ASSIGNMENT NO:12.2

Name : G.Dhanush Reddy

bt.no : 22

Ht.no : 2303A51619

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.

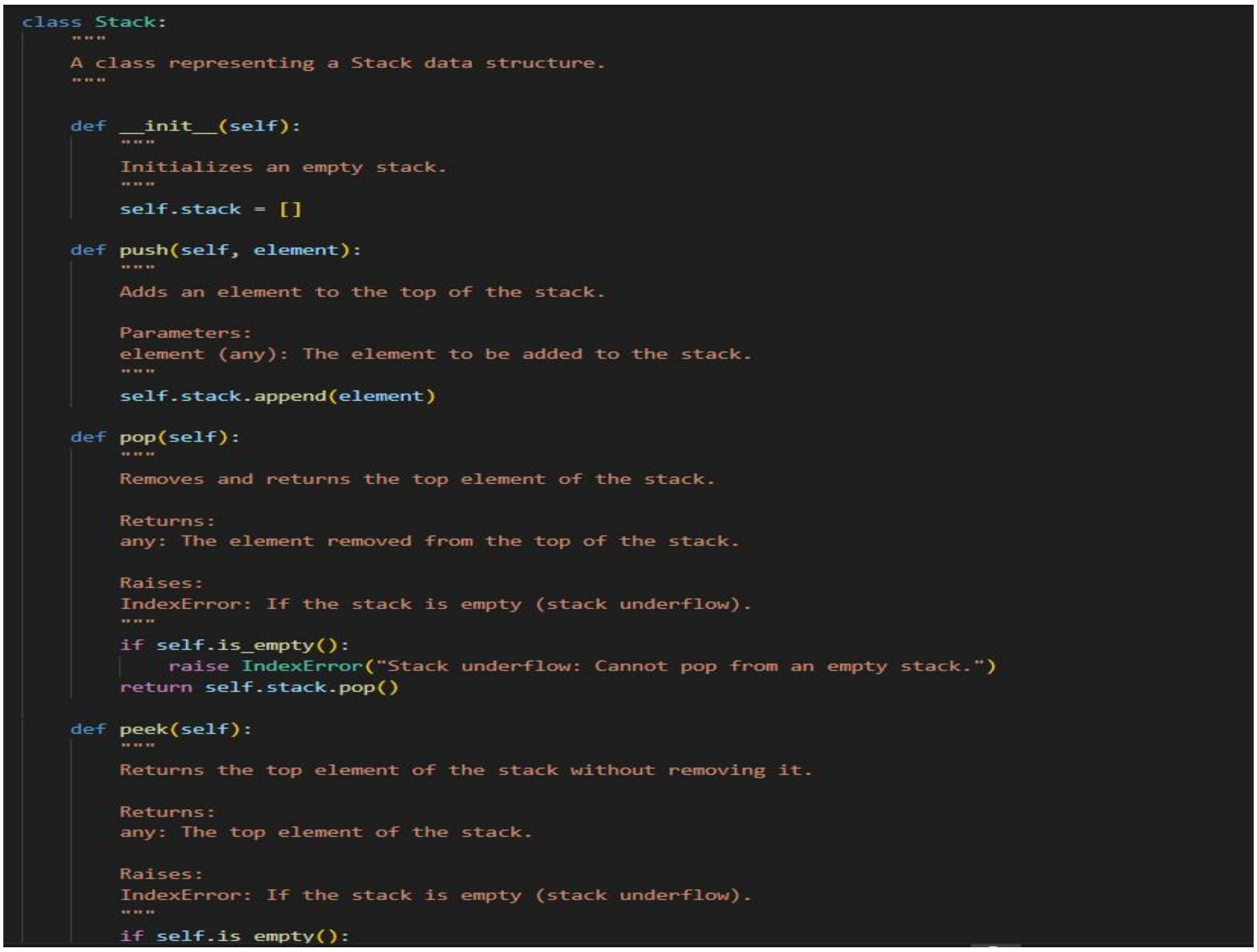
➢ Ask AI to include clear docstrings for each method.

Expected Output:

