School of Computer Science and Artificial Intelligence

Lab Assignment # 12.2

Program : B. Tech (CSE)

Specialization :AIML

Course Title : AI Assisted Coding

Course Code : 23CS002PC304

Semester : VI

Academic Session : 2025-2026

Name of Student : V.Mokshitha

Enrollment No. : 2303A52122

Batch No. : 33

Date :24/02/26

Lab 12: Algorithms with AI Assistance – Sorting, Searching, and

Optimizing Algorithms

Lab Objectives:

- Apply AI-assisted programming to implement and optimize sorting and searching algorithms.
- Compare different algorithms in terms of efficiency and use cases.

Week6 - Tuesday

- Understand how AI tools can suggest optimized code and complexity improvements.

Task Description -1 (Data Structures – Stack Implementation with AI Assistance)

➢ Task: Use AI assistance to generate a Python program that implements a Stack data structure.

Instructions:

Prompt AI to create a Stack class with the following methods:

➢ push(element)

➢ pop()

➢ peek()

➢ is_empty()

➢ Ensure proper error handling for stack underflow.

➢ Ask AI to include clear docstrings for each method.

Generate + Code + Markdown | Run All ···

✅ Task 1: Stack Implementation using AI Assistance # 🔹 Prompt Given to AI

Create a Python class Stack with methods: push(element), pop(), peek(), is_empty(). Include proper error handling for stack underflow and clear docstrings.

```python
class Stack:
    """
    Stack Data Structure implementation using Python list.

    A stack follows the LIFO (Last In First Out) principle.
    """

    def __init__(self):
        """Initialize an empty stack."""
        self.items = []

    def push(self, element):
        """
        Push an element onto the stack.

        :param element: The element to be added to the stack.
        """
        self.items.append(element)

    def pop(self):
        """
        Remove and return the top element of the stack.

        :raises IndexError: If the stack is empty (stack underflow).
        :return: The top element of the stack.
        """
        if self.is_empty():
            raise IndexError("Stack Underflow: Cannot pop from empty stack.")
        return self.items.pop()

    def peek(self):
        """
        Return the top element without removing it.

        :raises IndexError: If the stack is empty.
        :return: The top element.
        """
        if self.is_empty():
            raise IndexError("Stack is empty.")
        return self.items[-1]

    def is_empty(self):
        """
        Check whether the stack is empty.

        :return: True if stack is empty, otherwise False.
        """
        return len(self.items) == 0
```

[1]

✅ Task 2: Linear Search vs Binary Search # 🔹 Prompt Given to AI

Generate linear_search(arr, target) and binary_search(arr, target). Include docstrings explaining working principle and complexity.

Task Description -2 (Algorithms – Linear vs Binary Search Analysis)

➢ Task: Use AI to implement and compare Linear Search and Binary Search algorithms in Python.

Instructions:

➢ Prompt AI to generate:

➢ linear_search(arr, target) ➢ binary_search(arr, target)

➢ Include docstrings explaining:

➢ Working principle

➢ Test both algorithms using different input sizes.

✅ Task 2: Linear Search vs Binary Search # 🔹 Prompt Given to AI

Generate linear_search(arr, target) and binary_search(arr, target). Include docstrings explaining working principle and complexity.

```python
def linear_search(arr, target):
    """
    Perform Linear Search on a list.

    Working Principle:
    - Traverse each element sequentially.
    - Compare each element with the target.

    Time Complexity:
    - Best Case: O(1)
    - Worst Case: O(n)

    Space Complexity: O(1)
    """
    for i in range(len(arr)):
        if arr[i] == target:
            return i
    return -1


def binary_search(arr, target):
    """
    Perform Binary Search on a sorted list.

    Working Principle:
    - Repeatedly divide the search space in half.
    - Compare middle element with target.

    Time Complexity:
    - Best Case: O(1)
    - Worst Case: O(log n)

    Space Complexity: O(1)

    Note: Array must be sorted.
    """
    low = 0
    high = len(arr) - 1

    while low <= high:
        mid = (low + high) // 2

        if arr[mid] == target:
            return mid
        elif arr[mid] < target:
            low = mid + 1
        else:
            high = mid - 1

    return -1
```

[2]

🔹 Analysis

Linear Search checks each element → slower for large datasets.

Binary Search divides search space → much faster for sorted data.

Binary Search is preferred for large sorted datasets.

