Assignment-12.2

Name: Adepu Tejaswini

Ht.no: 2303A51423

Batch. No: 21

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

Prompt:

Generate a Python program that implements a Stack data structure using a class. Include the methods push(element), pop(), peek(), and is_empty(). Add proper error handling for stack underflow and include clear docstrings for each method explaining their functionality. Take input from user.

Code:

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

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

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

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

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

        Returns:
            The top element 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):
        """Return the top element of the stack without removing it.

        Returns:
            The top element of the stack.

        Raises:
            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):
        """Check if the stack is empty.

        Returns:
            True if the stack is empty, False otherwise.
        """
        return len(self.stack) == 0
def main():
    stack = Stack()
    while True:
        print("\nStack Operations:")
        print("1. Push")
        print("2. Pop")
        print("3. Peek")
        print("4. Check if Empty")
        print("5. Exit")

        choice = input("Enter your choice (1-5): ")

        if choice == '1':
            element = input("Enter the element to push: ")
            stack.push(element)
            print(f"'{element}' has been pushed to the stack.")
        elif choice == '2':
            try:
                popped_element = stack.pop()
                print(f"'{popped_element}' has been popped from the stack.")
            except IndexError as e:
                print(e)
        elif choice == '3':
            try:
                top_element = stack.peek()
                print(f"The top element is: '{top_element}'")
            except IndexError as e:
                print(e)
        elif choice == '4':
            if stack.is_empty():
                print("The stack is empty.")
            else:
                print("The stack is not empty.")
        elif choice == '5':
            print("Exiting the program.")
            break
        else:
            print("Invalid choice. Please enter a number between 1 and 5.")
if __name__ == "__main__":
    main()
```

Output:

```
PS C:\Users\AdepuTejaswini\AI assist> python -u "c:\Users\AdepuTejaswini\AI assist\code-12.py"
Stack Operations:
1. Push
2. Pop
3. Peek
4. Check if Empty
5. Exit
Enter your choice (1-5): 1
Enter the element to push: 10
'10' has been pushed to the stack.

Stack Operations:
1. Push
2. Pop
3. Peek
4. Check if Empty
5. Exit
Enter your choice (1-5): 3
The top element is: '10'

Stack Operations:
1. Push
2. Pop
3. Peek
4. Check if Empty
5. Exit
Enter your choice (1-5): 2
'10' has been popped from the stack.

Stack Operations:
1. Push
2. Pop
3. Peek
4. Check if Empty
5. Exit
Enter your choice (1-5): 4
The stack is empty.

Stack Operations:
1. Push
2. Pop
3. Peek
4. Check if Empty
5. Exit
Enter your choice (1-5): 5
Exiting the program.
```

Justification:

A **Stack** is used because it follows the **Last In First Out (LIFO)** principle, which is suitable for operations where the most recent element must be accessed first. Stacks allow efficient insertion and deletion operations using **push and pop** in constant time **O(1)**.
Other data structures like **queues or lists** do not strictly follow the LIFO order required for stack-based operations.

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

Prompt:

Generate Python code to implement Linear Search and Binary Search with functions:

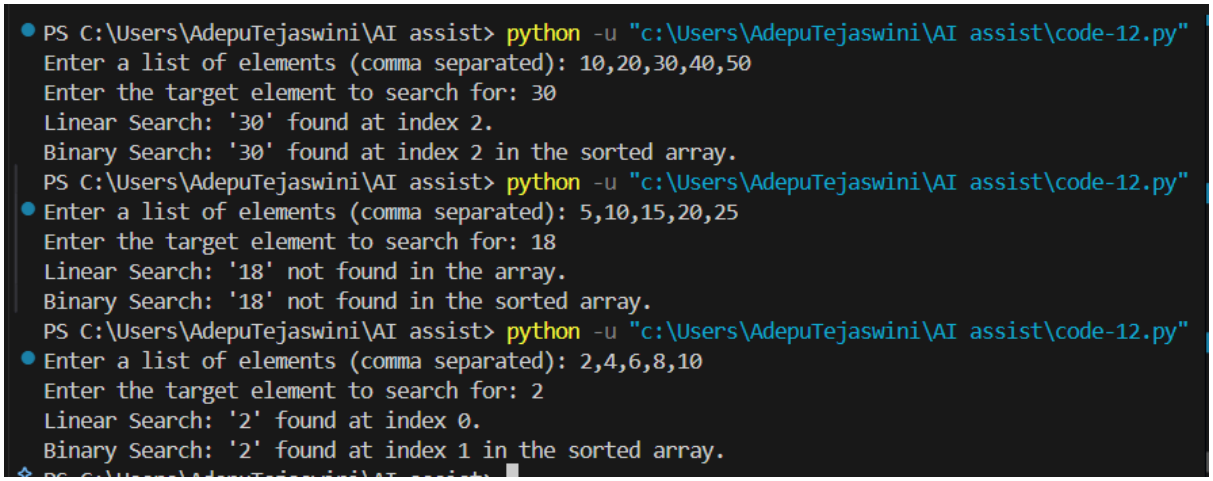
linear_search(arr, target) and binary_search(arr, target).

Include Proper docstrings explaining the working principle.Test the algorithms with different input sizes (from user) and print the results for comparison.

Code:

