AI TASK 1

def fizz\_buzz():

for i in range(1, 101):

if i % 3 == 0 and i % 5 == 0:

print("Fizz Buzz")

elif i % 3 == 0:

print("Fizz")

elif i % 5 == 0:

print("Buzz")

else:

print(i)

fizz\_buzz()

# project 2

def movie\_budget\_analysis():

movies = [

("Horror Excellence", 20000000),

("Laugh Riot", 9000000),

("Fantasy Saga", 4500000),

("Tearjerker Drama", 379000000),

("Comic Gold", 365000000),

("Fictional Masterpiece", 356000000),

("Magical Voyage", 200000000)

]

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

for \_ in range(num\_movies\_to\_add):

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

budget = int(input("Enter the movie budget: "))

movies.append((name, budget))

total\_budget = sum(movie[1] for movie in movies)

average\_budget = total\_budget / len(movies)

print(f"\nThe average budget is: ${average\_budget:,.2f}")

high\_budget\_movies = [(name, budget) for name, budget in movies if budget > average\_budget]

print("\nMovies with budgets higher than the average:")

for name, budget in high\_budget\_movies:

difference = budget - average\_budget

print(f"- {name}: ${budget:,.2f} (exceeds by ${difference:,.2f})")

print(f"\nNumber of movies with budgets higher than the average: {len(high\_budget\_movies)}")

movie\_budget\_analysis()

**Explanation of the Code:**

1. **FizzBuzz Function:**
   * The fizz\_buzz() function prints numbers from **1 to 100**.
   * If a number is **divisible by both 3 and 5**, it prints **"Fizz Buzz"**.
   * If a number is **only divisible by 3**, it prints **"Fizz"**.
   * If a number is **only divisible by 5**, it prints **"Buzz"**.
   * Otherwise, it prints the number itself.
2. **Movie Budget Analysis Function:**
   * A list of movies with their budgets is defined.
   * The user is asked how many new movies they want to add, and their details are collected.
   * The **total budget** and **average budget** of all movies are calculated.
   * The program identifies movies with **budgets higher than the average** and displays them.
   * It also prints how much each of these movies exceeds the average budget.
   * Finally, it shows the **total count** of movies that have a budget above average.

AI TASK 2

class ModelBasedReflexAgent:

    def \_\_init\_\_(self, threshold=22):

        self.threshold = threshold

        self.previous\_action = None

    def perceive\_environment(self, current\_temperature):

        return current\_temperature

    def decide\_action(self, current\_temperature):

        if current\_temperature < self.threshold:

            if self.previous\_action != "Turn Heater On":

                self.previous\_action = "Turn Heater On"

                return "Turn Heater On"

        elif current\_temperature > self.threshold:

            if self.previous\_action != "Turn Heater Off":

                self.previous\_action = "Turn Heater Off"

                return "Turn Heater Off"

        return "Do Nothing"

    def act(self, current\_temperature):

        action = self.decide\_action(current\_temperature)

        print(f"Current Temperature: {current\_temperature}°C -> Action: {action}")

        return action

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

    agent = ModelBasedReflexAgent(threshold=22)

    temperatures = [20, 21, 22, 23, 24, 22, 21, 20, 19, 23, 24, 22]

    for temp in temperatures:

        agent.act(temp)

**Explanation of the Code:**

1. **Class Definition (ModelBasedReflexAgent)**
   * This is a model-based reflex agent that controls a heater based on the room temperature.
   * It has a **threshold temperature** (default is **22°C**).
   * It keeps track of the **previous action** to avoid unnecessary actions.
2. **Methods of the Class:**
   * **perceive\_environment(current\_temperature)** → Takes the current temperature as input and returns it.
   * **decide\_action(current\_temperature)** →
     + If the temperature is **below** the threshold and the heater is not already on, it **turns the heater on**.
     + If the temperature is **above** the threshold and the heater is not already off, it **turns the heater off**.
     + If the temperature is exactly at the threshold or no change is needed, it **does nothing**.
   * **act(current\_temperature)** → Calls decide\_action(), prints the action, and returns it.
3. **Main Execution (if \_\_name\_\_ == "\_\_main\_\_")**
   * An instance of ModelBasedReflexAgent is created with a **threshold of 22°C**.
   * A **list of temperature readings** is given.
   * The agent processes each temperature and decides whether to **turn the heater on, off, or do nothing**.

