.graph:

self.graph[node1] = []

if node2 not in self.graph:

self.graph[node2] = []

self.graph[node1].append(node2)

self.graph[node2].append(node1)

def dfs(graph, current, goal, path=[]):

path = path + [current]

if current == goal:

return path

if current not in graph:

return None

for neighbor in graph[current]:

if neighbor not in path:

new\_path = dfs(graph, neighbor, goal, path)

if new\_path:

return new\_path

return None

# Example Usage:

# Create an undirected graph and add edges

g = Graph()

g.add\_edge('1', '2')

g.add\_edge('1', '3')

g.add\_edge('1', '4')

g.add\_edge('2', '1')

g.add\_edge('2', '4')

g.add\_edge('2', '5')

g.add\_edge('3', '1')

g.add\_edge('3', '4')

g.add\_edge('4', '2')

g.add\_edge('4', '7')

g.add\_edge('5', '2')

g.add\_edge('5', '6')

g.add\_edge('5', '7')

g.add\_edge('6', '5')

g.add\_edge('6', '7')

g.add\_edge('7', '4')

g.add\_edge('7', '5')

g.add\_edge('7', '6')

# Perform DFS from node '2' to '7'

initial\_node = '2'

goal\_node = '7'

result\_path = dfs(g.graph, initial\_node, goal\_node)

# Display the result

if result\_path:

print(f"Path from {initial\_node} to {goal\_node}: {result\_path}")

else:

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

**Output:**

Path from 2 to 7: ['2', '1', '3', '4', '7']

**Slip 4:**

Q.1) Write a program to implement Hangman game using python. [10 Marks]

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

import random

def choose\_word():

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

return random.choice(word\_list)

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

display = ""

for letter in word:

if letter in guessed\_letters:

display += letter

else:

display += "\_"

return display

def hangman():

word\_to\_guess = choose\_word()

guessed\_letters = []

max\_attempts = 6

attempts = 0

print("Welcome to Hangman!")

while True:

current\_display = display\_word(word\_to\_guess, guessed\_letters)

print("\nCurrent Word:", current\_display)

if current\_display == word\_to\_guess:

print("Congratulations! You guessed the word:", word\_to\_guess)

break

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

if guess in guessed\_letters:

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

continue

guessed\_letters.append(guess)

if guess not in word\_to\_guess:

attempts += 1

print(f"Wrong guess! Attempts left: {max\_attempts - attempts}")

if attempts == max\_attempts:

print("Sorry, you ran out of attempts. The correct word was:", word\_to\_guess)

break

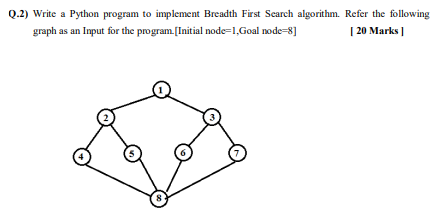
if "\_" not in display\_word(word\_to\_guess, guessed\_letters):

print("Congratulations! You guessed the word:", word\_to\_guess)

break

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

hangman()



#BFS using python

from collections import deque

class Graph:

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

self.graph = {}

def add\_edge(self, node1, node2):

# Add edges for an undirected graph

if node1 not in self.graph:

self.graph[node1] = []

if node2 not in self.graph:

self.graph[node2] = []

self.graph[node1].append(node2)

self.graph[node2].append(node1)

def bfs(graph, start, goal):

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

visited = set()

while queue:

current, path = queue.popleft()

if current == goal:

return path

if current not in visited:

visited.add(current)

for neighbor in graph[current]:

if neighbor not in visited:

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

return None

# Example Usage:

# Create an undirected graph and add edges

g = Graph()

g.add\_edge('1', '2')

g.add\_edge('1', '3')

g.add\_edge('2', '4')

g.add\_edge('2', '5')

g.add\_edge('3', '6')

g.add\_edge('3', '7')

g.add\_edge('4', '8')

g.add\_edge('5', '8')

g.add\_edge('6', '8')

g.add\_edge('7', '8')

# Perform BFS from node '1' to '8'

initial\_node = '1'

goal\_node = '8'

result\_path = bfs(g.graph, initial\_node, goal\_node)

# Display the result

if result\_path:

print(f"Path from {initial\_node} to {goal\_node}: {result\_path}")

else:

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

**Output:**

Path from 1 to 8: ['1', '2', '4', '8']

**Slip 5:**

Q.1) Write a python program to implement Lemmatization using NLTK [ 10 Marks ]

To perform lemmatization using NLTK (Natural Language Toolkit) in Python, you'll need to install the NLTK library first. You can install it using:

pip install nltk

After installing NLTK, you can use the following Python program to implement lemmatization:

import nltk

from nltk.stem import WordNetLemmatizer

from nltk.tokenize import word\_tokenize

from nltk.corpus import wordnet

nltk.download('punkt')

nltk.download('wordnet')

def get\_wordnet\_pos(tag):

if tag.startswith('N'):

return wordnet.NOUN

elif tag.startswith('V'):

return wordnet.VERB

elif tag.startswith('R'):

return wordnet.ADV

elif tag.startswith('J'):

return wordnet.ADJ

else:

return wordnet.NOUN # Default to noun if the part of speech is not recognized

def lemmatize\_text(text):

lemmatizer = WordNetLemmatizer()

tokens = word\_tokenize(text)

pos\_tags = nltk.pos\_tag(tokens)

lemmatized\_tokens = []

for token, tag in pos\_tags:

pos = get\_wordnet\_pos(tag)

lemmatized\_token = lemmatizer.lemmatize(token, pos=pos)

lemmatized\_tokens.append(lemmatized\_token)

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

return lemmatized\_text

# Example Usage:

input\_text = "The dogs are barking loudly in the garden."

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

print("Original Text:")

print(input\_text)

print("\nLemmatized Text:")

print(lemmatized\_result)

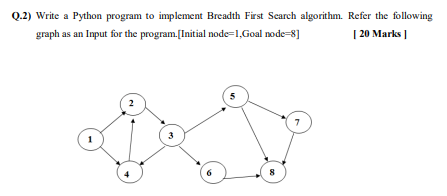
**Output:**

Original Text:

The dogs are barking loudly in the garden.

