Name : Aoun Muhammad

Roll No : 045

Lab task 1to 8

Lab :01

movies = [

    ("Eternal Sunshine of the Spotless Mind", 20000000),

    ("Memento", 9000000),

    ("Requiem for a Dream", 4500000),

    ("Pirates of the Caribbean: On Stranger Tides", 379000000),

    ("Avengers: Age of Ultron", 365000000),

    ("Avengers: Endgame", 356000000),

    ("Incredibles 2", 200000000)

]

def calculate\_average\_budget(movie\_list):

    total\_budget = sum([budget for name, budget in movie\_list])

    average\_budget = total\_budget / len(movie\_list)

    return average\_budget

def print\_high\_budget\_movies(movie\_list, average\_budget):

    print(f"\nMovies with a budget higher than the average ({average\_budget:.2f}):")

    count = 0

    for name, budget in movie\_list:

        if budget > average\_budget:

            difference = budget - average\_budget

            print(f"{name} (Budget: {budget}): Exceeds the average by {difference}")

            count += 1

    return count

def add\_movies():

    num\_movies = int(input("\nHow many movies would you like to add? "))

    for \_ in range(num\_movies):

        name = input("Enter the movie name: ")

        budget = int(input(f"Enter the budget for {name}: "))

        movies.append((name, budget))

print("Initial Movie Dataset: ", movies)

add\_more = input("\nWould you like to add more movies to the dataset? (yes/no): ").lower()

if add\_more == 'yes':

    add\_movies()

average\_budget = calculate\_average\_budget(movies)

print(f"\nThe average budget of the movies is: {average\_budget:.2f}")

high\_budget\_count = print\_high\_budget\_movies(movies, average\_budget)

print(f"\nNumber of movies with a budget higher than the average: {high\_budget\_count}")

Lab No 2

class SmartHome:

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

        self.room\_temps = room\_temps

    def sense\_temperature(self):

        print("Sensing current temperatures in all rooms...")

        for room, temp in self.room\_temps.items():

            print(f"{room}: {temp}°C")

    def control\_heater(self):

        print("\nChecking room temperatures and controlling heaters...")

        for room, temp in self.room\_temps.items():

            if temp < 22:

                print(f"{room}: Temperature is {temp}°C. Turning the heater ON.")

            else:

                print(f"{room}: Temperature is {temp}°C. Turning the heater OFF.")

    def update\_temperature(self, room, new\_temp):

        if room in self.room\_temps:

            self.room\_temps[room] = new\_temp

            print(f"\nTemperature of {room} updated to {new\_temp}°C.")

        else:

            print(f"Room '{room}' not found in the system.")

rooms = {

    "Living Room": 19,

    "Bedroom": 22,

    "Kitchen": 20,

    "Bathroom": 23

}

smart\_home = SmartHome(rooms)

smart\_home.sense\_temperature()

smart\_home.control\_heater()

smart\_home.update\_temperature("Living Room", 23)

smart\_home.control\_heater()

Lab task : 04

Task 1: Code for LUHN Algorithm

def luhn\_algorithm(card\_number):

    card\_number = [int(digit) for digit in str(card\_number)]

    checksum = 0

    card\_number.reverse()

    for i in range(len(card\_number)):

        if i % 2 == 1:

            card\_number[i] \*= 2

            if card\_number[i] > 9:

                card\_number[i] -= 9

        checksum += card\_number[i]

    return checksum % 10 == 0

card\_number = "4532015112830366"

if luhn\_algorithm(card\_number):

    print("The card number is valid.")

else:

    print("The card number is invalid.")

Task 2: Remove Punctuations from UserInput String (without using remove function)

import string

def remove\_punctuation(user\_input):

    punctuations = string.punctuation

    result = ""

    for char in user\_input:

        if char not in punctuations:

            result += char

    return result

user\_input = "Hello, World! How's it going?"

cleaned\_input = remove\_punctuation(user\_input)

print(cleaned\_input)

Task 3: Sort text (word) in Alphabetical Order (without using sort function)

def alphabetical\_sort(text):

    words = text.split()

    for i in range(len(words) - 1):

        for j in range(len(words) - i - 1):

            if words[j] > words[j + 1]:

                words[j], words[j + 1] = words[j + 1], words[j]

    return " ".join(words)

user\_input = "apple banana mango cherry orange"

sorted\_text = alphabetical\_sort(user\_input)

print(sorted\_text)

lab no 5

Task 1 DFS with node and stack

class Node:

    def \_\_init\_\_(self, value):

        self.value = value

        self.adjacent = []

    def add\_adjacent(self, node):

        self.adjacent.append(node)

def dfs\_with\_stack(start\_node):

    stack = [start\_node]

    visited = set()

    while stack:

        node = stack.pop()

        if node not in visited:

            print(node.value)

            visited.add(node)

            for adjacent in node.adjacent:

                if adjacent not in visited:

                    stack.append(adjacent)

node\_a = Node('a')

node\_b = Node('b')

node\_c = Node('c')

node\_d = Node('d')

node\_e = Node('f')

node\_a.add\_adjacent(node\_b)

node\_a.add\_adjacent(node\_c)

node\_b.add\_adjacent(node\_d)

node\_b.add\_adjacent(node\_e)

dfs\_with\_stack(node\_a)