**For example:**

Current Temperature: 20°C -> Action: Turn Heater On

Current Temperature: 21°C -> Action: Do Nothing

Current Temperature: 22°C -> Action: Do Nothing

Current Temperature: 23°C -> Action: Turn Heater Off

Current Temperature: 24°C -> Action: Do Nothing

Current Temperature: 22°C -> Action: Do Nothing

Current Temperature: 21°C -> Action: Turn Heater On

Current Temperature: 20°C -> Action: Do Nothing

Current Temperature: 19°C -> Action: Do Nothing

Current Temperature: 23°C -> Action: Turn Heater Off

Current Temperature: 24°C -> Action: Do Nothing

Current Temperature: 22°C -> Action: Do Nothing

This ensures the heater is only toggled **when necessary**, preventing unnecessary actions.

AI TASK 3

import random

def get\_word():

    words = ["python", "zainu", "dynamic", "programming", "challenge", "faiqadon"]

    return random.choice(words).upper()

def print\_hangman(tries):

    stages = [

        """

           --------

           |      |

           |      O

           |     \|/

           |      |

           |     / \\

          ---

        """,

        """

           --------

           |      |

           |      O

           |     \|/

           |      |

           |     /

          ---

        """,

        """

           --------

           |      |

           |      O

           |     \|/

           |      |

           |

          ---

        """,

        """

           --------

           |      |

           |      O

           |     \|

           |      |

           |

          ---

        """,

        """

           --------

           |      |

           |      O

           |      |

           |      |

           |

          ---

        """,

        """

           --------

           |      |

           |      O

           |

           |

           |

          ---

        """,

        """

           --------

           |      |

           |

           |

           |

           |

          ---

        """

    ]

    print(stages[tries])

def hangman():

    word = get\_word()

    word\_completion = ["\_"] \* len(word)

    guessed = set()

    tries = 6

    print("Let's play Hangman!")

    print\_hangman(tries)

    print(" ".join(word\_completion))

    while tries > 0 and "\_" in word\_completion:

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

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

            print("Invalid input. Please enter a single letter.")

            continue

        if guess in guessed:

            print("You already guessed that letter.")

            continue

        guessed.add(guess)

        if guess in word:

            for i in range(len(word)):

                if word[i] == guess:

                    word\_completion[i] = guess

        else:

            tries -= 1

            print("Incorrect guess!")

        print\_hangman(tries)

        print(" ".join(word\_completion))

    if "\_" not in word\_completion:

        print("Congratulations! You guessed the word.")

    else:

        print(f"Game over! The word was {word}.")

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

    hangman()

**Explanation of the Hangman Game Code:**

1. **Function get\_word()**
   * Selects a random word from a list (["python", "zainu", "dynamic", "programming", "challenge", "faiqadon"]).
   * Converts the selected word to **uppercase** and returns it.
2. **Function print\_hangman(tries)**
   * Displays the **current hangman stage** based on remaining tries (6 tries at the start).
   * The stages visually represent the **progressive drawing** of the hangman as wrong guesses increase.
3. **Function hangman()**
   * Picks a random word using get\_word().
   * Creates a list (word\_completion) with **underscores (\_)** to represent the hidden word.
   * Initializes:
     + guessed: A set to keep track of **already guessed letters**.
     + tries = 6: Player has **6 chances** before losing.
   * Displays the **initial hangman drawing** and the hidden word format.
4. **Game Loop (while tries > 0 and "\_" in word\_completion)**
   * **Takes input from the user** (guess):
     + Ensures it's a **single letter** (no numbers or special characters).
     + Checks if the letter was **already guessed**.
   * **If the letter is in the word:**
     + Updates word\_completion to reveal the correctly guessed letter(s).
   * **If the letter is not in the word:**
     + Reduces tries by 1.
     + Prints "Incorrect guess!".
   * **Displays updated hangman drawing and word progress.**
5. **End of Game:**
   * If the player **guesses the word completely**, they win.
   * If tries reach **0**, they lose, and the correct word is revealed.