Lemmatized Text:

The dog be bark loudly in the garden .



from collections import deque

class Graph:

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

self.graph = {}

def add\_edge(self, start, end):

if start not in self.graph:

self.graph[start] = []

self.graph[start].append(end)

def bfs(graph, start, goal):

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

visited = set()

while queue:

current, path = queue.popleft()

if current == goal:

return path

if current not in visited:

visited.add(current)

for neighbor in graph.get(current, []):

if neighbor not in visited:

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

return None

# Example Usage:

# Create a directed graph and add edges

g = Graph()

g.add\_edge('1', '2')

g.add\_edge('1', '4')

g.add\_edge('2', '3')

g.add\_edge('3', '4')

g.add\_edge('3', '5')

g.add\_edge('3', '6')

g.add\_edge('4', '2')

g.add\_edge('5', '7')

g.add\_edge('5', '8')

g.add\_edge('6', '8')

g.add\_edge('7', '8')

# Perform BFS from node '1' to '8'

initial\_node = '1'

goal\_node = '8'

result\_path = bfs(g.graph, initial\_node, goal\_node)

# Display the result

if result\_path:

print(f"Path from {initial\_node} to {goal\_node}: {result\_path}")

else:

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

**Output:**

Path from 1 to 8: ['1', '2', '3', '5', '8']

**Slip 6:**

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

To remove stop words from a given passage in a text file using NLTK (Natural Language Toolkit), you'll need to install the NLTK library first. You can install it using:

pip install nltk

After installing NLTK, you can use the following Python program to remove stop words from a passage in a text file:

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

return ' '.join(filtered\_words)

def process\_file(file\_path):

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

passage = file.read()

return remove\_stop\_words(passage)

# Example Usage:

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

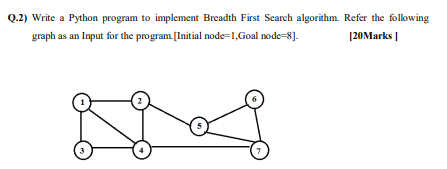
processed\_passage = process\_file(file\_path)

print("Original Passage:")

print(passage)

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

print(processed\_passage)

****

from collections import deque

class Graph:

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

self.graph = {}

def add\_edge(self, node1, node2):

# Add edges for an undirected graph

if node1 not in self.graph:

self.graph[node1] = []

if node2 not in self.graph:

self.graph[node2] = []

self.graph[node1].append(node2)

self.graph[node2].append(node1)

def bfs(graph, start, goal):

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

visited = set()

while queue:

current, path = queue.popleft()

if current == goal:

return path

if current not in visited:

visited.add(current)

for neighbor in graph[current]:

if neighbor not in visited:

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

return None

# Example Usage:

# Create an undirected graph and add edges

g = Graph()

g.add\_edge('1', '2')

g.add\_edge('1', '3')

g.add\_edge('1', '4')

g.add\_edge('2', '1')

g.add\_edge('2', '4')

g.add\_edge('3', '1')

g.add\_edge('3', '4')

g.add\_edge('4', '1')

g.add\_edge('4', '2')

g.add\_edge('4', '3')

g.add\_edge('5', '2')

g.add\_edge('5', '6')

g.add\_edge('5', '7')

g.add\_edge('6', '5')

g.add\_edge('6', '7')

g.add\_edge('7', '5')

g.add\_edge('7', '6')

g.add\_edge('7', '4')

# Perform BFS from node '1' to '8'

initial\_node = '1'

goal\_node = '8'

result\_path = bfs(g.graph, initial\_node, goal\_node)

# Display the result

if result\_path:

print(f"Path from {initial\_node} to {goal\_node}: {result\_path}")

else:

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

**Output:**

No path found from 1 to 8

**Slip 7**

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

**import** random

*# Function to initialize the board*

**def** initialize\_board():

**return** [' ' **for** \_ **in** range(9)] *# A list of 9 spaces*

*# Function to print the board*

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

print("\n")

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]} ")

print("\n")

*# Function to check for a win*

**def** check\_winner(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*

]

**return** any(all(board[i] **==** player **for** i **in** condition) **for** condition **in** win\_conditions)

*# Function to check for a draw*

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

**return** all(space **!=** ' ' **for** space **in** board)

*# Function to get a list of available moves*

**def** available\_moves(board):

**return** [i **for** i, space **in** enumerate(board) **if** space **==** ' ']

*# Function for the computer's move (random selection)*

**def** computer\_move(board):

move **=** random**.**choice(available\_moves(board))

board[move] **=** 'O'

*# Function for the player's move*

**def** player\_move(board):

**while** **True**:

**try**:

move **=** int(input("Enter your move (1-9): ")) **-** 1

**if** move **in** available\_moves(board):

board[move] **=** 'X'

**break**

**else**:

print("Invalid move. Try again.")

**except** ValueError:

print("Invalid input. Please enter a number between 1 and 9.")

*# Main game loop*

**def** play\_game():

board **=** initialize\_board()

print\_board(board)

**while** **True**:

*# Player's move*

player\_move(board)

print\_board(board)

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

print("Congratulations! You win!")

**break**

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

print("It's a draw!")

**break**

*# Computer's move*

computer\_move(board)

print\_board(board)

**if** check\_winner(board, 'O'):

print("Computer wins! Better luck next time.")

**break**

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

print("It's a draw!")