```python
def linear_search(arr, target):
    """Perform a linear search for the target element in the array.

    Args:
        arr: A list of elements to search through.
        target: The element to search for.

    Returns:
        The index of the target element if found, otherwise -1.
    """
    for index in range(len(arr)):
        if arr[index] == target:
            return index
    return -1
def binary_search(arr, target):
    """Perform a binary search for the target element in a sorted array.

    Args:
        arr: A sorted list of elements to search through.
        target: The element to search for.

    Returns:
        The index of the target element if found, otherwise -1.
    """
    left, right = 0, len(arr) - 1
    while left <= right:
        mid = left + (right - left) // 2
        if arr[mid] == target:
            return mid
        elif arr[mid] < target:
            left = mid + 1
        else:
            right = mid - 1
    return -1
def main():
    arr = input("Enter a list of elements (comma separated): ").split(',')
    target = input("Enter the target element to search for: ")

    # Test Linear Search
    linear_result = linear_search(arr, target)
    if linear_result != -1:
        print(f"Linear Search: '{target}' found at index {linear_result}.")
    else:
        print(f"Linear Search: '{target}' not found in the array.")

    # Sort the array for Binary Search
    sorted_arr = sorted(arr)

    # Test Binary Search
    binary_result = binary_search(sorted_arr, target)
    if binary_result != -1:
        print(f"Binary Search: '{target}' found at index {binary_result} in the sorted array.")
    else:
        print(f"Binary Search: '{target}' not found in the sorted array.")
if __name__ == "__main__":
    main()
```

Output:



Justification:

Linear Search and Binary Search use the **Array/List data structure** because it allows easy sequential and indexed access to elements.
Binary Search specifically requires a **sorted array** to efficiently divide the search space.
Other data structures like **linked lists or trees** are not suitable because they do not support efficient direct indexing required for Binary Search.

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

Prompt:

Generate unit test cases first for a simple calculator in Python using unittest.

The calculator should support two operations: addition and subtraction.Take input from user for the numbers and the operation to perform.

Create a separate test file named test_calculator.py.

Do not implement the calculator functions yet.

Code:

```python
import unittest
from calculator import add, subtract
class TestCalculator(unittest.TestCase):
    """Unit test cases for the simple calculator functions."""

    def test_add(self):
        """Test the addition function."""
        self.assertEqual(add(2, 3), 5)
        self.assertEqual(add(-1, 1), 0)
        self.assertEqual(add(0, 0), 0)

    def test_subtract(self):
        """Test the subtraction function."""
        self.assertEqual(subtract(5, 2), 3)
        self.assertEqual(subtract(0, 5), -5)
        self.assertEqual(subtract(-1, -1), 0)
if __name__ == '__main__':
    unittest.main()
```

Now implement the calculator functions add() and subtract() in calculator.py so that all the existing unit tests pass.