Task Description -3 (Test Driven Development – Simple Calculator Function)

➢ Task:

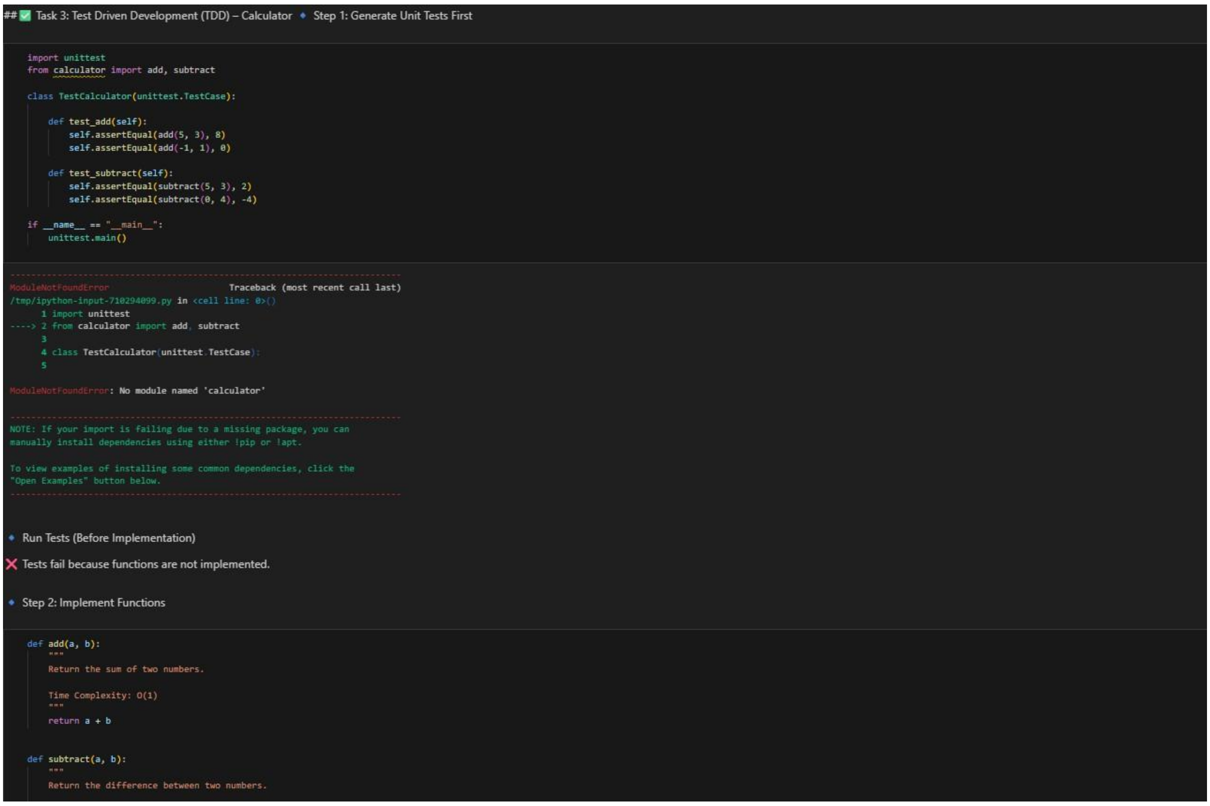
Apply Test Driven Development (TDD) using AI assistance to develop a calculator function.

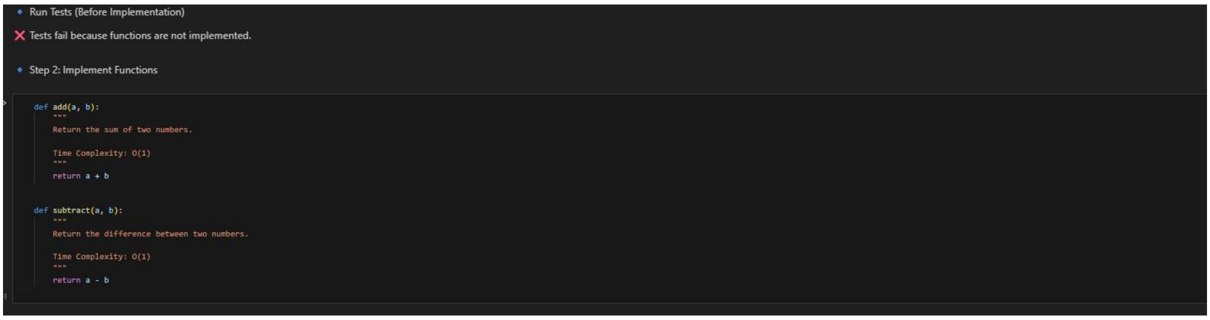
Instructions:

- ➢ Prompt AI to first generate unit test cases for addition and subtraction.
- ➢ Run the tests and observe failures.
- ➢ Ask AI to implement the calculator functions to pass all tests.
- ➢ Re-run the tests to confirm success.