**Example Gameplay Output:**

Let's play Hangman!

(Initial Hangman Drawing)

\_ \_ \_ \_ \_ \_ \_

Guess a letter: P

Incorrect guess!

(Updated Hangman Drawing)

\_ \_ \_ \_ \_ \_ \_

Guess a letter: A

(A appears in the word)

(Updated Word Progress)

\_ A \_ \_ \_ A \_

...

Congratulations! You guessed the word!

OR

Game over! The word was DYNAMIC.

This is a **simple and fun Hangman game** where the player guesses a hidden word before running out of tries!

AI task 4

import string

def luhn\_algorithm(card\_number):

    digits = [int(d) for d in str(card\_number)][::-1]

    for i in range(1, len(digits), 2):

        digits[i] \*= 2

        if digits[i] > 9:

            digits[i] -= 9

    return sum(digits) % 10 == 0

def remove\_punctuation(text):

    return text.translate(str.maketrans('', '', string.punctuation))

def sort\_sentence(sentence):

    words = sentence.split()

    words.sort(key=str.lower)

    return ' '.join(words)

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

    card\_number = "45487525712830366"

    print(f"Is the card number valid? {luhn\_algorithm(card\_number)}")

    sample\_text = "Hello, Earth! How are you?"

    print(f"Text without punctuation: {remove\_punctuation(sample\_text)}")

    sample\_sentence = "This is a test sentence for sorting."

    print(f"Sorted sentence: {sort\_sentence(sample\_sentence)}")

**Explanation of the Code:**

This script contains three functions:

1. **Luhn Algorithm (luhn\_algorithm)** - Validates a credit card number.
2. **Remove Punctuation (remove\_punctuation)** - Removes punctuation from a given text.
3. **Sort Sentence (sort\_sentence)** - Sorts words in a sentence alphabetically.

**1. Luhn Algorithm (Credit Card Validation)**

**Function:** luhn\_algorithm(card\_number)

* Converts the card number into a list of digits in **reverse order**.
* **Doubles every second digit** from the right:
  + If the result is **greater than 9**, subtract 9.
* **Sums all digits** and checks if divisible by **10**.
* Returns True if **valid**, otherwise False.

**For Example:**

luhn\_algorithm("79927398713") # True (Valid)

luhn\_algorithm("1234567812345678") # False (Invalid)

**2. Remove Punctuation from Text**

**Function:** remove\_punctuation(text)

* Uses str.translate() with string.punctuation to remove all punctuation.

**For Example:**

remove\_punctuation("Hello, Earth! How are you?")

# Output: "Hello Earth How are you"

**3. Sort Words in a Sentence Alphabetically**

**Function:** sort\_sentence(sentence)

* **Splits** the sentence into words.
* **Sorts** words alphabetically (case-insensitive).
* **Joins** them back into a sentence.

**For Example:**

sort\_sentence("This is a test sentence for sorting.")

# Output: "a for is sentence sorting. test This"

**Main Execution (if \_\_name\_\_ == "\_\_main\_\_")**

* **Checks if a card number is valid** using luhn\_algorithm().
* **Removes punctuation** from a sample text.
* **Sorts words** in a sample sentence.

**For Example:**

Is the card number valid? False

Text without punctuation: Hello Earth How are you

Sorted sentence: a for is sentence sorting. test This

This script demonstrates **credit card validation, text cleaning, and sorting** efficiently!