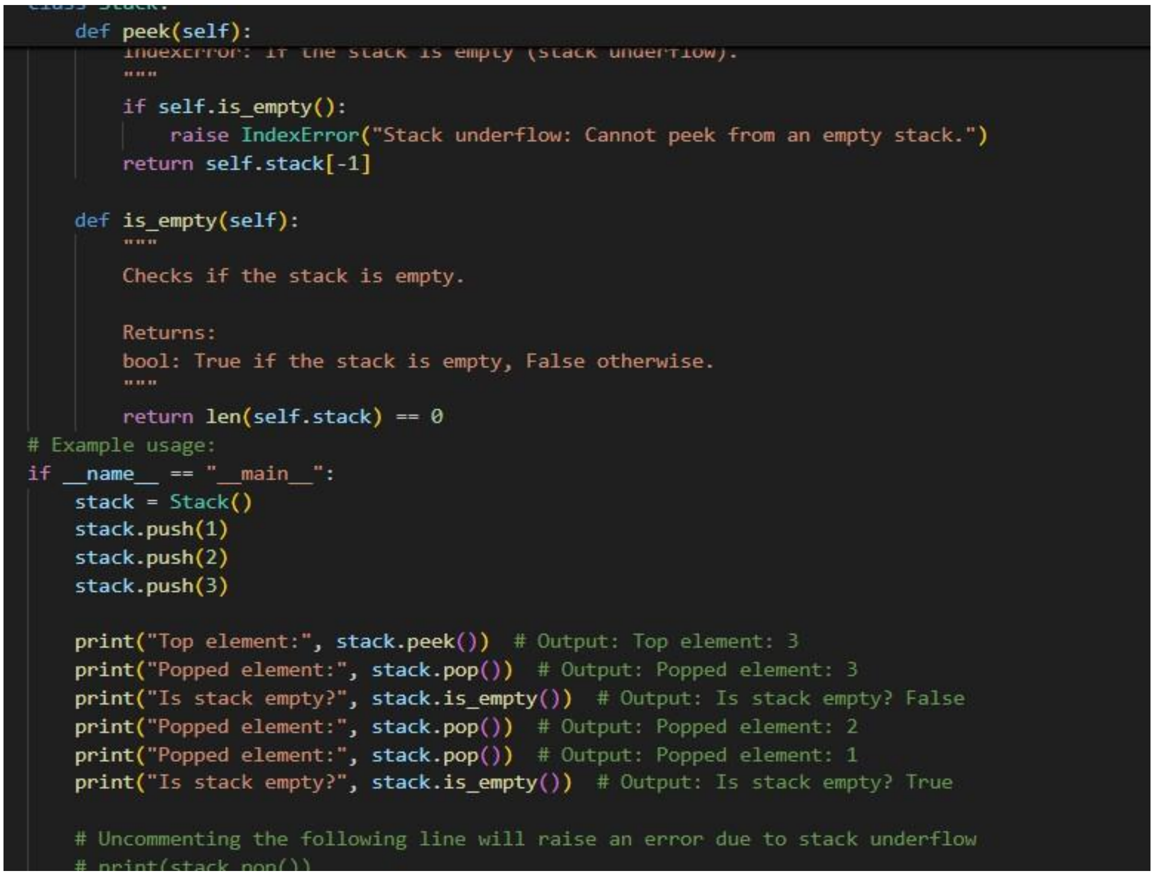
➢ A functional Python program implementing a Stack using a class.

➢ Properly documented methods with docstrings.

Output screenshot:

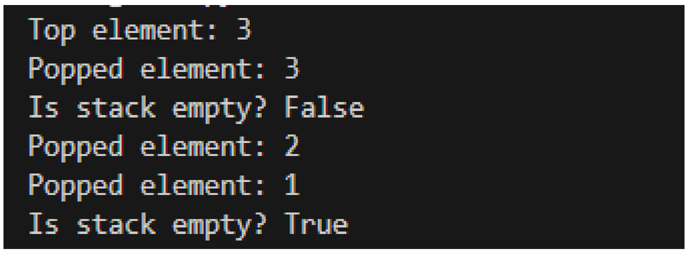


```python
class Stack:
    """
    A class representing a Stack data structure.
    """

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

    def push(self, element):
        """
        Adds an element to the top of the stack.

        Parameters:
        element (any): The element to be added to the stack.
        """
        self.stack.append(element)

    def pop(self):
        """
        Removes and returns the top element of the stack.

        Returns:
        any: The element removed from the top of the stack.

        Raises:
        IndexError: If the stack is empty (stack underflow).
        """
        if self.is_empty():
            raise IndexError("Stack underflow: Cannot pop from an empty stack.")
        return self.stack.pop()

    def peek(self):
        """
        Returns the top element of the stack without removing it.

        Returns:
        any: The top element of the stack.

        Raises:
        IndexError: If the stack is empty (stack underflow).
        """
        if self.is_empty():
```



```
    def peek(self):
        IndexError: If the stack is empty (stack underflow).
        """
        if self.is_empty():
            raise IndexError("Stack underflow: Cannot peek from an empty stack.")
        return self.stack[-1]

    def is_empty(self):
        """
        Checks if the stack is empty.

        Returns:
        bool: True if the stack is empty, False otherwise.
        """
        return len(self.stack) == 0
# Example usage:
if __name__ == "__main__":
    stack = Stack()
    stack.push(1)
    stack.push(2)
    stack.push(3)

    print("Top element:", stack.peek())  # Output: Top element: 3
    print("Popped element:", stack.pop())  # Output: Popped element: 3
    print("Is stack empty?", stack.is_empty())  # Output: Is stack empty? False
    print("Popped element:", stack.pop())  # Output: Popped element: 2
    print("Popped element:", stack.pop())  # Output: Popped element: 1
    print("Is stack empty?", stack.is_empty())  # Output: Is stack empty? True

    # Uncommenting the following line will raise an error due to stack underflow
    # print(stack.pop())
```

Output:



```
Top element: 3
Popped element: 3
Is stack empty? False
Popped element: 2
Popped element: 1
Is stack empty? True
```

