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	**: K. Shailaja**
Enrollment No.	**: 2303A52155**
Batch No.	**: 34**

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

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.

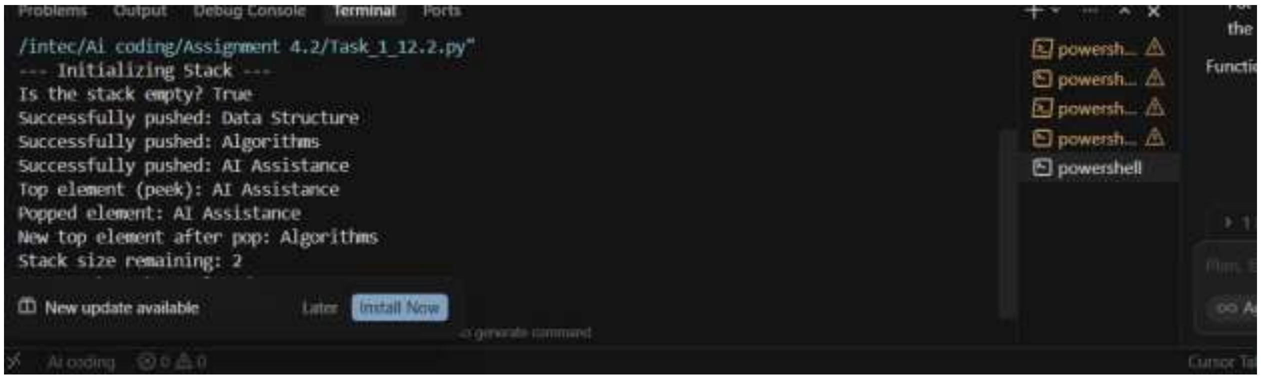
➢ Properly documented methods with docstrings.

Prompt:

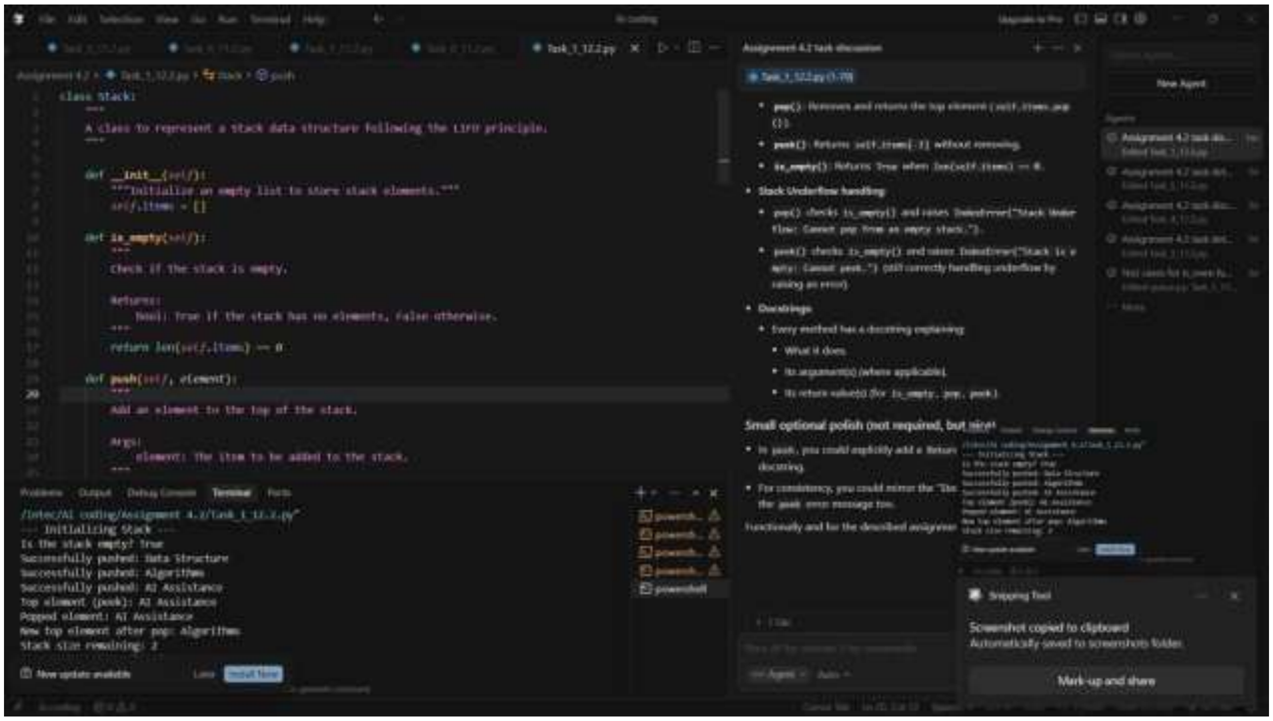
"Act as a Python developer. Create a Stackclass that follows the LIFO (Last-In, First-Out) principle. The class must include the methods push(element), pop(), peek(), and is_empty(). Please ensure you handle 'Stack Underflow' by raising an error if popor peekis called on an empty stack. Include clear docstrings for every method explaining what it does, its arguments, and its return values."

```
class Stack:
    """
    A class to represent a Stack data structure following the LIFO
principle.
    """

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

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

        Returns:
            bool: True if the stack has no elements, False otherwise.
        """
        return len(self.items) == 0

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

        Args:
            element: The item to be added to the stack.
        """
        self.items.append(element)
        print(f"Successfully pushed: {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.items.pop()

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

```python
        Returns:
            The top element of the stack.