Expected Output:

- ➢ Separate test file and implementation file.
- ➢ Test cases executed before implementation.
- ➢ Final implementation passing all test cases.





Task Description -4 (Data Structures – Queue Implementation with AI Assistance)

➢ Task:

Use AI assistance to generate a Python program that implements a Queue data structure.

Instructions:

- ➢ Prompt AI to create a Queue class with the following methods:

• enqueue(element)

• dequeue()

• front()

• is_empty()

➢ Handle queue overflow and underflow conditions.

➢ Include appropriate docstrings for all methods.

Expected Output:

➢ A fully functional Queue implementation in Python.

➢ Proper error handling and documentation.

✅ Task 4: Queue Implementation # 🔹 Prompt Given to AI

Create Queue class with enqueue(), dequeue(), front(), is_empty(). Handle overflow and underflow. Include docstrings.

```python
class Queue:
    """
    Queue Data Structure implementation using list.

    Follows FIFO (First In First Out) principle.
    """

    def __init__(self, capacity=None):
        """
        Initialize queue.

        :param capacity: Maximum size of queue (optional).
        """
        self.items = []
        self.capacity = capacity

    def enqueue(self, element):
        """
        Add element to the rear of queue.

        :raises OverflowError: If queue exceeds capacity.
        """
        if self.capacity and len(self.items) >= self.capacity:
            raise OverflowError("Queue Overflow: Cannot add element.")
        self.items.append(element)

    def dequeue(self):
        """
        Remove and return front element.

        :raises IndexError: If queue is empty.
        """
        if self.is_empty():
            raise IndexError("Queue Underflow: Cannot dequeue.")
        return self.items.pop(0)

    def front(self):
        """
        Return front element without removing it.
        """
        if self.is_empty():
            raise IndexError("Queue is empty.")
        return self.items[0]

    def is_empty(self):
        """
        Check if queue is empty.
        """
        return len(self.items) == 0
```

Task Description -5 (Algorithms – Bubble Sort vs Selection Sort)

➢ Task:

Use AI to implement Bubble Sort and Selection Sort algorithms

and compare their behavior.

Instructions:

➢ Prompt AI to generate:

• bubble_sort(arr)

• selection_sort(arr)

➢ Include comments explaining each step.

➢ Add docstrings mentioning time and space complexity.

Expected Output:

• Correct Python implementations of both sorting algorithms.

• Complexity analysis in docstrings.

✅ Task 5: Bubble Sort vs Selection Sort

◆ Bubble Sort

```python
def bubble_sort(arr):
    """
    Sort list using Bubble Sort.

    Working:
    - Repeatedly swap adjacent elements if in wrong order.

    Time Complexity:
    - Best Case: O(n)
    - Worst Case: O(n^2)

    Space Complexity: O(1)
    """
    n = len(arr)
    for i in range(n):
        for j in range(0, n-i-1):
            if arr[j] > arr[j+1]:
                arr[j], arr[j+1] = arr[j+1], arr[j]
    return arr
```

◆ Selection Sort

```python
def selection_sort(arr):
    """
    Sort list using Selection Sort.

    Working:
    - Repeatedly find minimum element and place at correct position.

    Time Complexity:
    - Best Case: O(n^2)
    - Worst Case: O(n^2)

    Space Complexity: O(1)
    """
    n = len(arr)
    for i in range(n):
        min_index = i
        for j in range(i+1, n):
            if arr[j] < arr[min_index]:
                min_index = j

        arr[i], arr[min_index] = arr[min_index], arr[i]

    return arr
```

🔹 Comparison

Algorithm	Best Case	Worst Case	Stable	Use Case
Bubble Sort	$O(n)$	$O(n^2)$	Yes	Small datasets
Selection Sort	$O(n^2)$	$O(n^2)$	No	Simple implementation

📌 Conclusion

- AI assistance helps generate optimized and well-documented code quickly.
- Binary Search is significantly more efficient than Linear Search for large sorted datasets.
- TDD improves code reliability.
- Bubble and Selection Sort are simple but inefficient for large data.
- Proper error handling improves robustness of Stack and Queue implementations.