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

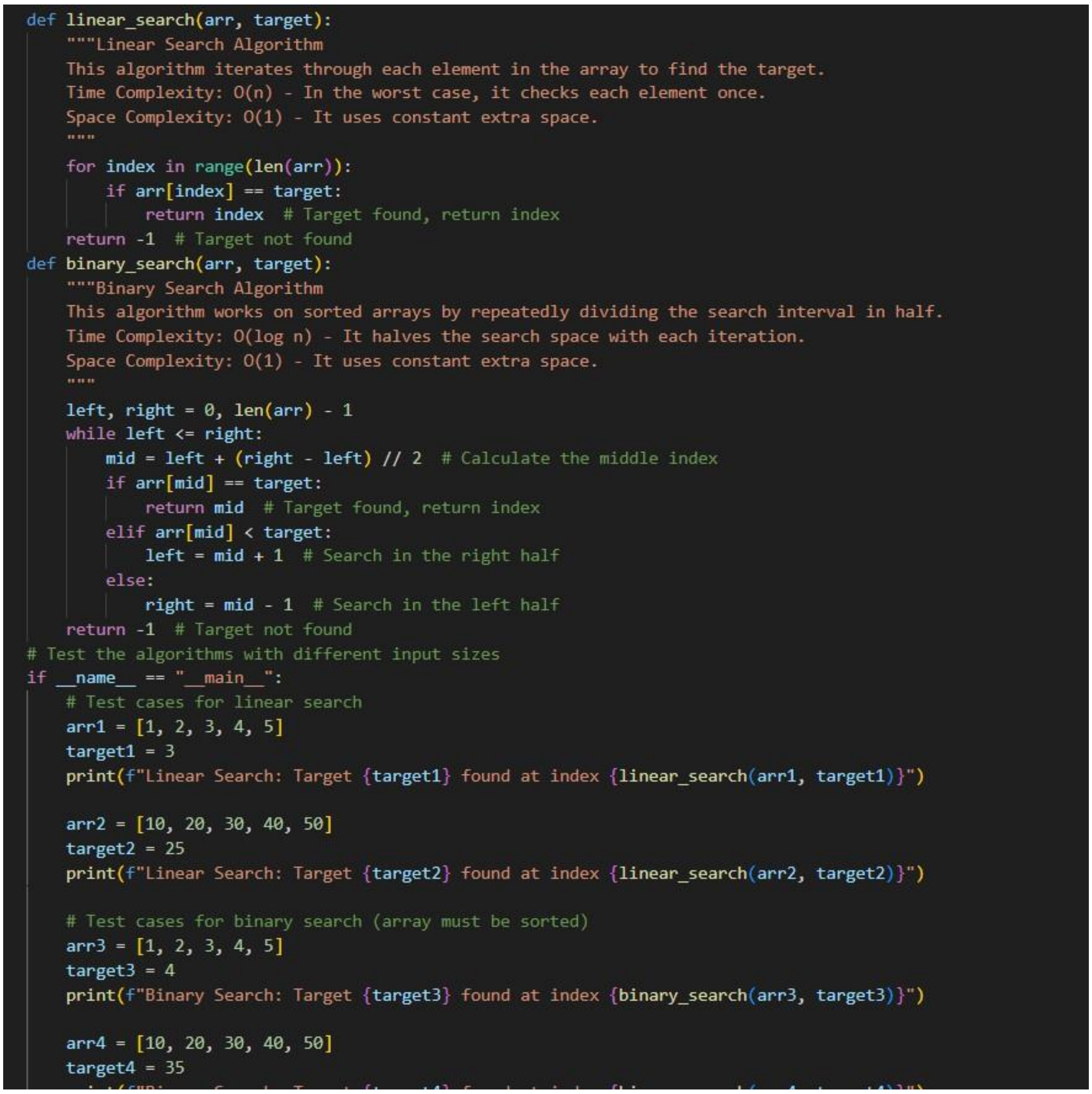
➢ Test both algorithms using different input sizes.

Expected Output:

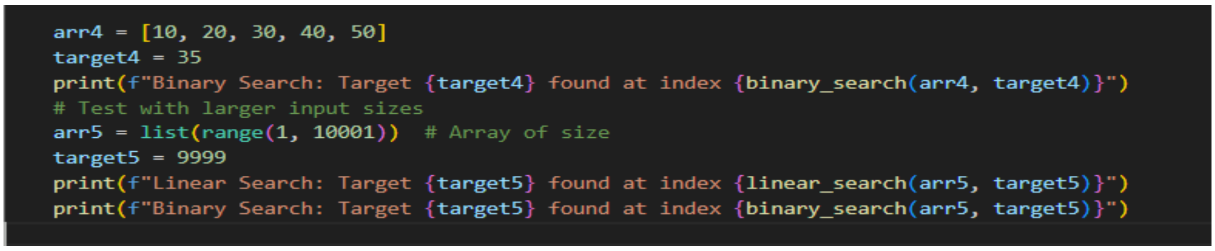
• Python implementations of both search algorithms.

• AI-generated comments and complexity analysis.

• Test results showing correctness and comparison.