AI TASK 5

class Node:

    def \_\_init\_\_(self, value):

        self.value = value

        self.left = None

        self.right = None

def dfs\_with\_stack(root):

    if not root:

        return

    stack = [root]

    while stack:

        node = stack.pop()

        print(node.value, end=' ')

        if node.right:

            stack.append(node.right)

        if node.left:

            stack.append(node.left)

def inorder\_traversal(root):

    if root:

        inorder\_traversal(root.left)

        print(root.value, end=' ')

        inorder\_traversal(root.right)

def preorder\_traversal(root):

    if root:

        print(root.value, end=' ')

        preorder\_traversal(root.left)

        preorder\_traversal(root.right)

def postorder\_traversal(root):

    if root:

        postorder\_traversal(root.left)

        postorder\_traversal(root.right)

        print(root.value, end=' ')

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

    root = Node(7)

    root.left = Node(8)

    root.right = Node(1)

    root.left.left = Node(0)

    root.left.right = Node(3)

    root.right.left = Node(9)

    root.right.right = Node(6)

    print("DFS with Stack:")

    dfs\_with\_stack(root)

    print("\nInorder Traversal:")

    inorder\_traversal(root)

    print("\nPreorder Traversal:")

    preorder\_traversal(root)

    print("\nPostorder Traversal:")

    postorder\_traversal(root)

**Explanation of the Code (Binary Tree Traversals)**

This script defines a **binary tree** and implements different **tree traversal methods**, including **Depth-First Search (DFS)**.

**1. Node Class (Node)**

* Represents a **binary tree node**.
* Each node stores a **value** and has two **child pointers**: left and right.

class Node:

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

self.value = value

self.left = None

self.right = None

**2. DFS with Stack (dfs\_with\_stack)**

* Uses an **explicit stack** (iterative DFS) instead of recursion.
* **Process:**
  + Start with the root node in the stack.
  + Pop a node, print its value.
  + **Push right child first**, then left (so left is processed first when popped).

**Output Example:**

DFS with Stack:

7 8 0 3 1 9 6

*(Order: Root → Left → Right, similar to Preorder Traversal)*

**3. Inorder Traversal (inorder\_traversal)**

* **Left → Root → Right** (recursive method).

**Output Example:**

Inorder Traversal:

0 8 3 7 9 1 6

*(Sorted order for Binary Search Trees)*

**4. Preorder Traversal (preorder\_traversal)**

* **Root → Left → Right** (recursive method).

**Output Example:**

Preorder Traversal:

7 8 0 3 1 9 6

*(Same order as DFS with stack)*

**5. Postorder Traversal (postorder\_traversal)**

* **Left → Right → Root** (recursive method).

**Output Example:**

Postorder Traversal:

0 3 8 9 6 1 7

*(Used in deleting trees and evaluating expressions)*

**6. Tree Structure Used**

The tree in the example is:

7

/ \

8 1

/ \ / \

0 3 9 6

**7. Execution (if \_\_name\_\_ == "\_\_main\_\_")**

* Creates the **binary tree**.
* Calls all **traversal functions** and prints results.

**Final Output**

DFS with Stack:

7 8 0 3 1 9 6

Inorder Traversal:

0 8 3 7 9 1 6

Preorder Traversal:

7 8 0 3 1 9 6

Postorder Traversal:

0 3 8 9 6 1 7

This script efficiently demonstrates **tree traversals** using **both recursion and iteration (DFS with stack).**

AI TASK 6

def bfs\_without\_queue(graph, start):

    visited = set()

    def visit(node):

        if node not in visited:

            print(node, end=' ')

            visited.add(node)

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

                visit(neighbor)

    visit(start)

from collections import deque

class Node:

    def \_\_init\_\_(self, value):

        self.value = value

        self.children = []

def bfs\_with\_queue(root):

    if not root:

        return

    queue = deque([root])

    while queue:

        node = queue.popleft()

        print(node.value, end=' ')

        queue.extend(node.children)