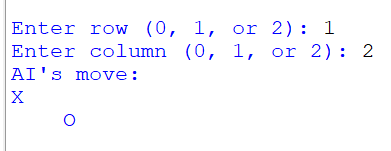
**break**

*# Start the game*

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

play\_game()

output:



Q.2) Write a Python program to implement Simple Chatbot. [ 20 Marks ]

import random

def simple\_chatbot(user\_input):

greetings = ["hello", "hi", "hey", "greetings", "howdy"]

goodbyes = ["bye", "goodbye", "see you", "farewell"]

questions = ["how are you", "what's your name", "how's the weather", "tell me a joke"]

user\_input = user\_input.lower()

if any(greeting in user\_input for greeting in greetings):

return "Hello! How can I help you?"

elif any(goodbye in user\_input for goodbye in goodbyes):

return "Goodbye! Have a great day."

elif any(question in user\_input for question in questions):

return respond\_to\_question(user\_input)

else:

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

def respond\_to\_question(question):

if "how are you" in question:

return "I'm just a chatbot, but I'm doing well. Thanks for asking!"

elif "what's your name" in question:

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

elif "how's the weather" in question:

return "I'm sorry, I don't have real-time weather information. You can check a weather website."

elif "tell me a joke" in question:

jokes = ["Why did the computer go to therapy? It had too many bytes of emotional baggage!",

"Why don't scientists trust atoms? Because they make up everything!"]

return random.choice(jokes)

else:

return "I'm not sure how to respond to that. Ask me something else!"

def main():

print("Simple Chatbot: Hi there! Ask me anything or just say hello. Type 'exit' to end the conversation.")

while True:

user\_input = input("You: ")

if user\_input.lower() == 'exit':

print("Simple Chatbot: Goodbye!")

break

response = simple\_chatbot(user\_input)

print("Simple Chatbot:", response)

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

main()

**Output:**

Simple Chatbot: Hi there! Ask me anything or just say hello. Type 'exit' to end the conversation.

You: How are you?

Simple Chatbot: I'm just a chatbot, but I'm doing well. Thanks for asking!

You: What is your name?

Simple Chatbot: I'm sorry, I didn't understand that. Can you please rephrase?

You:

**Slip 8:**

Q.1) Write a Python program to accept a string. Find and print the number of upper case alphabets and lower case alphabets. [ 10 Marks ]

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

upper\_count = 0

lower\_count = 0

for char in input\_string:

if char.isupper():

upper\_count += 1

elif char.islower():

lower\_count += 1

return upper\_count, lower\_count

def main():

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

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

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

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

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

main()

**Output:**

Enter a string: haggjGHJYJ

Number of uppercase alphabets: 5

Number of lowercase alphabets: 5

Q.2) Write a Python program to solve tic-tac-toe problem. [ 20 Marks ]

**import** random

*# Function to initialize the board*

**def** initialize\_board():

**return** [' ' **for** \_ **in** range(9)] *# A list of 9 spaces*

*# Function to print the board*

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

print("\n")

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]} ")

print("\n")

*# Function to check for a win*

**def** check\_winner(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*

]

**return** any(all(board[i] **==** player **for** i **in** condition) **for** condition **in** win\_conditions)

*# Function to check for a draw*

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

**return** all(space **!=** ' ' **for** space **in** board)

*# Function to get a list of available moves*

**def** available\_moves(board):

**return** [i **for** i, space **in** enumerate(board) **if** space **==** ' ']

*# Function for the computer's move (random selection)*

**def** computer\_move(board):

move **=** random**.**choice(available\_moves(board))

board[move] **=** 'O'

*# Function for the player's move*

**def** player\_move(board):

**while** **True**:

**try**:

move **=** int(input("Enter your move (1-9): ")) **-** 1

**if** move **in** available\_moves(board):

board[move] **=** 'X'

**break**

**else**:

print("Invalid move. Try again.")

**except** ValueError:

print("Invalid input. Please enter a number between 1 and 9.")

*# Main game loop*

**def** play\_game():

board **=** initialize\_board()

print\_board(board)

**while** **True**:

*# Player's move*

player\_move(board)

print\_board(board)

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

print("Congratulations! You win!")

**break**

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

print("It's a draw!")

**break**

*# Computer's move*

computer\_move(board)

print\_board(board)

**if** check\_winner(board, 'O'):

print("Computer wins! Better luck next time.")

**break**

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

print("It's a draw!")

**break**

*# Start the game*

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

play\_game()

**Output:**

Enter row (0, 1, or 2): 2

Enter column (0, 1, or 2): 2

AI's move:

X

O

**Slip 9:**

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

import heapq

class PuzzleNode:

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

self.state = state

self.parent = parent

self.action = action

self.cost = cost

self.heuristic = heuristic

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

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

def print\_puzzle(state):

for i in range(3):

for j in range(3):

print(state[i \* 3 + j], end=" ")

print()

def find\_blank(state):

return state.index(0)

def is\_goal\_state(state):

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

def get\_neighbors(state):

blank\_index = find\_blank(state)

neighbors = []

# Move blank space left

if blank\_index % 3 != 0:

new\_state = state[:]

new\_state[blank\_index], new\_state[blank\_index - 1] = new\_state[blank\_index - 1], new\_state[blank\_index]

neighbors.append(new\_state)

# Move blank space right

if blank\_index % 3 != 2:

new\_state = state[:]

new\_state[blank\_index], new\_state[blank\_index + 1] = new\_state[blank\_index + 1], new\_state[blank\_index]

neighbors.append(new\_state)

# Move blank space up

if blank\_index >= 3:

new\_state = state[:]

new\_state[blank\_index], new\_state[blank\_index - 3] = new\_state[blank\_index - 3], new\_state[blank\_index]

neighbors.append(new\_state)

# Move blank space down

if blank\_index < 6:

new\_state = state[:]

new\_state[blank\_index], new\_state[blank\_index + 3] = new\_state[blank\_index + 3], new\_state[blank\_index]

neighbors.append(new\_state)

return neighbors

def manhattan\_distance(state):

distance = 0

for i in range(3):

for j in range(3):

value = state[i \* 3 + j]

if value != 0:

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

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

return distance

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

start\_node = PuzzleNode(initial\_state, None, None, 0, manhattan\_distance(initial\_state))

priority\_queue = [start\_node]

while priority\_queue:

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

if is\_goal\_state(current\_node.state):

path = []

while current\_node.parent is not None:

path.append(current\_node.state)

current\_node = current\_node.parent

path.append(current\_node.state)

return reversed(path)

for neighbor\_state in get\_neighbors(current\_node.state):

neighbor\_node = PuzzleNode(

neighbor\_state,

current\_node,

None, # Action not needed for 8-puzzle

current\_node.cost + 1,

manhattan\_distance(neighbor\_state)

)

heapq.heappush(priority\_queue, neighbor\_node)

return None

def main():

# Example initial state

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

print("Initial State:")

print\_puzzle(initial\_state)

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

if solution:

print("\nSolution:")

for step, state in enumerate(solution):

print(f"Step {step + 1}:")

print\_puzzle(state)

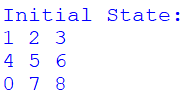
else:

print("\nNo solution found.")