```python
import unittest
from calculator import add, subtract

class TestCalculator(unittest.TestCase):

    def test_add_positive(self):
        self.assertEqual(add(5, 3), 8)

    def test_add_negative(self):
        self.assertEqual(add(-2, -3), -5)

    def test_add_zero(self):
        self.assertEqual(add(0, 5), 5)

    def test_subtract_positive(self):
        self.assertEqual(subtract(10, 4), 6)

    def test_subtract_negative(self):
        self.assertEqual(subtract(-5, -2), -3)

    def test_subtract_zero(self):
        self.assertEqual(subtract(5, 0), 5)

if __name__ == "__main__":
    unittest.main()
```

Output:

```
PS C:\Users\AdepuTejaswini\AI assist> python -u "c:\Users\AdepuTejaswini\AI assist\code-12.py"
..
----------------------------------------------------------------------
Ran 2 tests in 0.001s

OK
PS C:\Users\AdepuTejaswini\AI assist> python -m unittest test_calculator.py
......
----------------------------------------------------------------------
Ran 6 tests in 0.001s

OK
PS C:\Users\AdepuTejaswini\AI assist>
```

Justification:

Test Driven Development ensures that test cases are written before implementing the actual functionality. It helps verify that the program meets requirements and reduces bugs during development. Using AI tools like Copilot speeds up writing tests and implementation while maintaining code correctness.

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

Prompt:

Generate a Python program that implements a Queue data structure using a class.

Include methods enqueue(element), dequeue(), front(), and is_empty().

Handle queue overflow and underflow conditions and include proper docstrings for each method.

Take input from user to perform queue operations.

Code:

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

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

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

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

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

        Returns:
            The front element of the queue.

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

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

        Returns:
            The front element of the queue.

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

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

        Returns:
            True if the queue is empty, False otherwise.
        """
        return len(self.queue) == 0
def main():
    queue = Queue()
    while True:
        print("\nQueue Operations:")
        print("1. Enqueue")
        print("2. Dequeue")
        print("3. Front")
        print("4. Check if Empty")
        print("5. Exit")

        choice = input("Enter your choice (1-5): ")

        if choice == '1':
            element = input("Enter the element to enqueue: ")
            queue.enqueue(element)
            print(f"'{element}' has been enqueued to the queue.")
        elif choice == '2':
            try:
                dequeued_element = queue.dequeue()
                print(f"'{dequeued_element}' has been dequeued from the queue.")
            except IndexError as e:
                print(e)
        elif choice == '3':
            try:
                front_element = queue.front()
                print(f"The front element is: '{front_element}'")
            except IndexError as e:
                print(e)
        elif choice == '4':
            if queue.is_empty():
                print("The queue is empty.")
            else:
                print("The queue is not empty.")
        elif choice == '5':
            print("Exiting the program.")
            break
        else:
            print("Invalid choice. Please enter a number between 1 and 5.")
if __name__ == "__main__":
    main()
```

Output:

Justification:

Queue is suitable because it follows the **FIFO (First In First Out)** principle, which is ideal for processing elements in the order they arrive. It is commonly used in **task scheduling, buffering, and request handling systems**. Other data structures like **Stack (LIFO) or Linked List** do not naturally maintain FIFO order for processing elements.

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

Prompt:

Write Python functions bubble_sort(arr) and selection_sort(arr) with comments explaining each step.

Include docstrings mentioning time complexity and space complexity.