        Raises:
            IndexError: If the stack is empty.
        """
        if self.is_empty():
            raise IndexError("Stack is empty: Cannot peek.")
        return self.items[-1]

# --- TASK OUTPUT GENERATION ---
if __name__ == "__main__":
    print("--- Initializing Stack ---")
    my_stack = Stack()

    # Testing is_empty
    print(f"Is the stack empty? {my_stack.is_empty()}")

    # Testing push
    my_stack.push("Data Structure")
    my_stack.push("Algorithms")
    my_stack.push("AI Assistance")

    # Testing peek
    print(f"Top element (peek): {my_stack.peek()}")

    # Testing pop
    print(f"Popped element: {my_stack.pop()}")
    print(f"New top element after pop: {my_stack.peek()}")

    # Final check
    print(f"Stack size remaining: {len(my_stack.items)}")
    print("--- Stack Task Completed ---")
```



```
/intec/Ai coding/Assignment 4.2/Task_1_12.2.py"
--- Initializing Stack ---
Is the stack empty? True
Successfully pushed: Data Structure
Successfully pushed: Algorithms
Successfully pushed: AI Assistance
Top element (peek): AI Assistance
Popped element: AI Assistance
New top element after pop: Algorithms
Stack size remaining: 2
```



```python
class Stack:
    """
    A class to represent a stack data structure following the LIFO principle.
    """

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

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

        Returns:
            bool: True if the stack has no elements, False otherwise.
        """
        return len(self.items) == 0

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

        Args:
            element: The item to be added to the stack.
        """
```

Assignment 4.2 task discussion

- **pop()**: Removes and returns the top element (self.items.pop()).
- **peek()**: Returns self.items[-1] without removing.
- **is_empty()**: Returns True when len(self.items) == 0.
- Stack Underflow handling
  - pop() checks is_empty() and raises IndexError("Stack Underflow: Cannot pop from an empty stack.").
  - peek() checks is_empty() and raises IndexError("Stack is empty: Cannot peek.") (still correctly handling underflow by raising an error).
- Docstrings
  - Every method has a docstring explaining:
    - What it does.
    - Its argument(s) (where applicable).
    - Its return value(s) (for is_empty, pop, peek).

Small optional polish (not required, but nice)

- In peek, you could explicitly add a Return docstring.
- For consistency, you could mirror the "Sta the peek error message too.

Functionally and for the described assignmen

```
/intec/Ai coding/Assignment 4.2/Task_1_12.2.py"
--- Initializing Stack ---
Is the stack empty? True
Successfully pushed: Data Structure
Successfully pushed: Algorithms
Successfully pushed: AI Assistance
Top element (peek): AI Assistance
Popped element: AI Assistance
New top element after pop: Algorithms
Stack size remaining: 2
```

Snipping Tool
Screenshot copied to clipboard
Automatically saved to screenshots folder

Task:2

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)

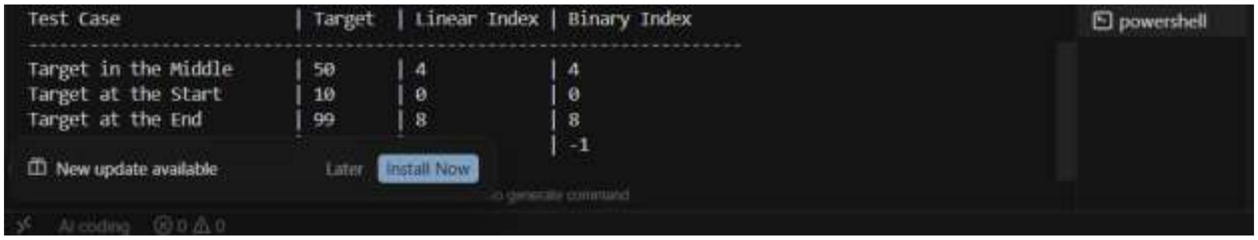
➢ Include docstrings explaining:
➢ Working principle
➢ Test both algorithms using different input sizes.

Expected Output:

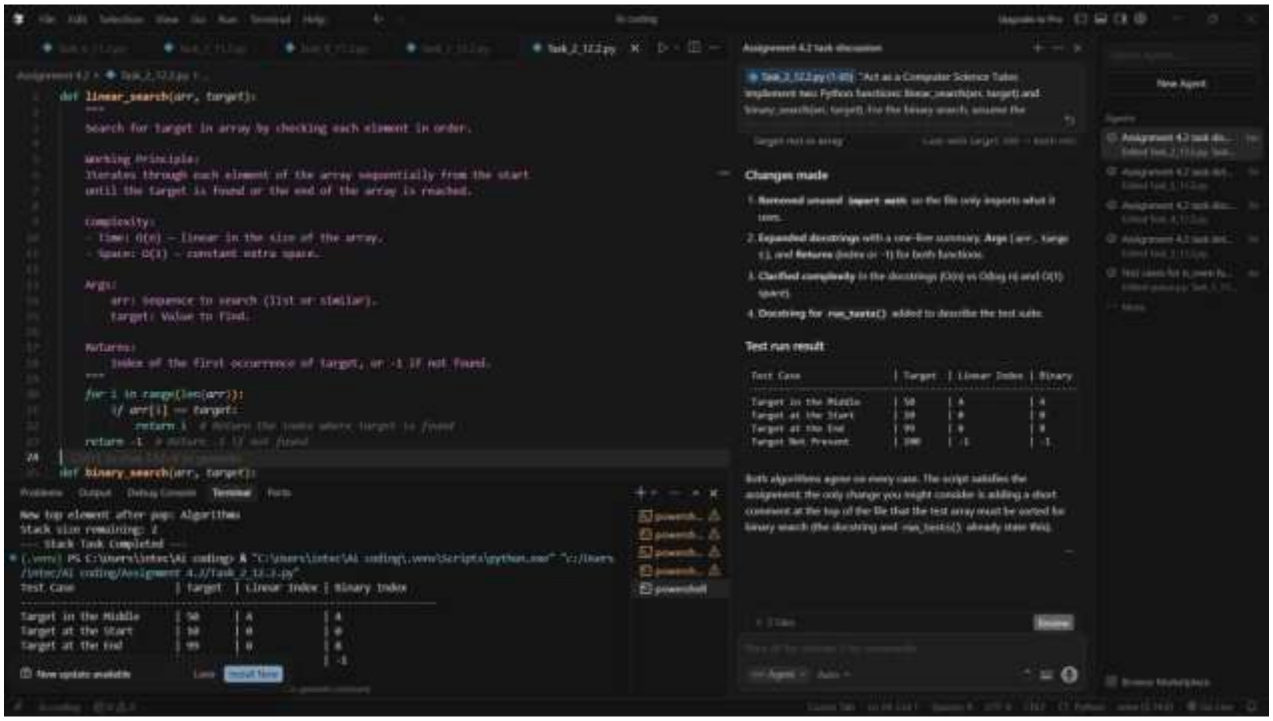
- Python implementations of both search algorithms.
- AI-generated comments and complexity analysis.
- Test results showing correctness and comparison.

Prompt:"Act as a Computer Science Tutor. Implement two Python functions: linear_search(arr, target) and binary_search(arr, target). For the binary search, assume the input array is sorted. Include detailed docstrings for both explaining their working principles and time complexity (\$O(n)\$ vs \$O(\log n)\$). Finally, create a test suite that compares both algorithms using multiple cases: a target at the start, at the end, in the middle, and a target that does not exist in the array."