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

main()

Output:



Q.2) Write a Python program to solve water jug problem. 2 jugs with capacity 5 gallon and 7 gallon are given with unlimited water supply respectively. The target to achieve is 4 gallon of water in second jug. [ 15 Marks ]

**from** collections **import** deque

*# Function to check if a state has been visited before*

**def** is\_visited(visited, state):

**return** visited**.**get(state, **False**)

*# Function to mark a state as visited*

**def** mark\_visited(visited, state):

visited[state] **=** **True**

*# Function to implement the BFS solution to the Water Jug Problem*

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

*# Queue to store the states (jug1, jug2) and the operations taken*

queue **=** deque()

visited **=** {} *# Dictionary to keep track of visited states*

*# Initial state (both jugs empty)*

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

queue**.**append((initial\_state, []))

mark\_visited(visited, initial\_state)

**while** queue:

*# Get the current state and the list of operations taken to reach it*

(jug1, jug2), operations **=** queue**.**popleft()

*# If the target is reached in either jug, return the operations*

**if** jug1 **==** target **or** jug2 **==** target:

**return** operations

*# Possible operations from the current state*

possible\_operations **=** [

((jug1\_capacity, jug2), "Fill Jug1"), *# Fill Jug1*

((jug1, jug2\_capacity), "Fill Jug2"), *# Fill Jug2*

((0, jug2), "Empty Jug1"), *# Empty Jug1*

((jug1, 0), "Empty Jug2"), *# Empty Jug2*

((min(jug1 **+** jug2, jug1\_capacity), max(0, jug2 **-** (jug1\_capacity **-** jug1))), "Pour Jug2 to Jug1"), *# Pour Jug2 to Jug1*

((max(0, jug1 **-** (jug2\_capacity **-** jug2)), min(jug1 **+** jug2, jug2\_capacity)), "Pour Jug1 to Jug2") *# Pour Jug1 to Jug2*

]

*# Process each possible operation*

**for** new\_state, operation **in** possible\_operations:

**if** **not** is\_visited(visited, new\_state):

mark\_visited(visited, new\_state)

queue**.**append((new\_state, operations **+** [operation]))

*# If no solution is found, return None*

**return** **None**

*# Driver code*

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

result **=** water\_jug\_bfs(jug1\_capacity, jug2\_capacity, target)

**if** result:

print("Steps to achieve the target:")

**for** step **in** result:

print(step)

**else**:

print("No solution exists.")

*# Example usage:*

solve\_water\_jug\_problem(4, 3, 2)

**Output:**

Steps to achieve the target:

Fill Jug2

Pour Jug2 to Jug1

Fill Jug2

Pour Jug2 to Jug1

**Slip 10:**

Q.1) Write Python program to implement crypt arithmetic problem TWO+TWO=FOUR [ 10 Marks ]

from itertools import permutations

def is\_valid\_assignment(assignment):

t, w, o, f, u, r = assignment

return 2 \* (100 \* t + 10 \* w + o) == 1000 \* f + 100 \* o + 10 \* u + r

def solve\_cryptarithmetic():

for permutation in permutations(range(10), 6):

assignment = tuple(permutation)

if assignment[0] != 0: # Ensure that T and F are not assigned 0

if is\_valid\_assignment(assignment):

return assignment

return None

def print\_solution(assignment):

t, w, o, f, u, r = assignment

print(f"T = {t}")

print(f"W = {w}")

print(f"O = {o}")

print(f"F = {f}")

print(f"U = {u}")

print(f"R = {r}")

def main():

solution = solve\_cryptarithmetic()

if solution:

print("Cryptarithmetic Puzzle Solved:")

print\_solution(solution)

else:

print("No solution found.")

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

main()

Output:

Cryptarithmetic Puzzle Solved:

T = 1

W = 3

O = 2

F = 0

U = 6

R = 4

Q.2) Write a Python program to implement Simple Chatbot. [ 20 Marks ]

import random

def simple\_chatbot(user\_input):

greetings = ["hello", "hi", "hey", "greetings", "howdy"]

goodbyes = ["bye", "goodbye", "see you", "farewell"]

questions = ["how are you", "what's your name", "how's the weather", "tell me a joke"]

user\_input = user\_input.lower()

if any(greeting in user\_input for greeting in greetings):

return "Hello! How can I help you?"

elif any(goodbye in user\_input for goodbye in goodbyes):

return "Goodbye! Have a great day."

elif any(question in user\_input for question in questions):

return respond\_to\_question(user\_input)

else:

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

def respond\_to\_question(question):

if "how are you" in question:

return "I'm just a chatbot, but I'm doing well. Thanks for asking!"

elif "what's your name" in question:

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

elif "how's the weather" in question:

return "I'm sorry, I don't have real-time weather information. You can check a weather website."

elif "tell me a joke" in question:

jokes = ["Why did the computer go to therapy? It had too many bytes of emotional baggage!",

"Why don't scientists trust atoms? Because they make up everything!"]

return random.choice(jokes)

else:

return "I'm not sure how to respond to that. Ask me something else!"

def main():

print("Simple Chatbot: Hi there! Ask me anything or just say hello. Type 'exit' to end the conversation.")

while True:

user\_input = input("You: ")

if user\_input.lower() == 'exit':

print("Simple Chatbot: Goodbye!")

break

response = simple\_chatbot(user\_input)

print("Simple Chatbot:", response)

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

main()

**Output:**

Simple Chatbot: Hi there! Ask me anything or just say hello. Type 'exit' to end the conversation.

You: How are you?

