AI ASSIST CODING ASSIGNMENT

Program Name: B. Tech
Course Code: 23CS002PC304
Course Title: AI Assisted Coding
Year/Sem: III/II
Regulation: R23
Student Name: seetharam
Roll No: 2303A52440
Assignment Number: 12.1 / 24

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

Task 1 – Data Structures: Stack Implementation with AI Assistance

Prompt Used

Generate a Python program that implements a Stack class with push(), pop(), peek(), and is_empty() methods. Include proper error handling for stack underflow and clear docstrings for each method.

Python Implementation

```python
class Stack:
    """
    A simple Stack implementation using a Python list.
    """

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

    def push(self, element):
        """Push an element onto 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 (underflow)."""
        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."""
        if self.is_empty():
            raise IndexError("Stack is empty.")
        return self.items[-1]

    def is_empty(self):
        """Return True if the stack is empty, otherwise False."""
        return len(self.items) == 0


# Testing the Stack
stack = Stack()
stack.push(10)
stack.push(20)
print("Top element:", stack.peek())
print("Popped element:", stack.pop())
print("Is stack empty?", stack.is_empty())
```

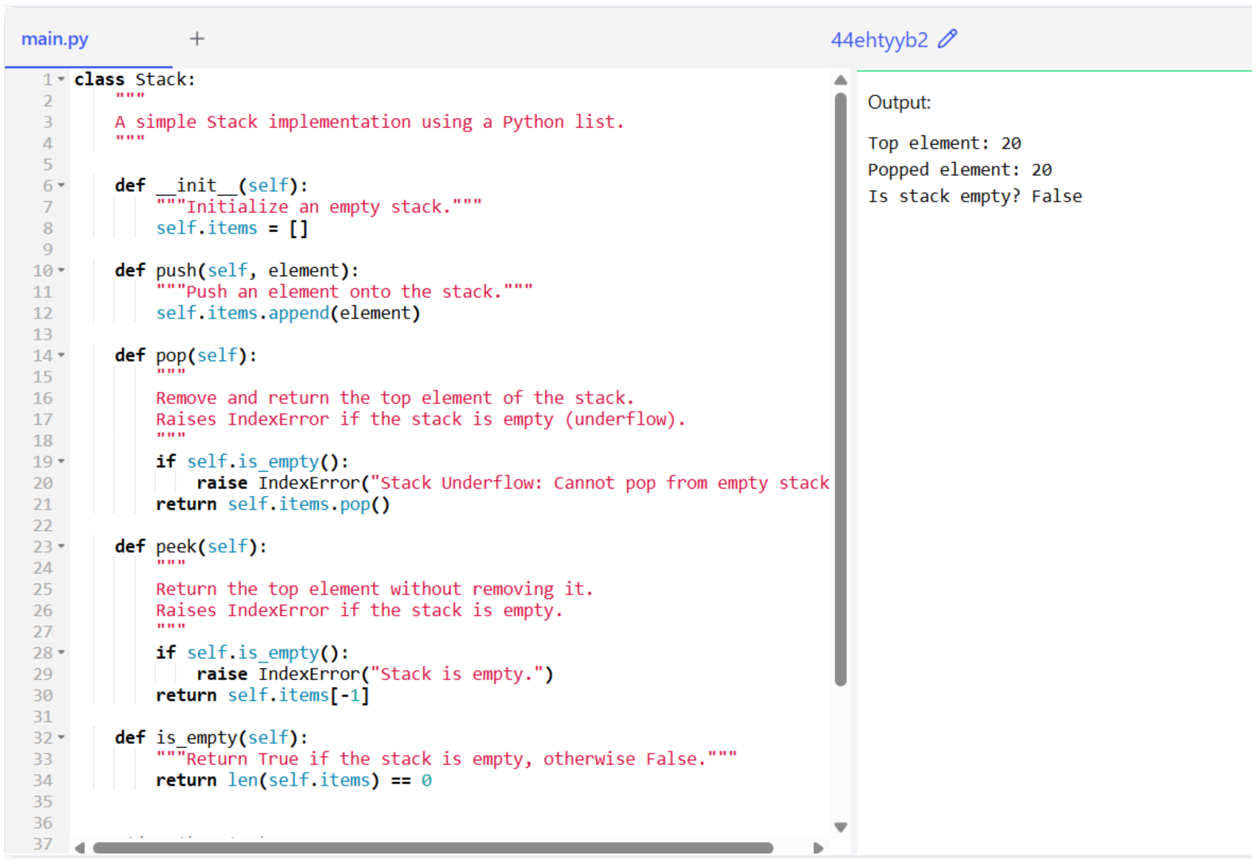
Explanation

- Implemented Stack using a list.
- Included push, pop, peek, and is_empty methods.
- Added proper error handling for stack underflow using exceptions.
- Included clear docstrings for each method.