```python
def linear_search(arr, target):
    """
    Search for target in array by checking each element in order.

    Working Principle:
    Iterates through each element of the array sequentially from the start
    until the target is found or the end of the array is reached.

    Complexity:
    - Time: O(n) — linear in the size of the array.
    - Space: O(1) — constant extra space.

    Args:
        arr: Sequence to search (list or similar).
        target: Value to find.

    Returns:
        Index of the first occurrence of target, or -1 if not found.
    """
    for i in range(len(arr)):
        if arr[i] == target:
            return i  # Return the index where target is found
    return -1  # Return -1 if not found

def binary_search(arr, target):
    """
    Search for target in a sorted array by repeatedly halving the search
range.

    Working Principle:
    Divide-and-conquer: compare target to the middle element; if equal,
return
    that index. If target is less than the middle, search the left half;
    otherwise search the right half. Assumes the input array is sorted in
    ascending order.

    Complexity:
    - Time: O(log n) — logarithmic in the size of the array.
    - Space: O(1) — iterative implementation uses constant extra space.

    Args:
        arr: Sorted sequence to search (must be sorted in ascending
order).
        target: Value to find.

    Returns:
        Index of target if present, or -1 if not found.
    """
    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

# --- TEST SUITE AND COMPARISON ---
def run_tests():
    """Compare linear_search and binary_search on the same cases."""
    # Sorted array (required for binary search)
    test_array = [10, 22, 35, 47, 50, 63, 75, 88, 99]

    test_cases = [
        {"name": "Target in the Middle", "target": 50},
        {"name": "Target at the Start ", "target": 10},
        {"name": "Target at the End   ", "target": 99},
        {"name": "Target Not Present ", "target": 100},
    ]

    print(f"{'Test Case':<25} | {'Target':<7} | {'Linear Index':<12} |
{'Binary Index'}")
    print("-" * 70)

    for case in test_cases:
        target = case["target"]
        l_result = linear_search(test_array, target)
        b_result = binary_search(test_array, target)

        print(f"{case['name']:<25} | {target:<7} | {l_result:<12} |
{b_result}")

if __name__ == "__main__":
    run_tests()
```



```
Test Case                | Target  | Linear Index | Binary Index
----------------------------------------------------------------
Target in the Middle     | 50      | 4            | 4
Target at the Start      | 10      | 0            | 0
Target at the End        | 99      | 8            | 8
                                                  | -1
```



```python
def linear_search(arr, target):
    """
    Search for target in array by checking each element in order.

    Working Principle:
    Iterates through each element of the array sequentially from the start
    until the target is found or the end of the array is reached.

    Complexity:
    - Time: O(n) — linear in the size of the array.
    - Space: O(1) — constant extra space.

    Args:
        arr: Sequence to search (list or similar).
        target: Value to find.

    Returns:
        Index of the first occurrence of target, or -1 if not found.
    """
    for i in range(len(arr)):
        if arr[i] == target:
            return i  # Return the index where target is found
    return -1  # Return -1 if not found

def binary_search(arr, target):
```

```
New top element after pop: Algorithms
Stack size remaining: 2
--- Stack Task Completed ---
(.venv) PS C:\Users\intec\AI coding> & "C:\Users\intec\AI coding\.venv\Scripts\python.exe" "c:/Users/intec/AI coding/Assignment 4.2/Task_2_12.2.py"
Test Case                | Target  | Linear Index | Binary Index
----------------------------------------------------------------
Target in the Middle     | 50      | 4            | 4
Target at the Start      | 10      | 0            | 0
Target at the End        | 99      | 8            | 8
                                                  | -1
```

Assignment 4.2 task discussion

Task_2_12.2.py (1-80) "Act as a Computer Science Tutor. Implement two Python functions: linear_search(arr, target) and binary_search(arr, target). For the binary search, assume the

Changes made

1. **Removed unused `import math`** so the file only imports what it uses.
2. **Expanded docstrings** with a one-line summary, **Args** (`arr`, `target`), and **Returns** (index or -1) for both functions.
3. **Clarified complexity** in the docstrings (O(n) vs O(log n) and O(1) space).
4. **Docstring for `run_tests()`** added to describe the test suite.

Test run result