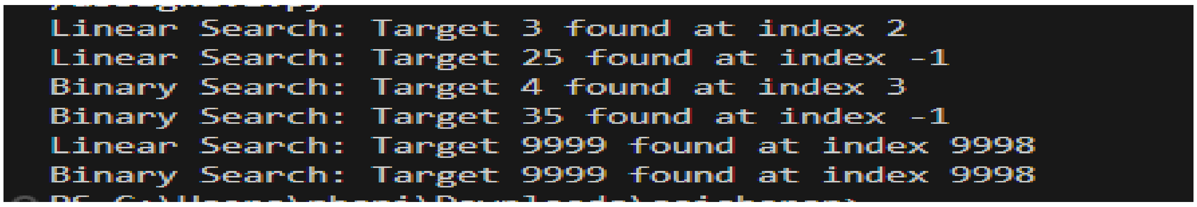


```python
def linear_search(arr, target):
    """Linear Search Algorithm
    This algorithm iterates through each element in the array to find the target.
    Time Complexity: O(n) - In the worst case, it checks each element once.
    Space Complexity: O(1) - It uses constant extra space.
    """
    for index in range(len(arr)):
        if arr[index] == target:
            return index  # Target found, return index
    return -1  # Target not found
def binary_search(arr, target):
    """Binary Search Algorithm
    This algorithm works on sorted arrays by repeatedly dividing the search interval in half.
    Time Complexity: O(log n) - It halves the search space with each iteration.
    Space Complexity: O(1) - It uses constant extra space.
    """
    left, right = 0, len(arr) - 1
    while left <= right:
        mid = left + (right - left) // 2  # Calculate the middle index
        if arr[mid] == target:
            return mid  # Target found, return index
        elif arr[mid] < target:
            left = mid + 1  # Search in the right half
        else:
            right = mid - 1  # Search in the left half
    return -1  # Target not found
# Test the algorithms with different input sizes
if __name__ == "__main__":
    # Test cases for linear search
    arr1 = [1, 2, 3, 4, 5]
    target1 = 3
    print(f"Linear Search: Target {target1} found at index {linear_search(arr1, target1)}")

    arr2 = [10, 20, 30, 40, 50]
    target2 = 25
    print(f"Linear Search: Target {target2} found at index {linear_search(arr2, target2)}")

    # Test cases for binary search (array must be sorted)
    arr3 = [1, 2, 3, 4, 5]
    target3 = 4
    print(f"Binary Search: Target {target3} found at index {binary_search(arr3, target3)}")

    arr4 = [10, 20, 30, 40, 50]
    target4 = 35
```



```python
    arr4 = [10, 20, 30, 40, 50]
    target4 = 35
    print(f"Binary Search: Target {target4} found at index {binary_search(arr4, target4)}")
    # Test with larger input sizes
    arr5 = list(range(1, 10001))  # Array of size
    target5 = 9999
    print(f"Linear Search: Target {target5} found at index {linear_search(arr5, target5)}")
    print(f"Binary Search: Target {target5} found at index {binary_search(arr5, target5)}")
```

Output:



```
Linear Search: Target 3 found at index 2
Linear Search: Target 25 found at index -1
Binary Search: Target 4 found at index 3
Binary Search: Target 35 found at index -1
Linear Search: Target 9999 found at index 9998
Binary Search: Target 9999 found at index 9998
```

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.

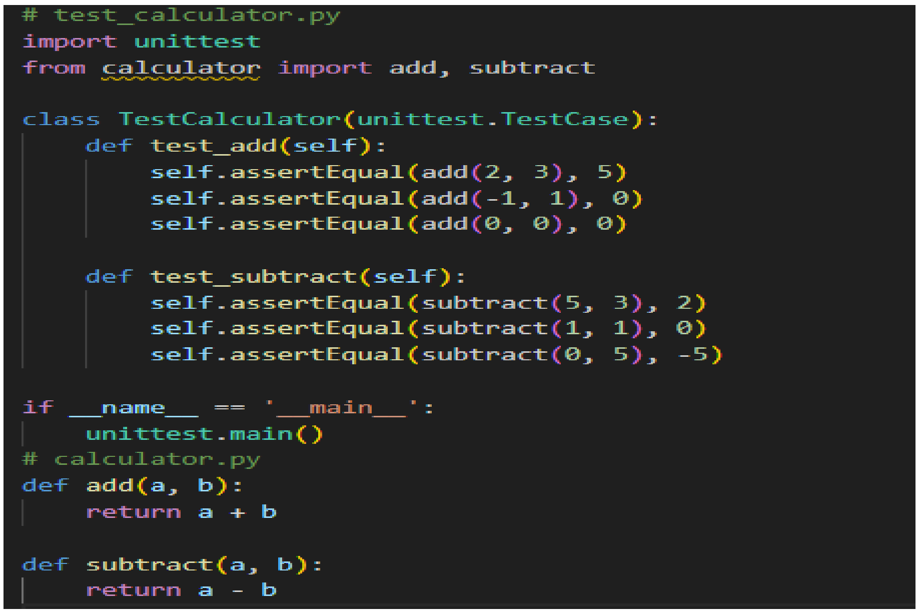
Expected Output:

➢ Separate test file and implementation file.

➢ Test cases executed before implementation.