Output

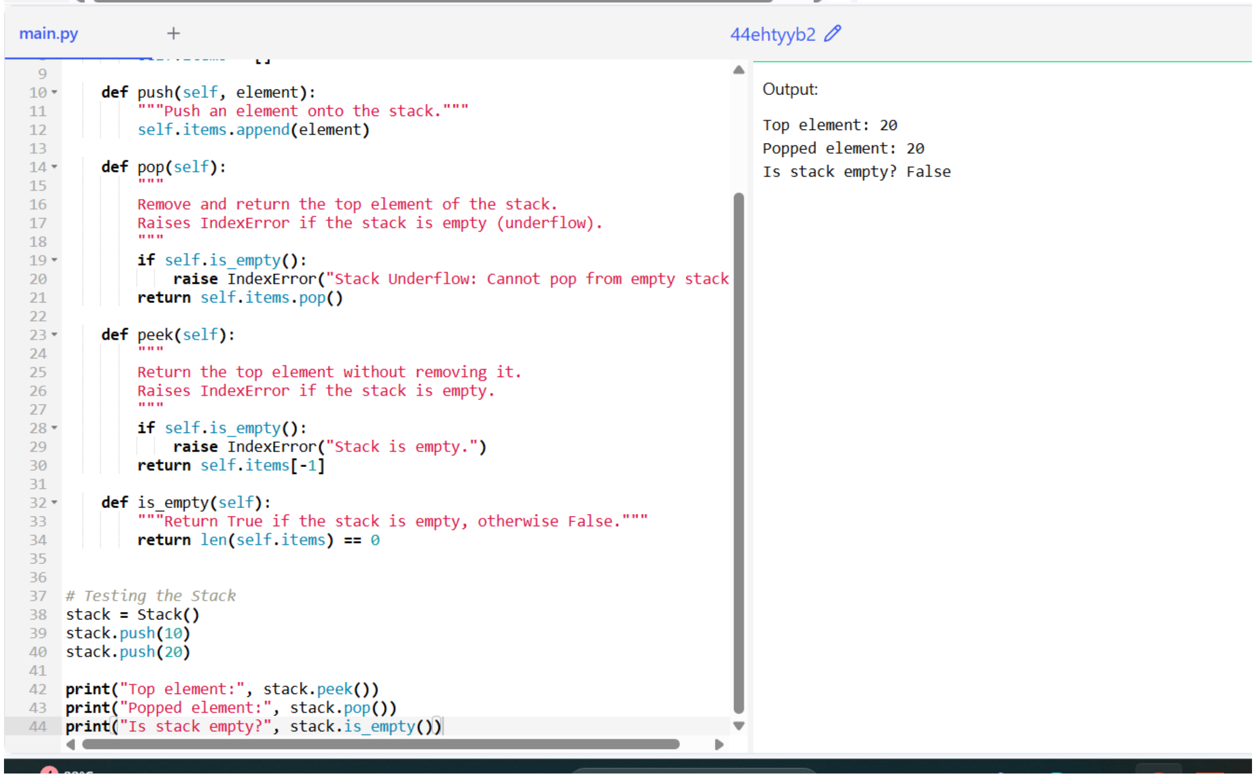
Top element: 20 Popped element: 20 Is stack empty? False

Screenshot of Execution



```python
class Stack:
    """
    A simple Stack implementation using a Python list.
    """

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

    def push(self, element):
        """Push an element onto 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 (underflow).
        """
        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.
        """
        if self.is_empty():
            raise IndexError("Stack is empty.")
        return self.items[-1]

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

```
Output:

Top element: 20
Popped element: 20
Is stack empty? False
```



```python
    def push(self, element):
        """Push an element onto 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 (underflow).
        """
        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.
        """
        if self.is_empty():
            raise IndexError("Stack is empty.")
        return self.items[-1]

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

```python
# Testing the Stack
stack = Stack()
stack.push(10)
stack.push(20)

print("Top element:", stack.peek())
print("Popped element:", stack.pop())
print("Is stack empty?", stack.is_empty())
```

```
Output:

Top element: 20
Popped element: 20
Is stack empty? False
```

Task 2 – Algorithms: Linear vs Binary Search Analysis

Prompt Used

Generate Python implementations of linear_search(arr, target) and binary_search(arr, target). Include docstrings explaining working principle and perform testing with different input sizes.