```
Test Case                | Target  | Linear Index | Binary
---------------------------------------------------------
Target in the Middle     | 50      | 4            | 4
Target at the Start      | 10      | 0            | 0
Target at the End        | 99      | 8            | 8
Target Not Present       | 100     | -1           | -1
```

Both algorithms agree on every case. The script satisfies the assignment; the only change you might consider is adding a short comment at the top of the file that the test array must be sorted for binary search (the docstring and `run_tests()` already state this).

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.

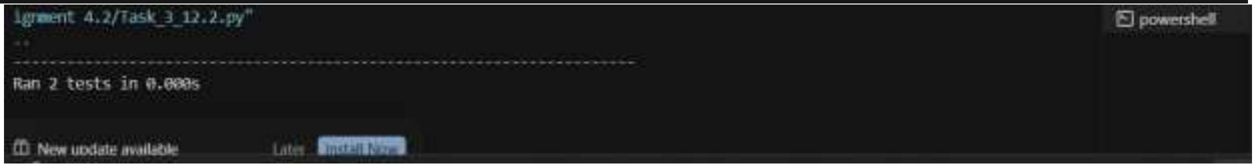
➢ Test cases executed before implementation.
➢ Final implementation passing all test cases.

Prompt:"Act as a QA Engineer. Write a Python unit test script using the unittestlibrary for a Calculatorclass. The tests should cover add(a, b)and subtract(a, b)methods.
Include test cases for positive numbers, negative numbers, and zeros. Do not provide the implementation yet." The implementation:

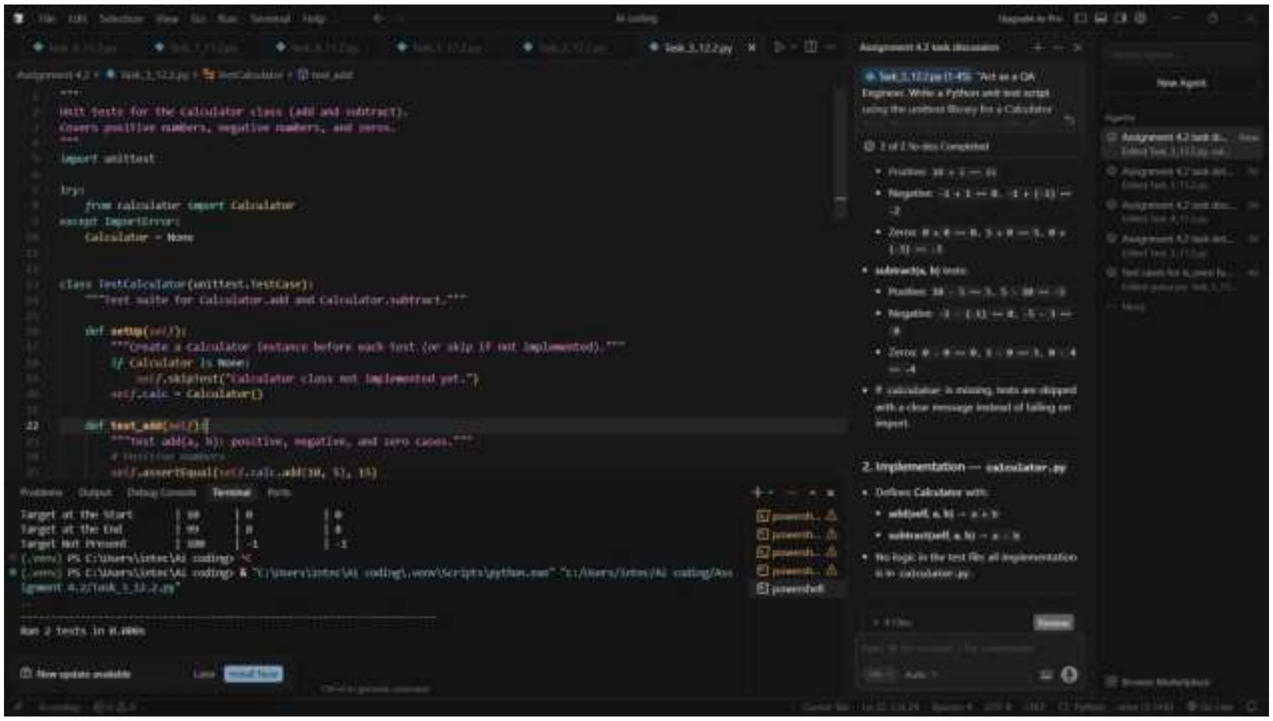
"Now, acting as a Developer, write the Calculatorclass that satisfies the unit tests provided in the previous step. Ensure the methods addand subtractare implemented correctly to pass all test cases."

```
"""
Unit tests for the Calculator class (add and subtract).
Covers positive numbers, negative numbers, and zeros.
"""
import unittest

try:
    from calculator import Calculator
except ImportError:
    Calculator = None


class TestCalculator(unittest.TestCase):
    """Test suite for Calculator.add and Calculator.subtract."""

    def setUp(self):
        """Create a Calculator instance before each test (or skip if not
implemented)."""
        if Calculator is None:
            self.skipTest("Calculator class not implemented yet.")
        self.calc = Calculator()

    def test_add(self):
```

```python
        """Test add(a, b): positive, negative, and zero cases."""
        # Positive numbers
        self.assertEqual(self.calc.add(10, 5), 15)
        # Negative numbers
        self.assertEqual(self.calc.add(-1, 1), 0)
        self.assertEqual(self.calc.add(-1, -1), -2)
        # Zeros
        self.assertEqual(self.calc.add(0, 0), 0)
        self.assertEqual(self.calc.add(5, 0), 5)
        self.assertEqual(self.calc.add(0, -3), -3)

    def test_subtract(self):
        """Test subtract(a, b): positive, negative, and zero cases."""
        # Positive numbers
        self.assertEqual(self.calc.subtract(10, 5), 5)
        self.assertEqual(self.calc.subtract(5, 10), -5)
        # Negative numbers
        self.assertEqual(self.calc.subtract(-1, -1), 0)
        self.assertEqual(self.calc.subtract(-5, 3), -8)
        # Zeros
        self.assertEqual(self.calc.subtract(0, 0), 0)
        self.assertEqual(self.calc.subtract(5, 0), 5)
        self.assertEqual(self.calc.subtract(0, 4), -4)


