# Lab-12.2

# Course : AI Assisted Coding

*Topic:- Algorithms with AI Assistance – Sorting, Searching, and Optimizing Algorithms*

---

Student Details:

Name : Mohammed Sabir

Hall Ticket No : 2303A51506

Batch : 22

---

## Task-01

**Final Optimal Prompt:**

generate a python code that implements fully a stack data structure which it has methods like push,pop,peek and is_empty

and insure that the code is well documented without docstrings for each method and the class itself.

and ensure that code handle error like stack overflow and stack underflow

---

**Code Screenshot :**

```python
class Stack:
    def __init__(self, capacity):
        self.capacity = capacity  # Maximum number of items the stack can hold
        self.stack = []  # List to store stack items

    def push(self, item):
        if len(self.stack) >= self.capacity:
            raise OverflowError("Stack overflow: Cannot push item, stack is full.")
        self.stack.append(item)  # Add item to the top of the stack

    def pop(self):
        if self.is_empty():
            raise IndexError("Stack underflow: Cannot pop item, stack is empty.")
        return self.stack.pop()  # Remove and return the top item from the stack

    def peek(self):
        if self.is_empty():
            raise IndexError("Stack underflow: Cannot peek item, stack is empty.")
        return self.stack[-1]  # Return the top item without removing it

    def is_empty(self):
        return len(self.stack) == 0  # Check if the stack is empty
# Example usage:
if __name__ == "__main__":
    stack = Stack(5)  # Create a stack with a capacity of 5
    stack.push(1)
    stack.push(2)
    stack.push(3)
    print(stack.peek())  # Output: 3
    print(stack.pop())  # Output: 3
    print(stack.is_empty())  # Output: False
    stack.pop()
    stack.pop()
    print(stack.is_empty())  # Output: True
    # Testing stack overflow
    try:
        stack.push(1)
        stack.push(2)
        stack.push(3)
        stack.push(4)
        stack.push(5)
        stack.push(6)  # This will raise an OverflowError
    except OverflowError as e:
        print(e)  # Output: Stack overflow: Cannot push item, stack is full.
    # Testing stack underflow    try:
        stack.pop()  # This will raise an IndexError since the stack is empty
    except IndexError as e:
        print(e)  # Output: Stack underflow: Cannot pop item, stack is empty.
    try:
        stack.peek()  # This will raise an IndexError since the stack is empty
    except IndexError as e:
        print(e)  # Output: Stack underflow: Cannot peek item, stack is empty.
```

**Output Screenshot:**

```
PS C:\Users\sakir\OneDrive\Desktop\Ai-Assistant> ^C
PS C:\Users\sakir\OneDrive\Desktop\Ai-Assistant>  c:; cd 'c:\Users\sakir\OneDrive\Desktop\Ai-Assistant'; & 'c:\python314\python.exe' 'c:\Users\sakir\.vscode\extensions\ms-python.debugpy-2025.18.0-win32-x64\bundled\libs\debugpy\launcher' '57941' '--' 'C:\Users\sakir\OneDrive\Desktop\Ai-Assistant\lab12.py'
3
3
False
True
Stack overflow: Cannot push item, stack is full.
PS C:\Users\sakir\OneDrive\Desktop\Ai-Assistant>
```

**Explanation/Justification/Observation (100 words / 5 – 6 sentence) :**

The program implements a Stack data structure using Object-Oriented Programming in Python. It defines a fixed capacity stack using a list to store elements. The push() method adds elements while checking for overflow, raising an OverflowError if the stack is full. The pop() and peek() methods handle underflow by raising an IndexError when the stack is empty. The is_empty() method checks whether the stack contains elements. The example usage demonstrates stack operations and exception handling for overflow and underflow conditions. Overall, the program clearly illustrates LIFO behavior, proper error handling, and clean class-based design.

---

## Task-02

**Final Optimal Prompt:**

generati a python code of linear search and binary search and compare between them in terms of time complexity and space complexity and also provide an example for each one of them

and the instruction is linear_search(arr, target) and binary_search(arr, target) and ensure that the code is well documented without docstrings for each method and the class itself.

user should give the input of the array and the target number to search for in the array and the code should handle the case when the target number is not found in the array

---

**Code Screenshot :**

```python
def linear_search(arr, target):
    for i in range(len(arr)):
        if arr[i] == target:
            return i  # Return the index of the target if found
    return -1  # Return -1 if the target is not found
def binary_search(arr, target):
    left, right = 0, len(arr) - 1
    while left <= right:
        mid = left + (right - left) // 2  # Calculate the middle index
        if arr[mid] == target:
            return mid  # Return the index of the target if found
        elif arr[mid] < target:
            left = mid + 1  # Search in the right half
        else:
            right = mid - 1  # Search in the left half
    return -1  # Return -1 if the target is not found
# Example usage:
if __name__ == "__main__":
    arr = list(map(int, input("Enter a sorted array (space-separated): ").split()))
    target = int(input("Enter the target number to search for: "))

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

    # Binary Search
    binary_result = binary_search(arr, target)
    if binary_result != -1:
        print(f"Binary Search: Target found at index {binary_result}.")
    else:
        print("Binary Search: Target not found in the array.")
'''
Time Complexity:
- Linear Search: O(n) - In the worst case, it checks each element once.
- Binary Search: O(log n) - It halves the search space with each iteration.
Space Complexity:
- Linear Search: O(1) - It uses constant extra space.
- Binary Search: O(1) - It also uses constant extra space.
In summary, binary search is more efficient than linear search for sorted arrays, while linear search can be used for unsorted arrays. However, binary search requires the array to be sorted beforehand, which may add additional time complexity if sorting is needed.
    '''
```