Python Implementation

```python
def linear_search(arr, target):
    """
    Perform Linear Search.
    Checks each element sequentially until target is found.
    Time Complexity: O(n)
    """
    for i in range(len(arr)):
        if arr[i] == target:
            return i
    return -1


def binary_search(arr, target):
    """
    Perform Binary Search.
    Works only on sorted arrays. Divides search space in half each step.
    Time Complexity: O(log n)
    """
    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


# Testing
arr_small = [1, 3, 5, 7, 9]
arr_large = list(range(1, 10001))
```

```python
print("Linear Search (small array):", linear_search(arr_small, 7))
print("Binary Search (small array):", binary_search(arr_small, 7))

print("Linear Search (large array):", linear_search(arr_large, 9999))
print("Binary Search (large array):", binary_search(arr_large, 9999))
```

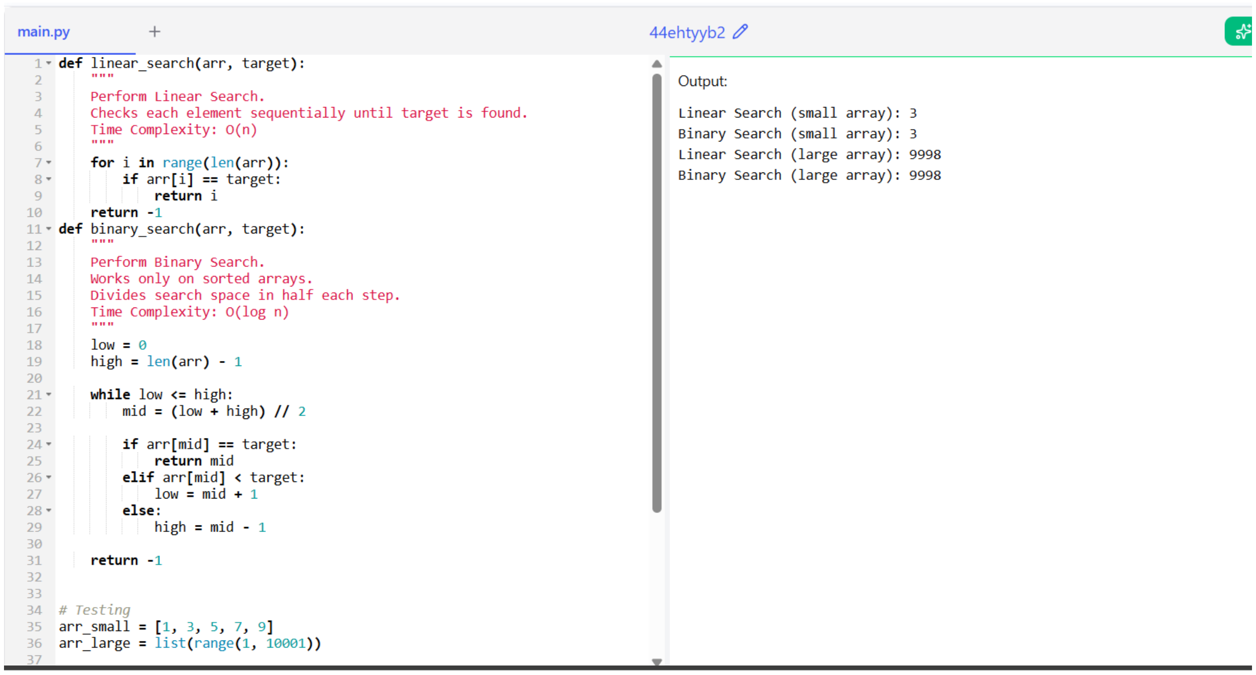
Explanation

- Linear Search checks elements sequentially.
- Binary Search divides the sorted array into halves.
- Binary Search is more efficient for large sorted datasets.
- Complexity Comparison:
    - Linear Search: O(n)
    - Binary Search: O(log n)

Output

Linear Search (small array): 3 Binary Search (small array): 3 Linear Search (large array): 9998 Binary Search (large array): 9998