Simple Chatbot: I'm just a chatbot, but I'm doing well. Thanks for asking!

You: What is your name?

Simple Chatbot: I'm sorry, I didn't understand that. Can you please rephrase?

You:

**Slip 11**

Q.1) Write a python program using mean end analysis algorithm problem of transforming a string of lowercase letters into another string. [ 10 Marks ]

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

current\_state = list(initial)

goal\_state = list(goal)

steps = []

while current\_state != goal\_state:

for i in range(len(current\_state)):

if current\_state[i] != goal\_state[i]:

steps.append(f"Change {current\_state[i]} to {goal\_state[i]} at position {i + 1}")

current\_state[i] = goal\_state[i]

return steps

def main():

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

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

if len(initial\_string) != len(goal\_string):

print("Error: The lengths of the initial and goal strings should be the same.")

return

transformation\_steps = mean\_end\_analysis(initial\_string, goal\_string)

if transformation\_steps:

print("\nTransformation Steps:")

for step in transformation\_steps:

print(step)

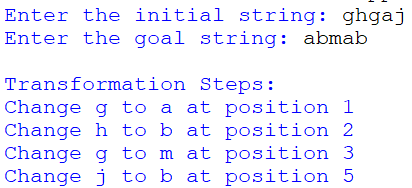
else:

print("\nThe strings are already the same.")

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

main()

Output:



Q.2) 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. [ 20 Marks ]

**from** collections **import** deque

*# Function to check if a state has been visited before*

**def** is\_visited(visited, state):

**return** visited**.**get(state, **False**)

*# Function to mark a state as visited*

**def** mark\_visited(visited, state):

visited[state] **=** **True**

*# Function to implement the BFS solution to the Water Jug Problem*

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

*# Queue to store the states (jug1, jug2) and the operations taken*

queue **=** deque()

visited **=** {} *# Dictionary to keep track of visited states*

*# Initial state (both jugs empty)*

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

queue**.**append((initial\_state, []))

mark\_visited(visited, initial\_state)

**while** queue:

*# Get the current state and the list of operations taken to reach it*

(jug1, jug2), operations **=** queue**.**popleft()

*# If the target is reached in either jug, return the operations*

**if** jug1 **==** target **or** jug2 **==** target:

**return** operations

*# Possible operations from the current state*

possible\_operations **=** [

((jug1\_capacity, jug2), "Fill Jug1"), *# Fill Jug1*

((jug1, jug2\_capacity), "Fill Jug2"), *# Fill Jug2*

((0, jug2), "Empty Jug1"), *# Empty Jug1*

((jug1, 0), "Empty Jug2"), *# Empty Jug2*

((min(jug1 **+** jug2, jug1\_capacity), max(0, jug2 **-** (jug1\_capacity **-** jug1))), "Pour Jug2 to Jug1"), *# Pour Jug2 to Jug1*

((max(0, jug1 **-** (jug2\_capacity **-** jug2)), min(jug1 **+** jug2, jug2\_capacity)), "Pour Jug1 to Jug2") *# Pour Jug1 to Jug2*

]

*# Process each possible operation*

**for** new\_state, operation **in** possible\_operations:

**if** **not** is\_visited(visited, new\_state):

mark\_visited(visited, new\_state)

queue**.**append((new\_state, operations **+** [operation]))

*# If no solution is found, return None*

**return** **None**

*# Driver code*

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

result **=** water\_jug\_bfs(jug1\_capacity, jug2\_capacity, target)

**if** result:

print("Steps to achieve the target:")

**for** step **in** result:

print(step)

**else**:

print("No solution exists.")

*# Example usage:*

solve\_water\_jug\_problem(4, 3, 2)

Output: Steps to achieve the target:

Fill Jug2

Pour Jug2 to Jug1

Fill Jug2

Pour Jug2 to Jug1

**Slip 12:**

Q.1) Write a python program to generate Calendar for the given month and year?. [ 10Marks ]

**import calendar**

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

**# Create a TextCalendar instance**

**cal = calendar.TextCalendar(calendar.SUNDAY)**

**# Generate the calendar for the given month and year**

**calendar\_text = cal.formatmonth(year, month)**

**# Print the calendar**

**print(calendar\_text)**

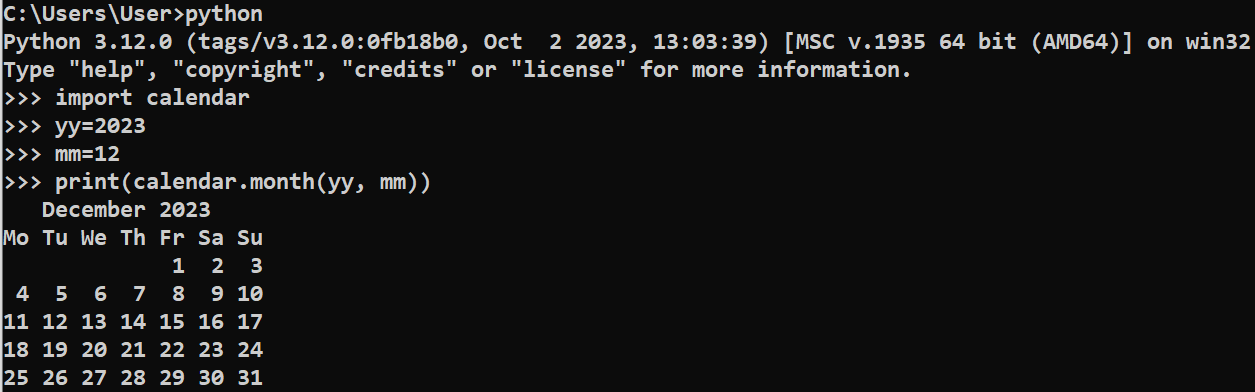
**# Get user input for the year and month**

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

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

**# Generate and print the calendar**

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



Q.2) Write a Python program to simulate 4-Queens problem. [ 20Marks ]

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

# Check if there is a queen in the same column

for i in range(row):

if board[i] == col or board[i] - i == col - row or board[i] + i == col + row:

return False

return True

def print\_solution(board):

for row in range(len(board)):

line = ""

for col in range(len(board)):

line += "Q" if board[row] == col else "."