**task 2**

class TreeNode:

    def \_\_init\_\_(self, value):

        self.value = value

        self.left = None

        self.right = None

def preorder\_traversal(node):

    if node:

        print(node.value, end=' ')

        preorder\_traversal(node.left)

        preorder\_traversal(node.right)

def inorder\_traversal(node):

    if node:

        inorder\_traversal(node.left)

        print(node.value, end=' ')

        inorder\_traversal(node.right)

def postorder\_traversal(node):

    if node:

        postorder\_traversal(node.left)

        postorder\_traversal(node.right)

        print(node.value, end=' ')

root = TreeNode('A')

root.left = TreeNode('B')

root.right = TreeNode('C')

root.left.left = TreeNode('D')

root.left.right = TreeNode('E')

print("Preorder Traversal:")

preorder\_traversal(root)

print("\nInorder Traversal:")

inorder\_traversal(root)

print("\nPostorder Traversal:")

postorder\_traversal(root)

lab 06

Task 1 BFS without Queue & without Node

def bfs(maze, start, goal):

    rows = len(maze)

    cols = len(maze[0])

    def bfs\_recursive(current, visited, path):

        x, y = current

        if current == goal:

            path.append(current)

            return True

        visited[x][y] = True

        path.append(current)

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

        for direction in directions:

            new\_x = x + direction[0]

            new\_y = y + direction[1]

            if 0 <= new\_x < rows and 0 <= new\_y < cols and not visited[new\_x][new\_y] and maze[new\_x][new\_y] == 0:

                if bfs\_recursive((new\_x, new\_y), visited, path):

                    return True

        path.pop()

        return False

    visited = [[False for \_ in range(cols)] for \_ in range(rows)]

    path = []

    if bfs\_recursive(start, visited, path):

        return path

    else:

        return None

maze = [

    [0, 1, 0, 0, 0],

    [0, 1, 0, 1, 0],

    [0, 1, 0, 1, 0],

    [0, 0, 0, 1, 0],

    [1, 1, 0, 0, 0]

]

start = (0, 0)

goal = (3, 4)

path = bfs(maze, start, goal)

print(path)

task 2

class Node:

    def \_\_init\_\_(self, position, parent=None):

        self.position = position

        self.parent = parent

def bfs(maze, start, goal):

    start\_node = Node(start)

    goal\_node = Node(goal)

    queue = [start\_node]

    visited = set()

    visited.add(start\_node.position)

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

    while queue:

        current\_node = queue.pop(0)

        if current\_node.position == goal\_node.position:

            path = []

            while current\_node is not None:

                path.append(current\_node.position)

                current\_node = current\_node.parent

            return path[::-1]

        x, y = current\_node.position

        for direction in directions:

            new\_x = x + direction[0]

            new\_y = y + direction[1]

            new\_position = (new\_x, new\_y)

            if 0 <= new\_x < len(maze) and 0 <= new\_y < len(maze[0]) and maze[new\_x][new\_y] == 0:

                if new\_position not in visited:

                    visited.add(new\_position)

                    new\_node = Node(new\_position, current\_node)

                    queue.append(new\_node)

    return None

maze = [

    [0, 1, 0, 0, 0],

    [0, 1, 0, 1, 0],

    [0, 1, 0, 1, 0],

    [0, 0, 0, 1, 0],

    [1, 1, 0, 0, 0]

]

start = (0, 0)

goal = (3, 4)

path = bfs(maze, start, goal)

print(path)

lab no 07

Name Aoun Muhammad

Roll no 045

Lab 7

class Node:

    def \_\_init\_\_(self, parent=None, position=None):

        self.parent = parent

        self.position = position

        self.g = 0

        self.h = 0

        self.f = 0

def astar(maze, start, end):

    start\_node = Node(None, start)

    start\_node.g = start\_node.h = start\_node.f = 0

    end\_node = Node(None, end)

    end\_node.g = end\_node.h = end\_node.f = 0

    open\_list = []

    closed\_list = []

    open\_list.append(start\_node)

    while open\_list:

        current\_node = min(open\_list, key=lambda node: node.f)

        open\_list.remove(current\_node)

        closed\_list.append(current\_node)

        if current\_node.position == end\_node.position:

            path = []

            while current\_node is not None:

                path.append(current\_node.position)

                current\_node = current\_node.parent

            return path[::-1]

        children = []

        for new\_position in [(0, -1), (0, 1), (-1, 0), (1, 0)]:

            node\_position = (current\_node.position[0] + new\_position[0], current\_node.position[1] + new\_position[1])

            if node\_position[0] < 0 or node\_position[0] >= len(maze) or node\_position[1] < 0 or node\_position[1] >= len(maze[0]):

                continue

            if maze[node\_position[0]][node\_position[1]] != 0:

                continue

            new\_node = Node(current\_node, node\_position)

            children.append(new\_node)

        for child in children:

            if child in closed\_list:

                continue

            child.g = current\_node.g + 1

            child.h = ((child.position[0] - end\_node.position[0]) \*\* 2) + ((child.position[1] - end\_node.position[1]) \*\* 2)

            child.f = child.g + child.h

            if any(open\_node for open\_node in open\_list if child.position == open\_node.position and child.g > open\_node.g):

                continue

            open\_list.append(child)

maze = [

    [0, 1, 0, 0, 0, 0],

    [0, 1, 0, 1, 1, 0],

    [0, 0, 0, 1, 0, 0],

    [0, 1, 0, 0, 0, 0],

    [0, 1, 0, 1, 1, 0],

    [0, 0, 0, 0, 0, 0],

]

start = (0, 0)

end = (5, 5)

path = astar(maze, start, end)

print(path)