**Output Screenshot:**

```
PS C:\Users\sakir\OneDrive\Desktop\Ai-Assistant>  c:; cd 'c:\Users\sakir\OneDrive\Desktop\Ai-Assistant'; & 'c:\python314\python.exe' 'c:\Users\sakir\.vscode\extensions\ms-python.debugpy-2025.18.0-win32-x64\bundled\libs\debugpy\launcher' '50463' '--' 'C:\Users\sakir\OneDrive\Desktop\Ai-Assistant\lab12.py'
Enter a sorted array (space-separated): 5 10 3 6 7 8 9
Enter the target number to search for: 8
Linear Search: Target found at index 5.
Binary Search: Target found at index 5.
PS C:\Users\sakir\OneDrive\Desktop\Ai-Assistant>
```

**Explanation/Justification/Observation (100 words / 5 – 6 sentence) :**

The program implements two searching algorithms: Linear Search and Binary Search. The linear_search() function checks each element sequentially until the target is found or the list ends, making it suitable for both sorted and unsorted arrays. The binary_search() function is more efficient but requires the array to be sorted, as it repeatedly divides the search space in half. It uses two pointers, left and right, to narrow the range. The main section accepts user input and displays results from both methods. Overall, the program clearly demonstrates the difference in approach and efficiency between the two searching techniques.

---

## Task-03

**Final Optimal Prompt:**

write a python code for to develop a calculator function and apply Test Driven Development (TDD) approach to it and ensure that the code is well documented without docstrings for each method and the class itself.

The calculator should support basic operations like addition, subtraction, multiplication, and division. The TDD approach involves writing tests before implementing the functionality. Below is an example of how to implement this:

user should enter the input and make documment for every function and the class itself without using docstrings

---

**Code Screenshot :**

```python
class Calculator:
    def add(self, a, b):
        return a + b  # Return the sum of a and b

    def subtract(self, a, b):
        return a - b  # Return the difference of a and b

    def multiply(self, a, b):
        return a * b  # Return the product of a and b

    def divide(self, a, b):
        if b == 0:
            raise ValueError("Cannot divide by zero.")  # Handle division by zero
        return a / b  # Return the quotient of a and b
# Test cases for the Calculator class
def test_calculator():
    calc = Calculator()

    # Test addition
    assert calc.add(2, 3) == 5
    assert calc.add(-1, 1) == 0

    # Test subtraction
    assert calc.subtract(5, 2) == 3
    assert calc.subtract(0, 4) == -4

    # Test multiplication
    assert calc.multiply(3, 4) == 12
    assert calc.multiply(-2, 5) == -10

    # Test division
    assert calc.divide(10, 2) == 5
    try:
        calc.divide(5, 0)
    except ValueError as e:
        assert str(e) == "Cannot divide by zero."
if __name__ == "__main__":
    test_calculator()  # Run the tests
    print("All tests passed!")  # Print a message if all tests are successful
```

**Output Screenshot:**

```
y\launcher' '53800' '--' 'C:\Users\sakir\OneDrive\Desktop\Ai-Assistant\lab1
y\launcher' '53800' '--' 'C:\Users\sakir\OneDrive\Desktop\Ai-Assistant\lab1
2.py'
2.py'
All tests passed!
PS C:\Users\sakir\OneDrive\Desktop\Ai-Assistant> 
```

**Explanation/Justification/Observation (100 words / 5 – 6 sentence) :**

The program defines a Calculator class that performs basic arithmetic operations such as addition, subtraction, multiplication, and division. Each method returns the result of the respective operation, ensuring simplicity and clarity. The divide() method includes proper error handling by raising a ValueError when attempting to divide by zero, which improves program reliability. A separate test_calculator() function is used to verify correctness through assert statements, demonstrating good testing practice. The program runs these tests in the main block and confirms success if all pass. Overall, it shows clean class design, modularity, and effective unit testing.