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



```
ignment 4.2/Task_3_12.2.py"
..
----------------------------------------------------------------------
Ran 2 tests in 0.000s
```


```python
"""
Unit tests for the Calculator class (add and subtract).
Covers positive numbers, negative numbers, and zeros.
"""
import unittest

try:
    from calculator import Calculator
except ImportError:
    Calculator = None


class TestCalculator(unittest.TestCase):
    """Test suite for Calculator.add and Calculator.subtract."""

    def setUp(self):
        """Create a Calculator instance before each test (or skip if not implemented)."""
        if Calculator is None:
            self.skipTest("Calculator class not implemented yet.")
        self.calc = Calculator()

    def test_add(self):
        """Test add(a, b): positive, negative, and zero cases."""
        # Positive numbers
        self.assertEqual(self.calc.add(10, 5), 15)
```

```
Target at the Start       | 10     | 0          | 0
Target at the End         | 99     | 0          | 0
Target Not Present        | 100    | -1         | -1
(.venv) PS C:\Users\intec\AI coding> ^C
(.venv) PS C:\Users\intec\AI coding> & "C:\Users\intec\AI coding\.venv\Scripts\python.exe" "c:/Users/intec/AI coding/Assignment 4.2/Task_3_12.2.py"
..
----------------------------------------------------------------------
Ran 2 tests in 0.000s
```

Task_3_12.2.py (1-45) "Act as a QA Engineer. Write a Python unit test script using the unittest library for a Calculator

2 of 2 To-dos Completed

- Positive: 10 + 5 == 15
- Negative: -1 + 1 == 0, -1 + (-1) == -2
- Zeros: 0 + 0 == 0, 5 + 0 == 5, 0 + (-3) == -3
- subtract(a, b) tests:
  - Positive: 10 - 5 == 5, 5 - 10 == -5
  - Negative: -3 - (-3) == 0, -5 - 3 == -8
  - Zeros: 0 - 0 == 0, 5 - 0 == 5, 0 - 4 == -4
- If calculator is missing, tests are skipped with a clear message instead of failing on import.

2. Implementation — calculator.py

- Defines Calculator with:
  - add(self, a, b) → a + b
  - subtract(self, a, b) → a - b
- No logic in the test file; all implementation is in calculator.py.

File Edit Selection View Go Run Terminal Help
AI coding
Task_3_12.2.py
Assignment 4.2 task discussion
Problems Output Debug Console Terminal Ports
New Agent
New update available
Later
Install Now

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.

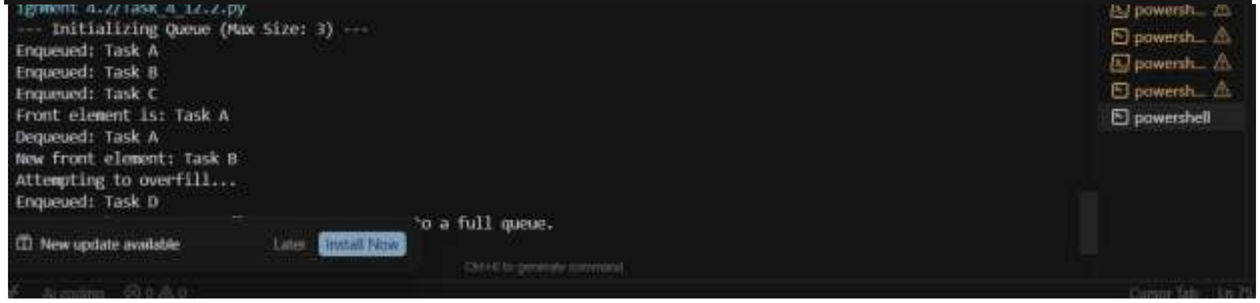
Expected Output:

➢ A fully functional Queue implementation in Python.

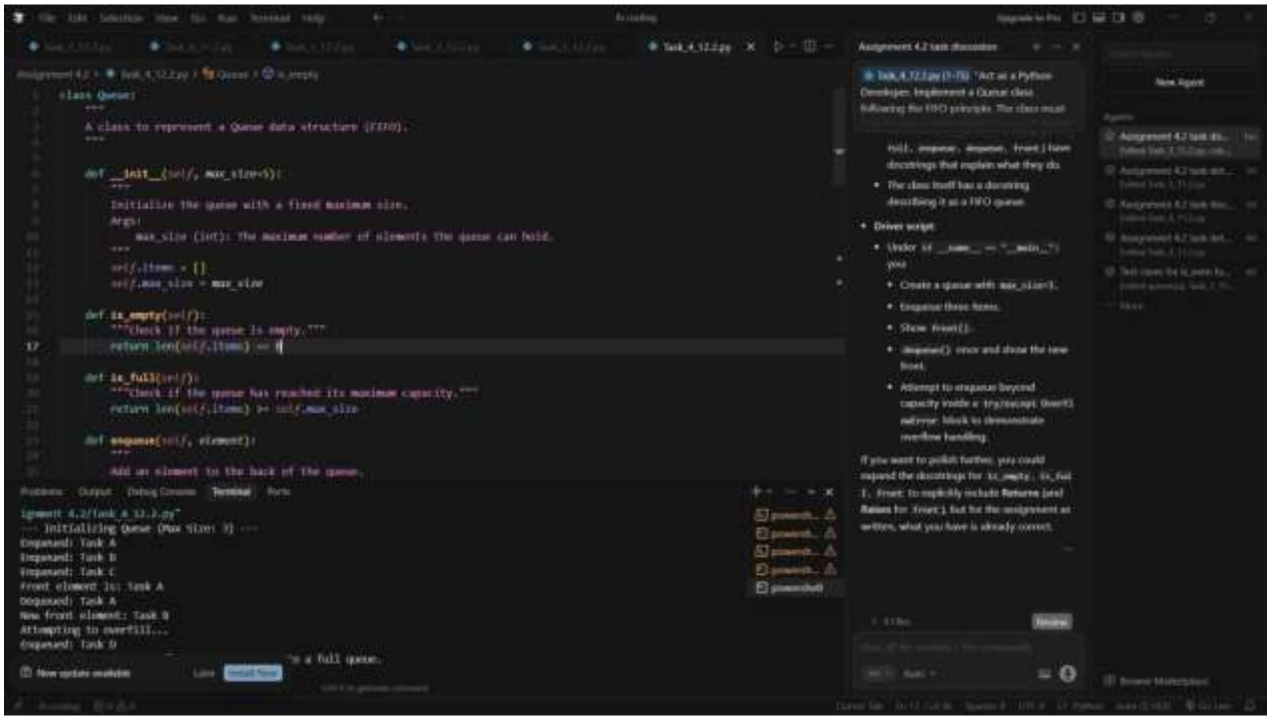
➢ Proper error handling and documentation.