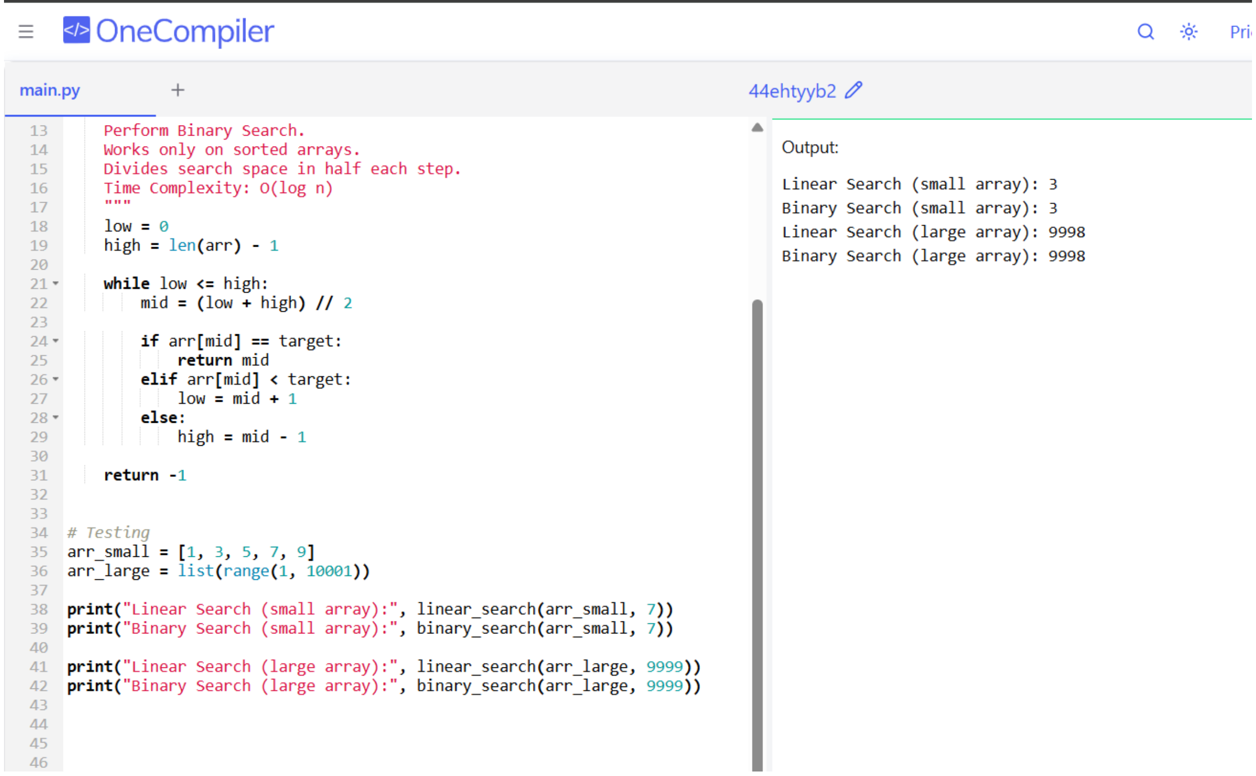
Screenshot of Execution



```python
def linear_search(arr, target):
    """
    Perform Linear Search.
    Checks each element sequentially until target is found.
    Time Complexity: O(n)
    """
    for i in range(len(arr)):
        if arr[i] == target:
            return i
    return -1
def binary_search(arr, target):
    """
    Perform Binary Search.
    Works only on sorted arrays.
    Divides search space in half each step.
    Time Complexity: O(log n)
    """
    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


# Testing
arr_small = [1, 3, 5, 7, 9]
arr_large = list(range(1, 10001))
```

Output:

```
Linear Search (small array): 3
Binary Search (small array): 3
Linear Search (large array): 9998
Binary Search (large array): 9998
```



```python
    Perform Binary Search.
    Works only on sorted arrays.
    Divides search space in half each step.
    Time Complexity: O(log n)
    """
    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


# Testing
arr_small = [1, 3, 5, 7, 9]
arr_large = list(range(1, 10001))

print("Linear Search (small array):", linear_search(arr_small, 7))
print("Binary Search (small array):", binary_search(arr_small, 7))

print("Linear Search (large array):", linear_search(arr_large, 9999))
print("Binary Search (large array):", binary_search(arr_large, 9999))
```

Output:

```
Linear Search (small array): 3
Binary Search (small array): 3
Linear Search (large array): 9998
Binary Search (large array): 9998
```

Comparison and Analysis

- Linear Search is simple and works on unsorted data.
- Binary Search requires sorted data but is significantly faster.
- AI assistance helped generate structured code, docstrings, and complexity analysis quickly.

Conclusion

This lab demonstrated how AI tools can assist in implementing data structures and algorithms efficiently. The Stack implementation included proper error handling and documentation. The comparison between Linear and Binary Search highlighted differences in performance and complexity, showing how optimized algorithms improve efficiency for large datasets.