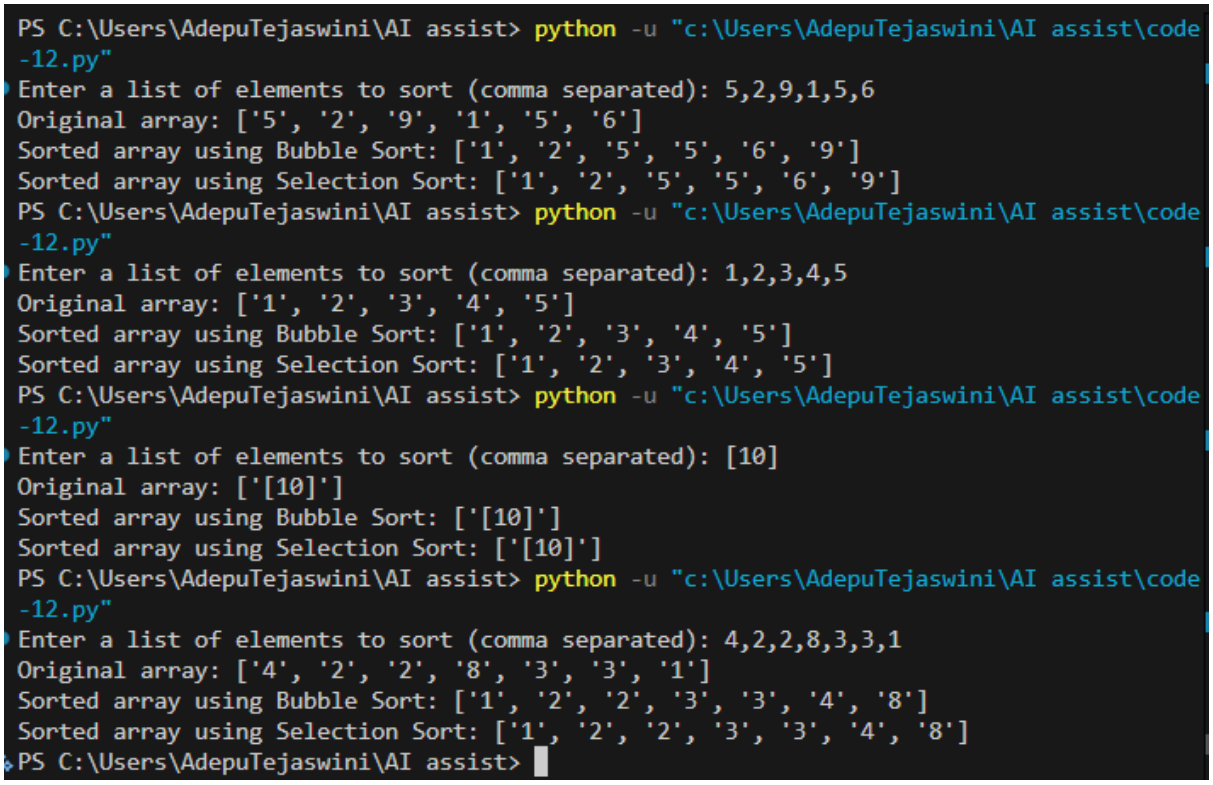
Return the sorted array and demonstrate the functions with sample test cases.

Take input from user for the array to be sorted.

Code:

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

    Args:
        arr: A list of elements to be sorted.

    Returns:
        A sorted list of elements.

    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) (in-place sorting).
    """
    n = len(arr)
    # Traverse through all elements in the array
    for i in range(n):
        # 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]
    return arr
def selection_sort(arr):
    """Sort an array using the Selection Sort algorithm.

    Args:
        arr: A list of elements to be sorted.

    Returns:
        A sorted list of elements.

    Time Complexity: O(n^2) in all cases (best, average, and worst).
    Space Complexity: O(1) (in-place sorting).
    """
    n = len(arr)
    # Traverse through all elements in the array
    for i in range(n):
        # Find the minimum element in the remaining unsorted array
        min_index = i
        for j in range(i + 1, n):
            if arr[j] < arr[min_index]:
                min_index = j
        # Swap the found minimum element with the first element of the unsorted array
        arr[i], arr[min_index] = arr[min_index], arr[i]
    return arr
def main():
    arr = input("Enter a list of elements to sort (comma separated): ").split(',')
    print("Original array:", arr)

    sorted_arr_bubble = bubble_sort(arr.copy())
    print("Sorted array using Bubble Sort:", sorted_arr_bubble)

    sorted_arr_selection = selection_sort(arr.copy())
    print("Sorted array using Selection Sort:", sorted_arr_selection)
if __name__ == "__main__":
    main()
```

Output:



Justification:

Sorting algorithms like **Bubble Sort and Selection Sort require indexed access**, which is efficiently supported by arrays (Python lists). Arrays allow **easy swapping of elements using indices**, which is essential for these algorithms. Other data structures like **linked lists, stacks, or queues are inefficient for direct element comparison and swapping**, making arrays the most suitable choice.