Prompt:"Act as a Python Developer. Implement a Queueclass following the FIFO principle. The class must include enqueue(element), dequeue(), front(), and is_empty() methods. Include a max_size parameter in the constructor to handle 'Queue Overflow' and ensure 'Queue Underflow' is handled when dequeuing from an empty queue. Provide clear docstrings for all methods and a driver script to demonstrate the output."

```
class Queue:
    """
    A class to represent a Queue data structure (FIFO).
    """

    def __init__(self, max_size=5):
        """
        Initialize the queue with a fixed maximum size.
        Args:
            max_size (int): The maximum number of elements the queue can
hold.
        """
        self.items = []
        self.max_size = max_size

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

    def is_full(self):
        """Check if the queue has reached its maximum capacity."""
        return len(self.items) >= self.max_size

    def enqueue(self, element):
        """
        Add an element to the back of the queue.
        Raises:
            OverflowError: If the queue is already full.
        """
        if self.is_full():
            raise OverflowError("Queue Overflow: Cannot enqueue to a full
queue.")
        self.items.append(element)
        print(f"Enqueued: {element}")

    def dequeue(self):
        """
        Remove and return the front element of the queue.
        Returns:
            The front element.
        Raises:
            IndexError: If the queue is empty (Underflow).
        """
        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 without removing it."""
        if self.is_empty():
            raise IndexError("Queue is empty: No front element.")
        return self.items[0]

# --- DRIVER CODE FOR OUTPUT ---
if __name__ == "__main__":
    print("--- Initializing Queue (Max Size: 3) ---")
    my_queue = Queue(max_size=3)

    # 1. Test Enqueue
    my_queue.enqueue("Task A")
    my_queue.enqueue("Task B")
    my_queue.enqueue("Task C")

    # 2. Test Front
    print(f"Front element is: {my_queue.front()}")

    # 3. Test Dequeue (FIFO Logic)
    print(f"Dequeued: {my_queue.dequeue()}")
    print(f"New front element: {my_queue.front()}")

    # 4. Test Overflow Catching
    try:
        print("Attempting to overfill...")
        my_queue.enqueue("Task D")
        my_queue.enqueue("Task E")  # This should trigger the Overflow
    except OverflowError as e:
        print(f"Error Caught: {e}")
```



```
ignment 4.2/Task_4_12.2.py
--- Initializing Queue (Max Size: 3) ---
Enqueued: Task A
Enqueued: Task B
Enqueued: Task C
Front element is: Task A
Dequeued: Task A
New front element: Task B
Attempting to overfill...
Enqueued: Task D
                                    'o a full queue.
```



```
class Queue:
    """
    A class to represent a Queue data structure (FIFO).
    """

    def __init__(self, max_size=5):
        """
        Initialize the queue with a fixed maximum size.
        Args:
            max_size (int): The maximum number of elements the queue can hold.
        """
        self.items = []
        self.max_size = max_size

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

    def is_full(self):
        """Check if the queue has reached its maximum capacity."""
        return len(self.items) >= self.max_size

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