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

    graph = {

        'A': ['B', 'C'],

        'B': ['D', 'E'],

        'C': ['F'],

        'D': [],

        'E': ['G'],

        'F': [],

        'G': []

    }

    print("BFS Without Queue:")

    bfs\_without\_queue(graph, 'A')

    print("\nBFS With Queue:")

    root = Node('A')

    root.children.append(Node('B'))

    root.children.append(Node('C'))

    root.children[0].children.append(Node('D'))

    root.children[0].children.append(Node('E'))

    root.children[1].children.append(Node('F'))

    root.children[0].children[1].children.append(Node('G'))

    bfs\_with\_queue(root)

**Explanation of the Code (Breadth-First Search - BFS)**

This script implements **Breadth-First Search (BFS)** in two ways:

1. **Without using a queue (recursive approach)**
2. **Using a queue (iterative approach)**

**1. BFS Without Queue (bfs\_without\_queue)**

* **Uses recursion** instead of a queue.
* **How it works:**
  + A **set** (visited) stores visited nodes.
  + A recursive visit() function is used to explore all neighbors.
  + Nodes are visited **before** their neighbors (preorder DFS-like traversal).

Example Graph Used:

A

/ \

B C

/ \ \

D E F

/

G

**Output Example:**

BFS Without Queue:

A B D E G C F

**Note:** This method does not guarantee correct BFS order. Instead, it behaves like **DFS**.

**2. BFS With Queue (bfs\_with\_queue)**

* **Uses an explicit queue** (implemented with collections.deque).
* **How it works:**
  + Start with the **root node** in the queue.
  + **Process nodes level by level** (FIFO order).
  + Add children to the queue and continue.

**Output Example:**

BFS With Queue:

A B C D E F G

This correctly follows BFS traversal order.

**3. Execution (if \_\_name\_\_ == "\_\_main\_\_")**

* **Graph-based BFS** (Recursive DFS-like traversal).
* **Tree-based BFS** (Correct BFS using a queue).
* Prints BFS results for both implementations.

**Final Output**

BFS Without Queue:

A B D E G C F

BFS With Queue:

A B C D E F G

This script shows both **incorrect (DFS-like)** and **correct (queue-based BFS)** implementations for **graph/tree traversal.**

AI TASK 7

from queue import PriorityQueue

def a\_star(graph, start, goal, h):

    open\_set = PriorityQueue()

    open\_set.put((0, start))

    came\_from = {}

    g\_score = {node: float('inf') for node in graph}

    g\_score[start] = 0

    f\_score = {node: float('inf') for node in graph}

    f\_score[start] = h[start]

    while not open\_set.empty():

        \_, current = open\_set.get()

        if current == goal:

            path = []

            while current in came\_from:

                path.append(current)

                current = came\_from[current]

            path.append(start)

            return path[::-1]

        for neighbor, cost in graph[current]:

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

            if tentative\_g\_score < g\_score[neighbor]:

                came\_from[neighbor] = current

                g\_score[neighbor] = tentative\_g\_score

                f\_score[neighbor] = g\_score[neighbor] + h[neighbor]

                open\_set.put((f\_score[neighbor], neighbor))

    return None

graph = {

    'A': [('B', 1), ('C', 4)],

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

    'C': [('A', 4), ('F', 3)],

    'D': [('B', 2), ('E', 1)],

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

    'F': [('C', 3), ('E', 2)]

}

h = {'A': 6, 'B': 4, 'C': 4, 'D': 2, 'E': 2, 'F': 0}

start, goal = 'A', 'F'

path = a\_star(graph, start, goal, h)

print(f"Shortest path from {start} to {goal}: {path}")

***Explanation of A Algorithm Code*\***

This Python script implements the A (A-star) Search Algorithm\* for **finding the shortest path** in a weighted graph.