print(line)

print()

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

if row == len(board):

print\_solution(board)

return

for col in range(len(board)):

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

board[row] = col

solve\_n\_queens\_util(board, row + 1)

board[row] = -1

def solve\_n\_queens(n):

board = [-1] \* n

solve\_n\_queens\_util(board, 0)

def main():

n = 4 # Change this to the desired number of queens

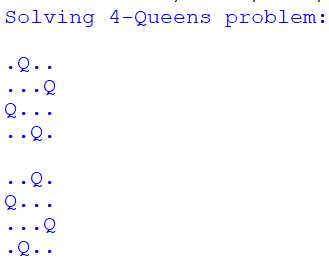
print(f"Solving {n}-Queens problem:\n")

solve\_n\_queens(n)

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

main()

**Output:**

****

**Slip 13:**

Q.1Write a Python program to implement Mini-Max Algorithm. [ 10 Marks ]

def print\_board(board):

for row in board:

print(" ".join(row))

print()

def evaluate(board):

# Check rows, columns, and diagonals for a win

for row in board:

if all(cell == row[0] and cell != ' ' for cell in row):

return 1 if row[0] == 'X' else -1

for col in range(3):

if all(board[row][col] == board[0][col] and board[row][col] != ' ' for row in range(3)):

return 1 if board[0][col] == 'X' else -1

if all(board[i][i] == board[0][0] and board[i][i] != ' ' for i in range(3)):

return 1 if board[0][0] == 'X' else -1

if all(board[i][2 - i] == board[0][2] and board[i][2 - i] != ' ' for i in range(3)):

return 1 if board[0][2] == 'X' else -1

return 0 # No winner

def is\_full(board):

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

def get\_empty\_cells(board):

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

def mini\_max(board, depth, is\_maximizing):

score = evaluate(board)

if score == 1:

return score - depth

if score == -1:

return score + depth

if is\_full(board):

return 0

if is\_maximizing:

max\_eval = float('-inf')

for i, j in get\_empty\_cells(board):

board[i][j] = 'X'

eval = mini\_max(board, depth + 1, False)

board[i][j] = ' '

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

return max\_eval

else:

min\_eval = float('inf')

for i, j in get\_empty\_cells(board):

board[i][j] = 'O'

eval = mini\_max(board, depth + 1, True)

board[i][j] = ' '

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

return min\_eval

def find\_best\_move(board):

best\_val = float('-inf')

best\_move = (-1, -1)

for i, j in get\_empty\_cells(board):

board[i][j] = 'X'

move\_val = mini\_max(board, 0, False)

board[i][j] = ' '

if move\_val > best\_val:

best\_move = (i, j)

best\_val = move\_val

return best\_move

def main():

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

while True:

print\_board(board)

# Player's move

row = int(input("Enter row (0, 1, or 2): "))

col = int(input("Enter column (0, 1, or 2): "))

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

print("Cell already taken. Try again.")

continue

board[row][col] = 'O'

# Check for player win

if evaluate(board) == -1:

print\_board(board)

print("You win!")

break

# Check for a draw

if is\_full(board):

print\_board(board)

print("It's a draw!")

break

# AI's move

print("AI's move:")

ai\_row, ai\_col = find\_best\_move(board)

board[ai\_row][ai\_col] = 'X'

# Check for AI win

if evaluate(board) == 1:

print\_board(board)

print("AI wins!")

break

# Check for a draw

if is\_full(board):

print\_board(board)

print("It's a draw!")

break

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

main()

**Output:**

Enter row (0, 1, or 2): 2

Enter column (0, 1, or 2): 2

AI's move:

X

O

Q.2) Write a Python program to simulate 8-Queens problem. [ 20 Marks]

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

# Check if there is a queen in the same row

if any(board[row][c] == 1 for c in range(col)):

return False

# Check upper diagonal on the left side

if any(board[i][j] == 1 for i, j in zip(range(row, -1, -1), range(col, -1, -1))):

return False

# Check lower diagonal on the left side

if any(board[i][j] == 1 for i, j in zip(range(row, len(board), 1), range(col, -1, -1))):

return False

return True

def print\_solution(board):

for row in board:

print(" ".join("Q" if cell == 1 else "." for cell in row))

print()

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

if col == len(board):

print\_solution(board)

return

for row in range(len(board)):

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

board[row][col] = 1

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

board[row][col] = 0

def solve\_n\_queens(n):

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

solve\_n\_queens\_util(board, 0)

def main():

n = 8 # Change this to the desired number of queens

print(f"Solving {n}-Queens problem:\n")

solve\_n\_queens(n)

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

main()

Output:

Solving 8-Queens problem:

Q . . . . . . .

. . . . . . Q .

. . . . Q . . .

. . . . . . . Q

. Q . . . . . .

. . . Q . . . .

. . . . . Q . .

. . Q . . . . .

**Slip 14:**

Q.1) Write a python program to sort the sentence in alphabetical order? [ 10Marks ]

def sort\_sentence(sentence):

words = sentence.split()

sorted\_words = sorted(words)

sorted\_sentence = " ".join(sorted\_words)

return sorted\_sentence

def main():

input\_sentence = input("Enter a sentence: ")

sorted\_sentence = sort\_sentence(input\_sentence)

print("\nSorted Sentence:")

print(sorted\_sentence)

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

main()

Output:

Enter a sentence: These are AI solutions

Sorted Sentence:

AI These are solutions

Q.2) Write a Python program to simulate n-Queens problem. [ 20Marks ]

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

# Check if there is a queen in the same row

if any(board[row][c] == 1 for c in range(col)):

return False

# Check upper diagonal on the left side

if any(board[i][j] == 1 for i, j in zip(range(row, -1, -1), range(col, -1, -1))):

return False

# Check lower diagonal on the left side

if any(board[i][j] == 1 for i, j in zip(range(row, n, 1), range(col, -1, -1))):

return False

return True

def print\_solution(board):

for row in board:

print(" ".join("Q" if cell == 1 else "." for cell in row))

print()