---

## Task-04

**Final Optimal Prompt:**

write a program to generate a queue data structure with methods like enqueue, dequeue, front and is_empty

it handle queue overflow and underflow conditions and give professianal documentation for each method and the class itself using docstrings

---

**Code Screenshot :**

```python
class Queue:
    def __init__(self, capacity):
        """Initialize the queue with a given capacity."""
        self.capacity = capacity  # Maximum number of items the queue can hold
        self.queue = []  # List to store queue items

    def enqueue(self, item):
        """Add an item to the rear of the queue.
        Raises an OverflowError if the queue is full.
        """
        if len(self.queue) >= self.capacity:
            raise OverflowError("Queue overflow: Cannot enqueue item, queue is full.")
        self.queue.append(item)  # Add item to the rear of the queue

    def dequeue(self):
        """Remove and return the front item from the queue.
        Raises an IndexError if the queue is empty.
        """
        if self.is_empty():
            raise IndexError("Queue underflow: Cannot dequeue item, queue is empty.")
        return self.queue.pop(0)  # Remove and return the front item from the queue

    def front(self):
        """Return the front item without removing it from the queue.
        Raises an IndexError if the queue is empty.
        """
        if self.is_empty():
            raise IndexError("Queue underflow: Cannot access front item, queue is empty.")
        return self.queue[0]  # Return the front item without removing it

    def is_empty(self):
        """Check if the queue is empty."""
        return len(self.queue) == 0  # Return True if the queue is empty, otherwise False
# Example usage:
if __name__ == "__main__":
    queue = Queue(3)  # Create a queue with a capacity of 3
    queue.enqueue(1)
    queue.enqueue(2)
    queue.enqueue(3)
    print(queue.front())  # Output: 1
    print(queue.dequeue())  # Output: 1
    print(queue.is_empty())  # Output: False
    queue.dequeue()
    queue.dequeue()
    print(queue.is_empty())  # Output: True
    # Testing queue overflow
    try:
        queue.enqueue(4)
        queue.enqueue(5)
        queue.enqueue(6)  # This will raise an OverflowError
    except OverflowError as e:
        print(e)  # Output: Queue overflow: Cannot enqueue item, queue is full.
    # Testing queue underflow
    try:
        queue.dequeue()  # This will raise an IndexError since the queue is empty
    except IndexError as e:
        print(e)  # Output: Queue underflow: Cannot dequeue item, queue is empty.
    try:
        queue.front()  # This will raise an IndexError since the queue is empty
    except IndexError as e:
        print(e)  # Output: Queue underflow: Cannot access front item, queue is empty.
```

**Output Screenshot:**

```
2.py'
1
1
False
True
PS C:\Users\sakir\OneDrive\Desktop\Ai-Assistant>
```

**Explanation/Justification/Observation (100 words / 5 – 6 sentence) :**

The program implements a Queue data structure using Object-Oriented Programming in Python. It follows the FIFO (First In, First Out) principle, where elements are added at the rear and removed from the front. The enqueue() method adds elements while checking for overflow, raising an OverflowError if the queue exceeds its capacity. The dequeue() and front() methods handle underflow by raising an IndexError when the queue is empty. The is_empty() method checks whether the queue contains elements. The example usage demonstrates normal operations and exception handling. Overall, the program clearly illustrates queue behavior and proper error management.

---

## Tack-05

**Final Optimal Prompt:**

write a python program to generate a bubble sort algorithm and selection sort algorithm and compare between them in terms of time complexity and space complexity and also provide an example for each one of them

and ensure that the code is well documented without docstrings for each method and the class itself.

and inclue comments in the code to explain the logic of each step in the algorithms

---

**Code Screenshot :**

```python
def bubble_sort(arr):
    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]  # Swap the elements
    return arr  # Return the sorted array
def selection_sort(arr):
    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_idx = i  # Assume the minimum is the first element of the unsorted array
        for j in range(i + 1, n):
            if arr[j] < arr[min_idx]:  # Update min_idx if the current element is smaller
                min_idx = j
        # Swap the found minimum element with the first element of the unsorted array
        arr[i], arr[min_idx] = arr[min_idx], arr[i]  # Swap the elements
    return arr  # Return the sorted array
# Example usage:
if __name__ == "__main__":
    arr1 = [64, 34, 25, 12, 22, 11, 90]
    arr2 = arr1.copy()  # Create a copy of the original array for selection sort

    print("Original array:", arr1)

    sorted_arr1 = bubble_sort(arr1)
    print("Sorted array using Bubble Sort:", sorted_arr1)

    sorted_arr2 = selection_sort(arr2)
    print("Sorted array using Selection Sort:", sorted_arr2)
```

**Output Screenshot:**

```
Original array: [64, 34, 25, 12, 22, 11, 90]
Sorted array using Bubble Sort: [11, 12, 22, 25, 34, 64, 90]
Sorted array using Selection Sort: [11, 12, 22, 25, 34, 64, 90]
PS C:\Users\sakir\OneDrive\Desktop\Ai-Assistant>
```

**Explanation/Justification/Observation (100 words / 5 – 6 sentence) :**

The program implements two basic sorting algorithms: Bubble Sort and Selection Sort. The bubble_sort() function repeatedly compares adjacent elements and swaps them if they are in the wrong order, gradually moving larger elements to the end of the array. The selection_sort() function selects the minimum element from the unsorted portion and swaps it with the first unsorted element in each iteration. Both algorithms sort the array in ascending order and operate with a time complexity of $O(n^2)$. The example usage demonstrates both methods clearly. Overall, the program effectively compares two simple sorting techniques and their working principles.