➢ Final implementation passing all test cases.

Code screenshot:



```python
# test_calculator.py
import unittest
from calculator import add, subtract

class TestCalculator(unittest.TestCase):
    def test_add(self):
        self.assertEqual(add(2, 3), 5)
        self.assertEqual(add(-1, 1), 0)
        self.assertEqual(add(0, 0), 0)

    def test_subtract(self):
        self.assertEqual(subtract(5, 3), 2)
        self.assertEqual(subtract(1, 1), 0)
        self.assertEqual(subtract(0, 5), -5)

if __name__ == '__main__':
    unittest.main()
# calculator.py
def add(a, b):
    return a + b

def subtract(a, b):
    return a - b
```

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

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()

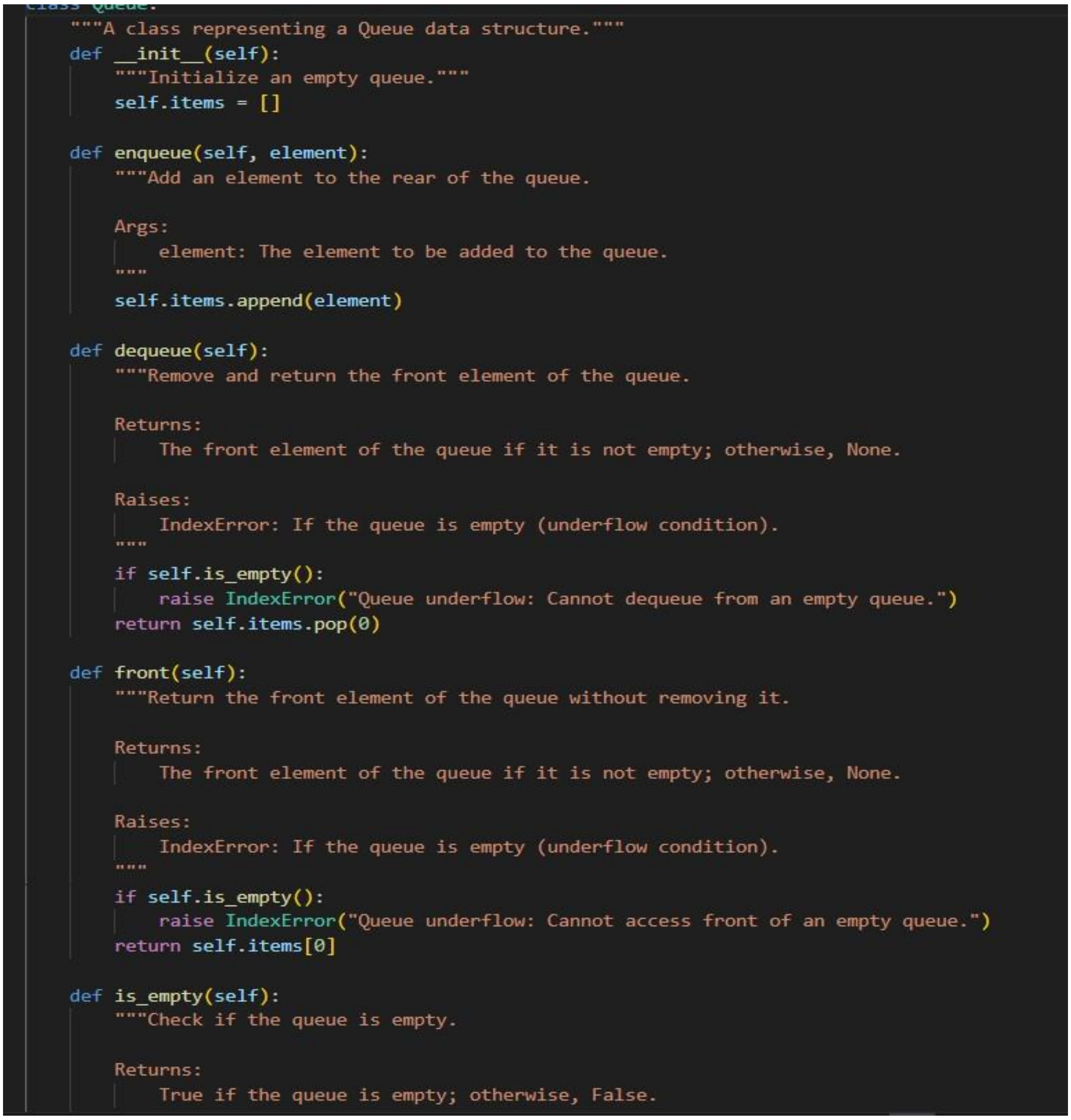
➢ Handle queue overflow and underflow conditions.

➢ Include appropriate docstrings for all methods.