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

if col == n:

print\_solution(board)

return

for row in range(n):

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

board[row][col] = 1

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

board[row][col] = 0

def solve\_n\_queens(n):

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

solve\_n\_queens\_util(board, 0, n)

def main():

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

print(f"\nSolving {n}-Queens problem:\n")

solve\_n\_queens(n)

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

main()

**Slip 15**

Q.1) Write a Program to Implement Monkey Banana Problem using Python [ 10 Marks ]

from queue import Queue

def monkey\_banana\_problem(room):

rows, cols = len(room), len(room[0])

# Find the initial position of the monkey and bananas

for i in range(rows):

for j in range(cols):

if room[i][j] == 'M':

monkey\_start = (i, j)

elif room[i][j] == 'B':

banana\_position = (i, j)

# Define possible moves: up, down, left, right

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

# Initialize the visited set to keep track of visited cells

visited = set()

# Initialize the queue for BFS

queue = Queue()

queue.put((monkey\_start, 0)) # Tuple of (current position, time taken)

while not queue.empty():

current\_position, time\_taken = queue.get()

x, y = current\_position

# Check if the monkey has reached the bananas

if current\_position == banana\_position:

return time\_taken

# Explore neighboring cells

for move in moves:

new\_x, new\_y = x + move[0], y + move[1]

# Check if the new position is within bounds and not a wall

if 0 <= new\_x < rows and 0 <= new\_y < cols and room[new\_x][new\_y] != 'W' and (new\_x, new\_y) not in visited:

queue.put(((new\_x, new\_y), time\_taken + 1))

visited.add((new\_x, new\_y))

# If the monkey cannot reach the bananas

return -1

def main():

# Example room layout (M for Monkey, B for Banana, W for Wall, and . for empty cell)

room = [

['M', '.', '.', 'W', 'B'],

['.', 'W', '.', '.', '.'],

['.', '.', '.', 'W', '.'],

['.', '.', '.', '.', '.']

]

time\_taken = monkey\_banana\_problem(room)

if time\_taken != -1:

print(f"The monkey can reach the bananas in {time\_taken} units of time.")

else:

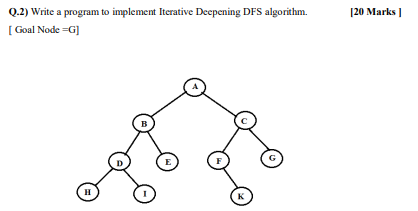
print("The monkey cannot reach the bananas.")

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

main()

**Output:**

The monkey can reach the bananas in 6 units of time.



class Node:

def \_\_init\_\_(self, name):

self.name = name

self.children = []

def add\_child(self, child):

self.children.append(child)

def iterative\_deepening\_dfs(root, goal):

depth\_limit = 0

while True:

result = depth\_limited\_dfs(root, goal, depth\_limit)

if result == goal:

return result

depth\_limit += 1

def depth\_limited\_dfs(node, goal, depth\_limit):

return recursive\_dls(node, goal, depth\_limit, 0)

def recursive\_dls(node, goal, depth\_limit, current\_depth):

if current\_depth > depth\_limit:

return None

if node.name == goal:

return node.name

for child in node.children:

result = recursive\_dls(child, goal, depth\_limit, current\_depth + 1)

if result:

return result

return None

def build\_sample\_tree():

# Building a sample tree

A = Node('A')

B = Node('B')

C = Node('C')

D = Node('D')

E = Node('E')

F = Node('F')

G = Node('G')

H = Node('H')

I = Node('I')

K = Node('K')

A.add\_child(B)

A.add\_child(C)

B.add\_child(D)

B.add\_child(E)

C.add\_child(F)

C.add\_child(G)

D.add\_child(H)

D.add\_child(I)

F.add\_child(K)

return A # The root node

def main():

root = build\_sample\_tree()

goal\_node = 'G'

result = iterative\_deepening\_dfs(root, goal\_node)

if result:

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

else:

print(f"Goal node '{goal\_node}' not found.")

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

main()

**Output:**

Goal node 'G' found!

**Slip 18**

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)

import heapq

class Object:

def \_\_init\_\_(self, name, width, height):

self.name = name

self.width = width

self.height = height

class Room:

def \_\_init\_\_(self, width, height):

self.width = width

self.height = height

class State:

def \_\_init\_\_(self, room, objects):

self.room = room

self.objects = objects

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

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

def cost(self):

# A\* cost function (heuristic + cost from start)

return self.heuristic() + len(self.objects)

def heuristic(self):

# Simple heuristic: total area of remaining empty space

used\_area = sum(obj.width \* obj.height for obj in self.objects)

empty\_area = max(0, self.room.width \* self.room.height - used\_area)

return empty\_area

def is\_valid(state):

# Check if the state is valid (objects do not overlap and fit in the room)

occupied\_area = set()

for obj in state.objects:

for i in range(obj.width):

for j in range(obj.height):

if (i + obj.x, j + obj.y) in occupied\_area:

return False

occupied\_area.add((i + obj.x, j + obj.y))

return True

def get\_actions(state):

# Generate possible actions (move and rotate objects)

actions = []

for obj in state.objects:

for dx in range(-obj.width + 1, state.room.width):

for dy in range(-obj.height + 1, state.room.height):

new\_obj = Object(obj.name, obj.width, obj.height)

new\_obj.x, new\_obj.y = dx, dy

new\_objects = [o if o.name != obj.name else new\_obj for o in state.objects]

new\_state = State(state.room, new\_objects)

if is\_valid(new\_state):

actions.append(new\_state)

# Rotate the object

new\_obj = Object(obj.name, obj.height, obj.width)

new\_objects = [o if o.name != obj.name else new\_obj for o in state.objects]

new\_state = State(state.room, new\_objects)

if is\_valid(new\_state):

actions.append(new\_state)

return actions

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

heap = [initial\_state]

visited = set()

while heap:

current\_state = heapq.heappop(heap)

if current\_state.heuristic() == 0:

return current\_state # Goal state reached

visited.add(tuple(current\_state.objects))

for action in get\_actions(current\_state):

if tuple(action.objects) not in visited:

heapq.heappush(heap, action)

return None # No solution found

def main():

# Define the room and objects

room = Room(width=10, height=10)

rectangular\_objects = [Object("R1", 2, 4), Object("R2", 3, 5), Object("R3", 2, 3), Object("R4", 4, 2), Object("R5", 3, 3)]

square\_objects = [Object("S1", 2, 2), Object("S2", 3, 3), Object("S3", 4, 4), Object("S4", 2, 2)]

initial\_state = State(room, rectangular\_objects + square\_objects)

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

if result\_state:

print("Arrangement found:")

for obj in result\_state.objects:

print(f"{obj.name} at position ({obj.x}, {obj.y})")

else:

print("No valid arrangement found.")