***How A Algorithm Works*\***

1. **Initialization**:
   * Uses a **priority queue** (open\_set) to store nodes with the lowest cost first.
   * Initializes **g\_score** (actual cost) and **f\_score** (estimated cost).
   * **g\_score** is set to 0 for the start node and ∞ for others.
   * **f\_score** (total cost estimate) is set to **heuristic (h) value** for the start node.
2. **Processing Nodes in Order of Cost**:
   * Picks the node with the lowest **f\_score** (from open\_set).
   * If it’s the goal, reconstructs the path.
   * Otherwise, updates neighbor nodes:
     + **g\_score** (actual cost from start).
     + **f\_score** (g\_score + heuristic).
     + Adds the neighbor to the priority queue.
3. **Loop Continues Until Goal is Reached**
   * If no path exists, returns None.

**Graph Used**

This is an **undirected weighted graph** where:

* Nodes represent locations.
* Edge weights represent travel costs.
* Heuristic (h) estimates the distance to goal F.

A --1-- B --2-- D

| | |

4 5 1

| | |

C --3-- F --2-- E

**Heuristic values (h) represent estimated distances to 'F':**

h = {'A': 6, 'B': 4, 'C': 4, 'D': 2, 'E': 2, 'F': 0}

**Example Execution**

Shortest path from A to F: ['A', 'B', 'D', 'E', 'F']

**Key Takeaways**

* *A is efficient*\* because it uses both:
  + **g(n): Actual cost from start**
  + **h(n): Estimated cost to goal**
* **Priority queue ensures optimal path** is explored first.
* **Heuristic function (h) should be well-designed** for best performance.

This implementation provides an **optimal and efficient** way to find paths in weighted graphs.

AI TASK 8

import math

def minmax(depth, node\_index, is\_maximizing, values, alpha, beta):

    if depth == 0 or node\_index >= len(values):

        return values[node\_index]

    if is\_maximizing:

        best = -math.inf

        for i in range(2):

            val = minmax(depth - 1, node\_index \* 2 + i, False, values, alpha, beta)

            best = max(best, val)

            alpha = max(alpha, best)

            if beta <= alpha:

                break

        return best

    else:

        best = math.inf

        for i in range(2):

            val = minmax(depth - 1, node\_index \* 2 + i, True, values, alpha, beta)

            best = min(best, val)

            beta = min(beta, best)

            if beta <= alpha:

                break

        return best

values = [3, 5, 6, 9, 1, 2, 0, -1]

depth = math.log2(len(values))

optimal\_value = minmax(int(depth), 0, True, values, -math.inf, math.inf)

print(f"Optimal value using Min-Max Algorithm: {optimal\_value}")

**Explanation of Minimax Algorithm with Alpha-Beta Pruning**

This Python script **implements the Minimax algorithm** with **Alpha-Beta Pruning**, which is commonly used in game theory for decision-making in **two-player games**.

**How Minimax with Alpha-Beta Pruning Works**

1. **Tree Representation**:
   * The tree consists of **maximizing** and **minimizing** nodes.
   * Each level alternates between a **Maximizer** and a **Minimizer**.
   * **Leaf nodes** represent the values (outcomes).
2. **Recursive Evaluation**:
   * The algorithm **recursively evaluates** the game tree from the leaves up.
   * It returns the best possible move for the **Maximizer** assuming the **Minimizer** plays optimally.
3. **Alpha-Beta Pruning**:
   * **Alpha (α):** Best value found for the maximizer.
   * **Beta (β):** Best value found for the minimizer.
   * **Pruning occurs** when beta ≤ alpha, skipping unnecessary branches to improve efficiency.

**Example Execution**

**Tree Representation**

(Max)

/ \

(Min) (Min)

/ \ / \

(3) (5) (6) (9)

/ \ / \ / \ / \

3 5 6 9 1 2 0 -1

**Given Values**

values = [3, 5, 6, 9, 1, 2, 0, -1]

* These are the **leaf node values**.

**Minimax Calculation**

* The function **recursively determines** the **optimal move**.
* Alpha-Beta pruning **skips unnecessary evaluations**, making it **faster**.

**Final Output**

Optimal value using Min-Max Algorithm: 5