```
ignment 4.2/Task_4_12.2.py"
--- Initializing Queue (Max Size: 3) ---
Enqueued: Task A
Enqueued: Task B
Enqueued: Task C
Front element is: Task A
Dequeued: Task A
New front element: Task B
Attempting to overfill...
Enqueued: Task D
                                    'o a full queue.
```

Assignment 4.2 task discussion

Task_4_12.2.py (1-75) "Act as a Python Developer. Implement a Queue class following the FIFO principle. The class must

full(), enqueue, dequeue, front()) have docstrings that explain what they do.

- The class itself has a docstring describing it as a FIFO queue.

- Driver script
  - Under if __name__ == "__main__": you:
    - Create a queue with max_size=3.
    - Enqueue three items.
    - Show front().
    - dequeue() once and show the new front.
    - Attempt to enqueue beyond capacity inside a try/except OverflowError block to demonstrate overflow handling.

If you want to polish further, you could expand the docstrings for is_empty, is_full(), front() to explicitly include Returns (and Raises for front()), but for the assignment as written, what you have is already correct.

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:

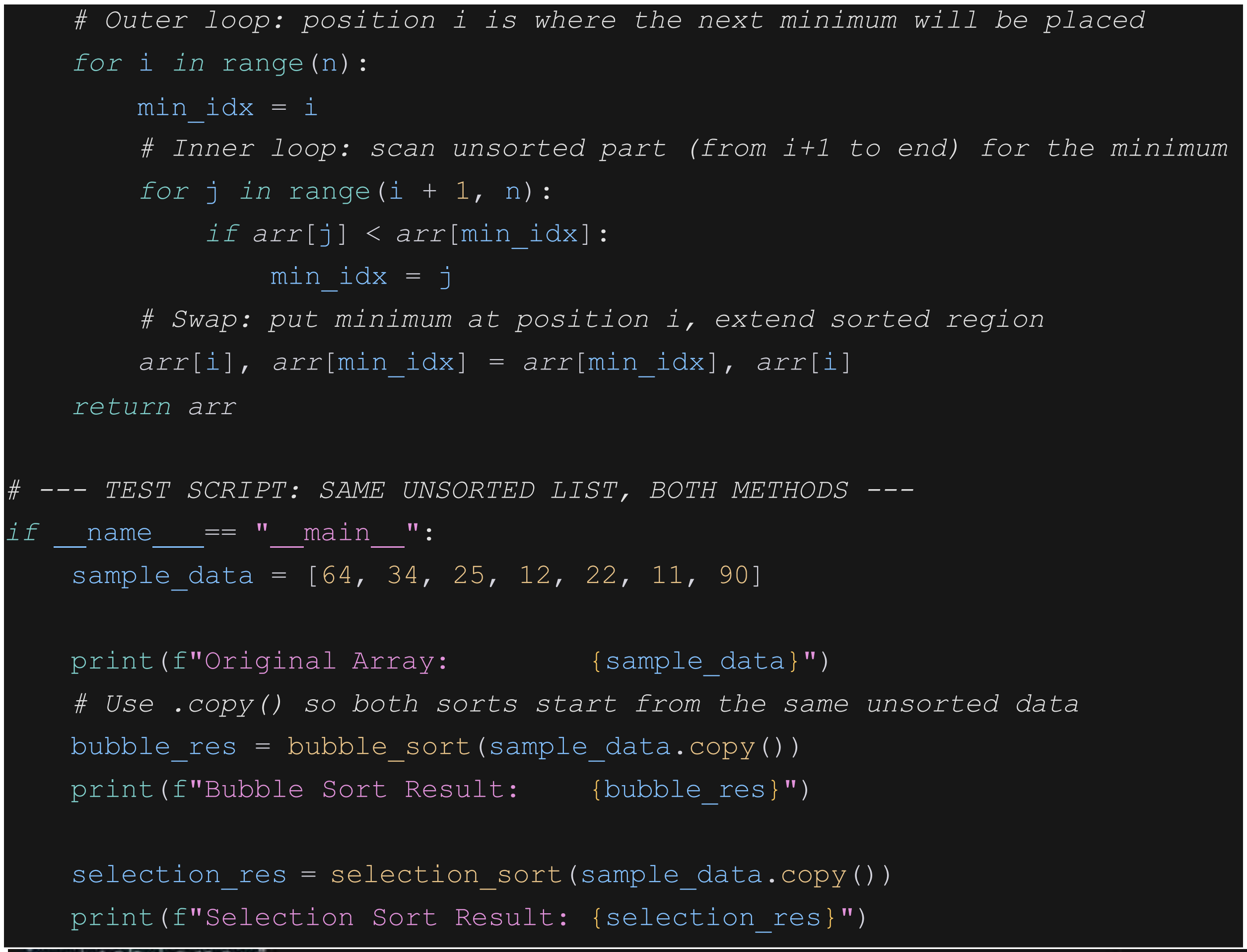
• Correct Python implementations of both sorting algorithms.

• Complexity analysis in docstrings.

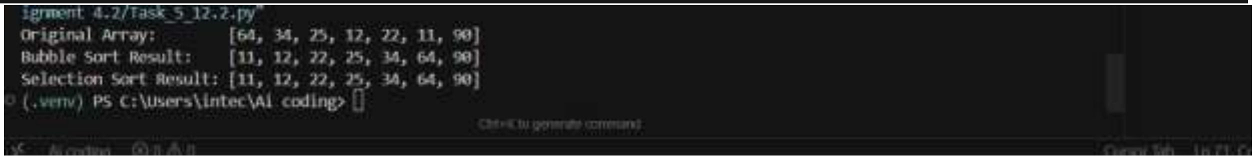
Prompt:

> *Act as an Algorithms Instructor. Implement bubble_sort(arr) and selection_sort(arr) in Python. For each function, include a detailed docstring explaining the time complexity ($O(n^2)$) and space complexity ($O(1)$). Inside the code, use comments to explain how the loops and swaps work. Finally, provide a test script that sorts the same unsorted list using both methods and prints the results."*

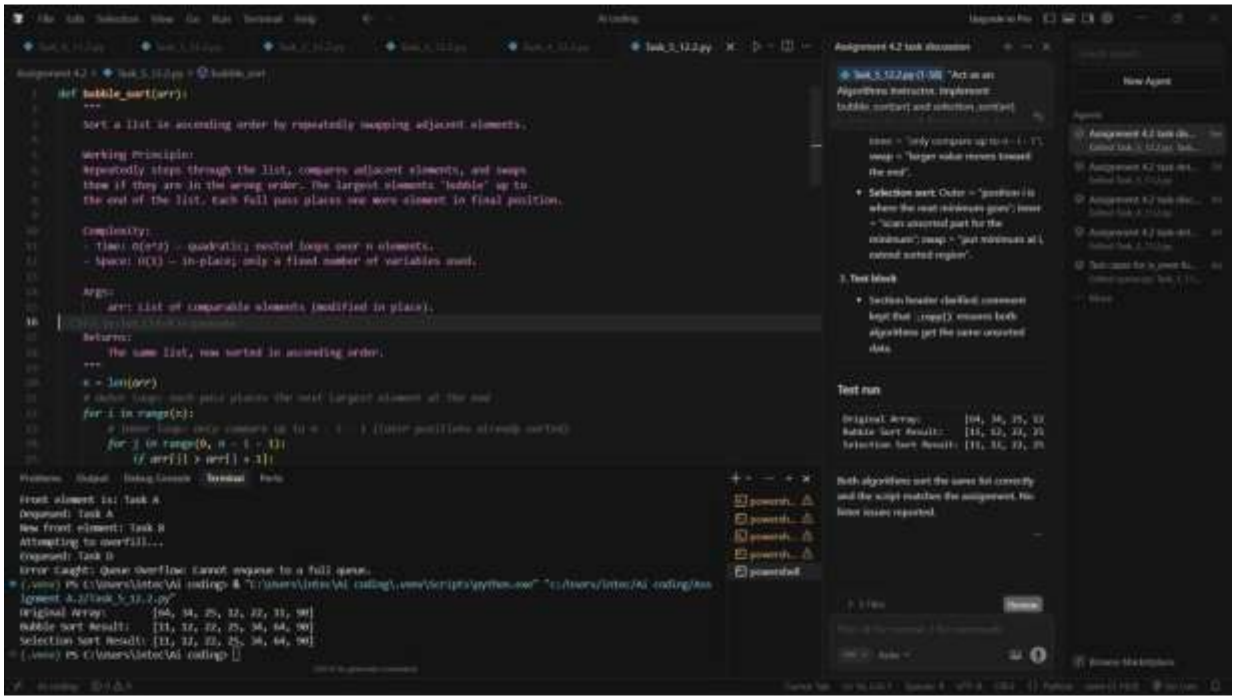
```python
def bubble_sort(arr):
    """
    Sort a list in ascending order by repeatedly swapping adjacent
elements.