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

main()

**Slip 20:**

Q.1) Build a bot which provides all the information related to you in college [ 10Marks ]

This example assumes a static dataset for simplicity, but in a real-world scenario, you might need to connect to databases, APIs, or other data sources.

First, you would need to install Flask if you haven't already:

pip install flask

Now, you can create a simple Flask app for your college bot:

from flask import Flask, request

app = Flask(\_\_name\_\_)

# Mock data (replace with real data or connect to databases/APIs)

college\_information = {

"name": "Your College",

"location": "City, Country",

"established\_year": 2000,

"programs": ["Computer Science", "Business Administration", "Engineering"],

"facilities": ["Library", "Sports Complex", "Cafeteria"],

}

@app.route('/')

def home():

return "Welcome to the College Bot! Ask me anything about your college."

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

def college\_info():

data = request.json

query = data.get('query', '').lower()

if 'name' in query:

return college\_information['name']

elif 'location' in query:

return college\_information['location']

elif 'established' in query:

return str(college\_information['established\_year'])

elif 'programs' in query:

return ', '.join(college\_information['programs'])

elif 'facilities' in query:

return ', '.join(college\_information['facilities'])

else:

return "I'm sorry, I don't understand the query."

if \_\_name\_\_ == '\_\_main\_\_':

app.run(debug=True)

This simple bot provides information about the college based on certain keywords.

For example:

* Send a POST request to http://127.0.0.1:5000/college\_info with JSON data {"query": "name"} to get the college name.
* Replace "name" in the query with other keywords like "location", "established", "programs", or "facilities" to get relevant information.

**Slip 21:**

Q.2)Write a Python program for the following Cryptarithmetic problems. [ 20 Marks ]

GO + TO = OUT

from itertools import permutations

def cryptarithmetic\_puzzle\_solver():

for p in permutations(range(10), 7):

# Assigning digits to letters

g, o, t, u = p[0], p[1], p[2], p[3]

n, s = p[4], p[5]

# Avoid assignments with leading zeros

if g == 0 or t == 0 or o == 0 or u == 0 or n == 0:

continue

# Evaluating the equation

go = g \* 10 + o

to = t \* 10 + o

out = o \* 100 + u \* 10 + t

if go + to == out:

return {'G': g, 'O': o, 'T': t, 'U': u, 'N': n, 'S': s}

return None

def main():

solution = cryptarithmetic\_puzzle\_solver()

if solution:

print("Solution found:")

print(f"{solution['G']} {solution['O']} + {solution['T']} {solution['O']} = {solution['O']} {solution['U']} {solution['T']}")

else:

print("No solution found.")

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

main()

**Slip 23:**

Q.1) Write a Program to Implement Tower of Hanoi using Python. [ 10 Marks ]

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

if n == 1:

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

return

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

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

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

def main():

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

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

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

main()

Q.2) Write a Python program for the following Cryptarithmetic problems SEND + MORE = MONEY

[ 20 Marks]

from itertools import permutations

def cryptarithmetic\_puzzle\_solver():

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

# Assigning digits to letters

s, e, n, d, m, o, r, y = p

# Avoid assignments with leading zeros

if s == 0 or m == 0:

continue

# Evaluating the equation

send = s \* 1000 + e \* 100 + n \* 10 + d

more = m \* 1000 + o \* 100 + r \* 10 + e

money = m \* 10000 + o \* 1000 + n \* 100 + e \* 10 + y

if send + more == money:

return {'S': s, 'E': e, 'N': n, 'D': d, 'M': m, 'O': o, 'R': r, 'Y': y}

return None

def main():

solution = cryptarithmetic\_puzzle\_solver()

if solution:

print("Solution found:")

print(f"{solution['S']}{solution['E']}{solution['N']}{solution['D']} + "

f"{solution['M']}{solution['O']}{solution['R']}{solution['E']} = "

f"{solution['M']}{solution['O']}{solution['N']}{solution['E']}{solution['Y']}")

else:

print("No solution found.")

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

main()

Slip 24

Q.2) Write a Python program for the following Cryptorithmetic problems CROSS+ROADS = DANGER

from itertools import permutations

def cryptarithmetic\_puzzle\_solver():

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

# Assigning digits to letters

c, r, o, s, a, d, n, g, e = p

# Avoid assignments with leading zeros

if c == 0 or r == 0 or d == 0:

continue

# Evaluating the equation

cross = c \* 10000 + r \* 1000 + o \* 100 + s \* 10 + s

roads = r \* 10000 + o \* 1000 + a \* 100 + d \* 10 + s

danger = d \* 100000 + a \* 10000 + n \* 1000 + g \* 100 + e \* 10 + r

if cross + roads == danger:

return {'C': c, 'R': r, 'O': o, 'S': s, 'A': a, 'D': d, 'N': n, 'G': g, 'E': e}

return None

def main():

solution = cryptarithmetic\_puzzle\_solver()

if solution:

print("Solution found:")

print(f"{solution['C']}{solution['R']}{solution['O']}{solution['S']}{solution['S']} + "

f"{solution['R']}{solution['O']}{solution['A']}{solution['D']}{solution['S']} = "

f"{solution['D']}{solution['A']}{solution['N']}{solution['G']}{solution['E']}{solution['R']}")

else:

print("No solution found.")

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

main()