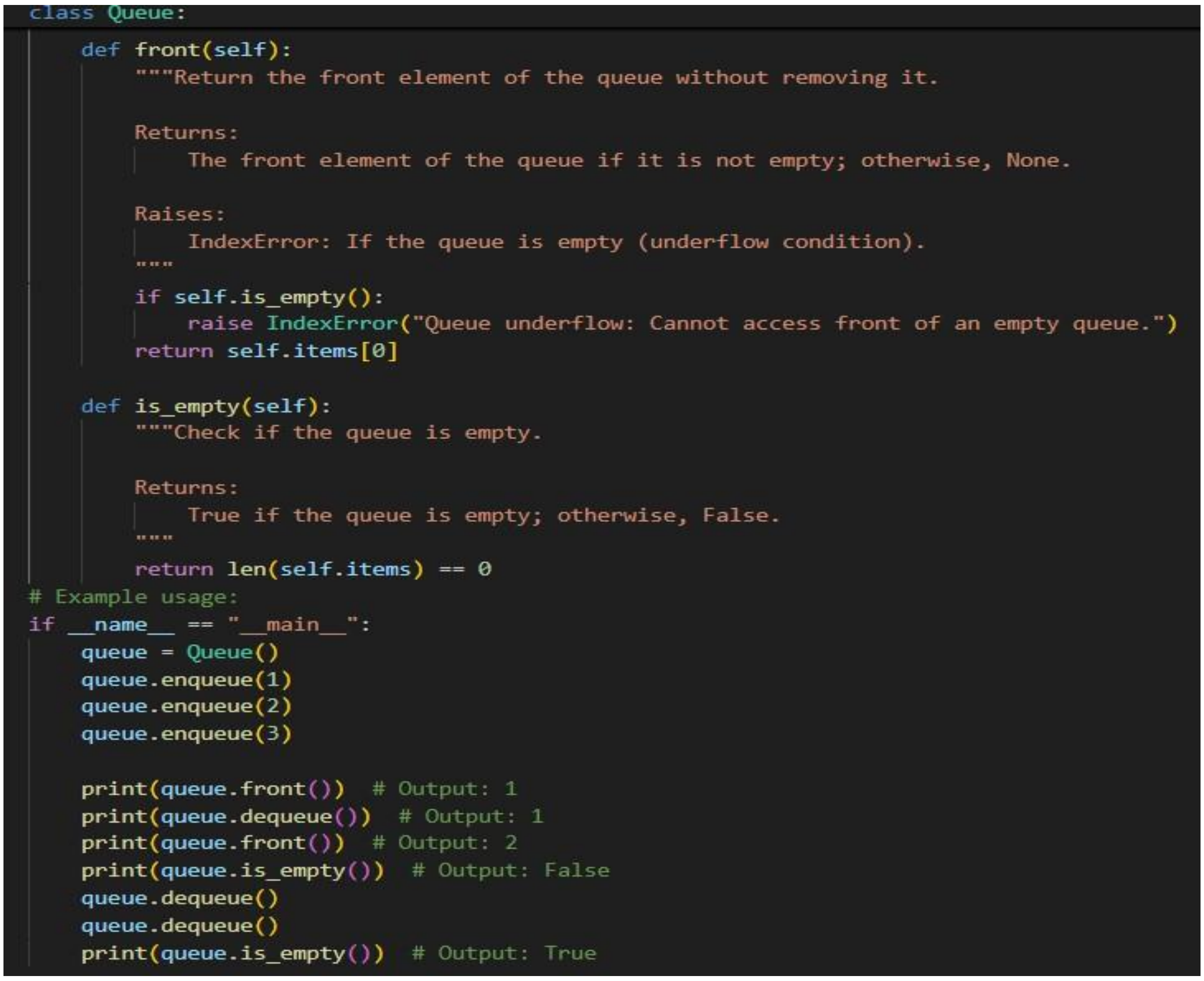
Expected Output:

➢ A fully functional Queue implementation in Python.

Code output:

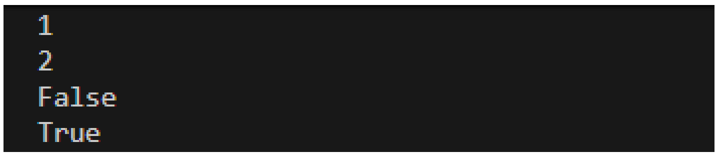


```python
class Queue:
    """A class representing a Queue data structure."""
    def __init__(self):
        """Initialize an empty queue."""
        self.items = []

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

        Args:
            element: The element to be added to the queue.
        """
        self.items.append(element)

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

        Returns:
            The front element of the queue if it is not empty; otherwise, None.

        Raises:
            IndexError: If the queue is empty (underflow condition).
        """
        if self.is_empty():
            raise IndexError("Queue underflow: Cannot dequeue from an empty queue.")
        return self.items.pop(0)

    def front(self):
        """Return the front element of the queue without removing it.

        Returns:
            The front element of the queue if it is not empty; otherwise, None.

        Raises:
            IndexError: If the queue is empty (underflow condition).
        """
        if self.is_empty():
            raise IndexError("Queue underflow: Cannot access front of an empty queue.")
        return self.items[0]

    def is_empty(self):
        """Check if the queue is empty.

        Returns:
            True if the queue is empty; otherwise, False.
```



```python
class Queue:
    def front(self):
        """Return the front element of the queue without removing it.

        Returns:
            The front element of the queue if it is not empty; otherwise, None.

        Raises:
            IndexError: If the queue is empty (underflow condition).
        """
        if self.is_empty():
            raise IndexError("Queue underflow: Cannot access front of an empty queue.")
        return self.items[0]

    def is_empty(self):
        """Check if the queue is empty.

        Returns:
            True if the queue is empty; otherwise, False.
        """
        return len(self.items) == 0
# Example usage:
if __name__ == "__main__":
    queue = Queue()
    queue.enqueue(1)
    queue.enqueue(2)
    queue.enqueue(3)

    print(queue.front())  # Output: 1
    print(queue.dequeue())  # Output: 1
    print(queue.front())  # Output: 2
    print(queue.is_empty())  # Output: False
    queue.dequeue()
    queue.dequeue()
    print(queue.is_empty())  # Output: True
```

Output:



```
1
2
False
True
```

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.

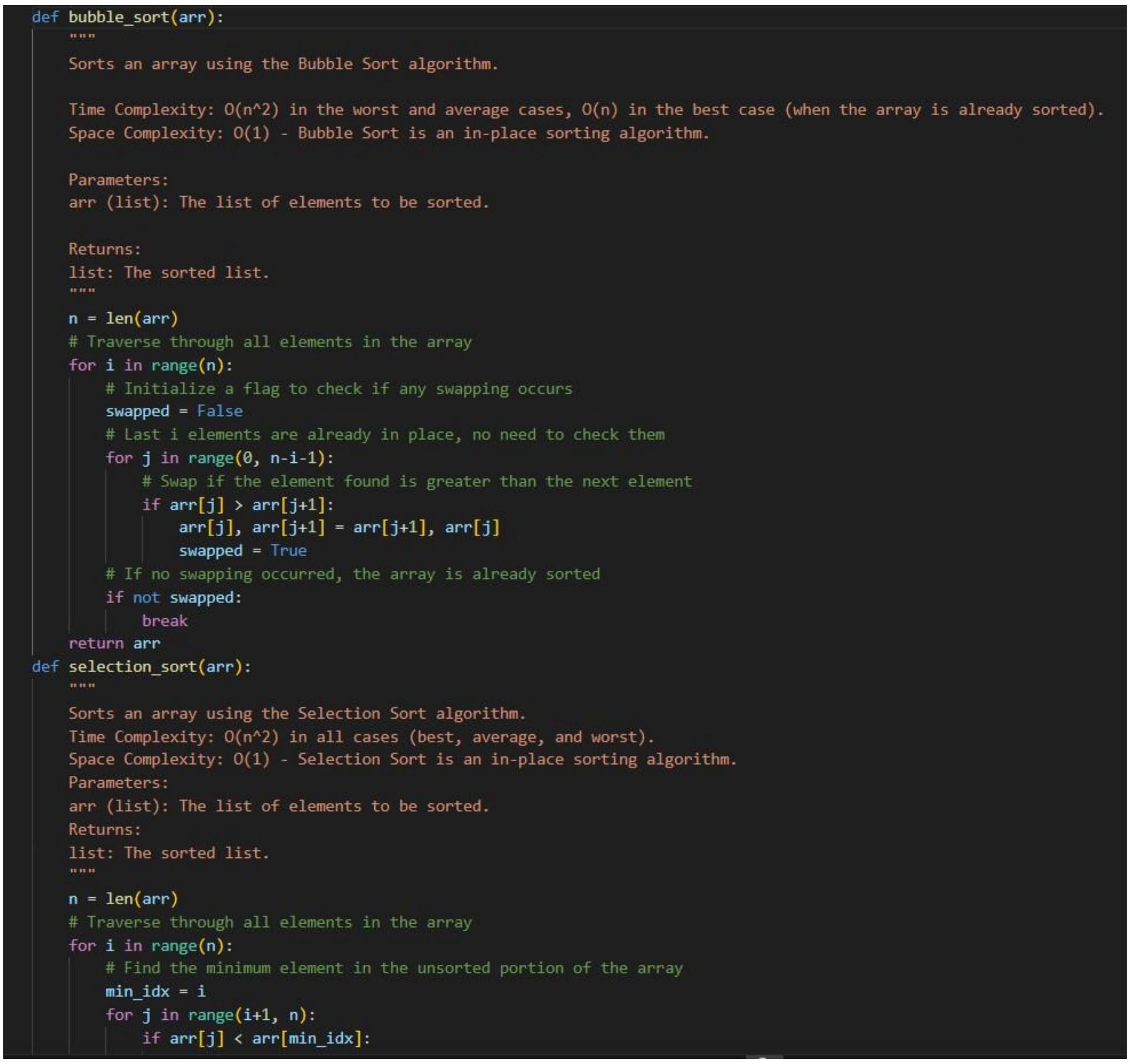
➢ Add docstrings mentioning time and space complexity.

Expected Output:

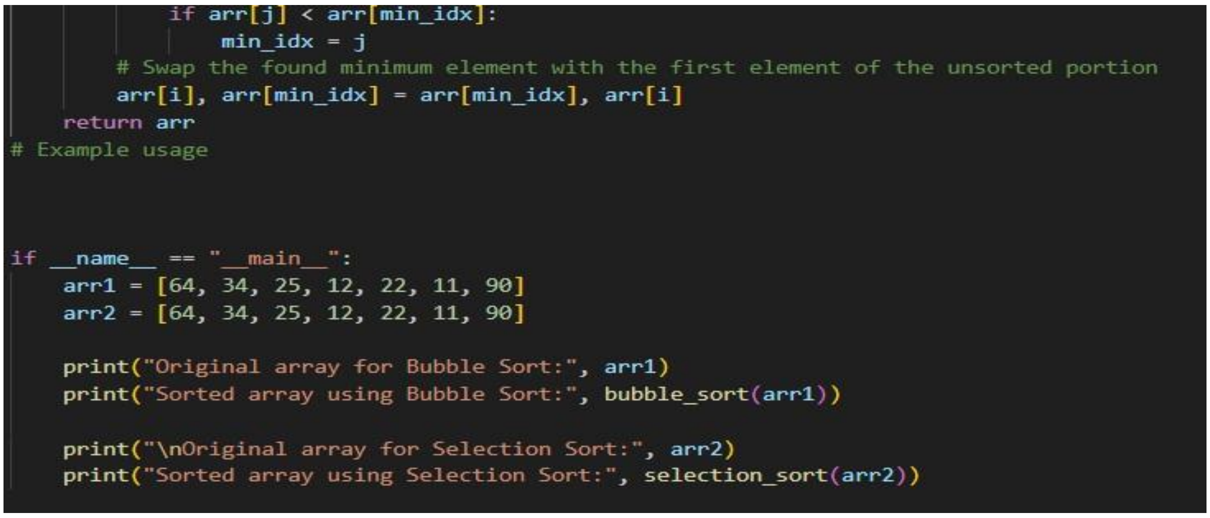
• Correct Python implementations of both sorting algorithms.

• Complexity analysis in docstrings.

Code output:

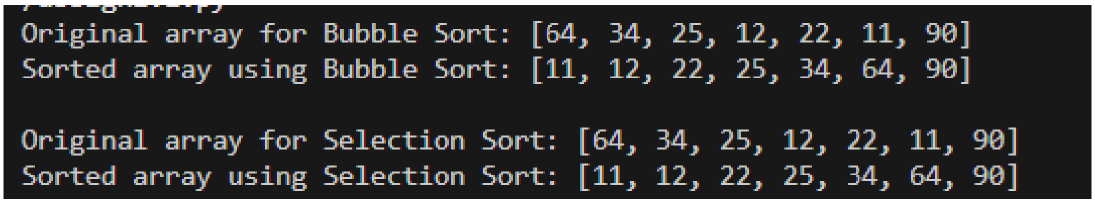


```python
def bubble_sort(arr):
    """
    Sorts an array using the Bubble Sort algorithm.

    Time Complexity: O(n^2) in the worst and average cases, O(n) in the best case (when the array is already sorted).
    Space Complexity: O(1) - Bubble Sort is an in-place sorting algorithm.

    Parameters:
    arr (list): The list of elements to be sorted.

    Returns:
    list: The sorted list.
    """
    n = len(arr)
    # Traverse through all elements in the array
    for i in range(n):
        # Initialize a flag to check if any swapping occurs
        swapped = False
        # Last i elements are already in place, no need to check them
        for j in range(0, n-i-1):
            # Swap if the element found is greater than the next element
            if arr[j] > arr[j+1]:
                arr[j], arr[j+1] = arr[j+1], arr[j]
                swapped = True
        # If no swapping occurred, the array is already sorted
        if not swapped:
            break
    return arr
def selection_sort(arr):
    """
    Sorts an array using the Selection Sort algorithm.
    Time Complexity: O(n^2) in all cases (best, average, and worst).
    Space Complexity: O(1) - Selection Sort is an in-place sorting algorithm.
    Parameters:
    arr (list): The list of elements to be sorted.
    Returns:
    list: The sorted list.
    """
    n = len(arr)
    # Traverse through all elements in the array
    for i in range(n):
        # Find the minimum element in the unsorted portion of the array
        min_idx = i
        for j in range(i+1, n):
            if arr[j] < arr[min_idx]:
```



```
            if arr[j] < arr[min_idx]:
                min_idx = j
        # Swap the found minimum element with the first element of the unsorted portion
        arr[i], arr[min_idx] = arr[min_idx], arr[i]
    return arr
# Example usage



if __name__ == "__main__":
    arr1 = [64, 34, 25, 12, 22, 11, 90]
    arr2 = [64, 34, 25, 12, 22, 11, 90]

    print("Original array for Bubble Sort:", arr1)
    print("Sorted array using Bubble Sort:", bubble_sort(arr1))

    print("\nOriginal array for Selection Sort:", arr2)
    print("Sorted array using Selection Sort:", selection_sort(arr2))
```

Output:



```
Original array for Bubble Sort: [64, 34, 25, 12, 22, 11, 90]
Sorted array using Bubble Sort: [11, 12, 22, 25, 34, 64, 90]

Original array for Selection Sort: [64, 34, 25, 12, 22, 11, 90]
Sorted array using Selection Sort: [11, 12, 22, 25, 34, 64, 90]
```