    Working Principle:
    Repeatedly steps through the list, compares adjacent elements, and
swaps
    them if they are in the wrong order. The largest elements 'bubble' up
to
    the end of the list. Each full pass places one more element in final
position.

    Complexity:
    - Time: O(n^2) — quadratic; nested loops over n elements.
    - Space: O(1) — in-place; only a fixed number of variables used.
    Args:
        arr: List of comparable elements (modified in place).

    Returns:
        The same list, now sorted in ascending order.
    """
    n = len(arr)
    # Outer loop: each pass places the next largest element at the end
    for i in range(n):
        # Inner loop: only compare up to n - i - 1 (later positions
already sorted)
        for j in range(0, n - i - 1):
            if arr[j] > arr[j + 1]:
                # Swap so the larger value moves toward the end
                arr[j], arr[j + 1] = arr[j + 1], arr[j]
    return arr

def selection_sort(arr):
    """
    Sort a list in ascending order by repeatedly selecting the minimum.

    Working Principle:
    Keeps a sorted region at the front and an unsorted region after it.
Each
    pass finds the smallest element in the unsorted region and swaps it
with
    the first unsorted element, extending the sorted region by one.

    Complexity:
    - Time: O(n^2) — quadratic; nested loops over n elements.
    - Space: O(1) — in-place; only a fixed number of variables used.

    Args:
        arr: List of comparable elements (modified in place).

    Returns:
        The same list, now sorted in ascending order.
    """
    n = len(arr)
```



```python
    # Outer loop: position i is where the next minimum will be placed
    for i in range(n):
        min_idx = i
        # Inner loop: scan unsorted part (from i+1 to end) for the minimum
        for j in range(i + 1, n):
            if arr[j] < arr[min_idx]:
                min_idx = j
        # Swap: put minimum at position i, extend sorted region
        arr[i], arr[min_idx] = arr[min_idx], arr[i]
    return arr

# --- TEST SCRIPT: SAME UNSORTED LIST, BOTH METHODS ---
if __name__ == "__main__":
    sample_data = [64, 34, 25, 12, 22, 11, 90]

    print(f"Original Array:        {sample_data}")
    # Use .copy() so both sorts start from the same unsorted data
    bubble_res = bubble_sort(sample_data.copy())
    print(f"Bubble Sort Result:    {bubble_res}")

    selection_res = selection_sort(sample_data.copy())
    print(f"Selection Sort Result: {selection_res}")
```



```
ignment 4.2/Task_5_12.2.py"
Original Array:        [64, 34, 25, 12, 22, 11, 90]
Bubble Sort Result:    [11, 12, 22, 25, 34, 64, 90]
Selection Sort Result: [11, 12, 22, 25, 34, 64, 90]
(.venv) PS C:\Users\intec\AI coding> 
```



```python
def bubble_sort(arr):
    """
    Sort a list in ascending order by repeatedly swapping adjacent elements.

    Working Principle:
    Repeatedly steps through the list, compares adjacent elements, and swaps
    them if they are in the wrong order. The largest elements "bubble" up to
    the end of the list. Each full pass places one more element in final position.

    Complexity:
    - Time: O(n^2) - quadratic; nested loops over n elements.
    - Space: O(1) - in-place; only a fixed number of variables used.

    Args:
        arr: List of comparable elements (modified in place).
[illegible]
    Returns:
        The same list, now sorted in ascending order.
    """
    n = len(arr)
    # Outer loop: each pass places the next largest element at the end
    for i in range(n):
        # Inner loop: only compare up to n - i - 1 (later positions already sorted)
        for j in range(0, n - i - 1):
            if arr[j] > arr[j + 1]:
```

```
Front element is: Task A
Dequeued: Task A
New front element: Task B
Attempting to overfill...
Enqueued: Task D
Error Caught: Queue Overflow: Cannot enqueue to a full queue.
(.venv) PS C:\Users\intec\AI coding> & "C:\Users\intec\AI coding\.venv\Scripts\python.exe" "c:/Users/intec/AI coding/Assignment 4.2/Task_5_12.2.py"
Original Array:        [64, 34, 25, 12, 22, 11, 90]
Bubble Sort Result:    [11, 12, 22, 25, 34, 64, 90]
Selection Sort Result: [11, 12, 22, 25, 34, 64, 90]
(.venv) PS C:\Users\intec\AI coding> 
```

Assignment 4.2 task discussion

Task_5_12.2.py (1-58) "Act as an Algorithms Instructor. Implement bubble_sort(arr) and selection_sort(arr)

Inner = "only compare up to n - i - 1"; swap = "larger value moves toward the end".

- **Selection sort**: Outer = "position i is where the next minimum goes"; inner = "scan unsorted part for the minimum"; swap = "put minimum at i, extend sorted region".

3. **Test block**
   - Section header clarified; comment kept that .copy() ensures both algorithms get the same unsorted data.

Test run

```
Original Array:        [64, 34, 25, 12
Bubble Sort Result:    [11, 12, 22, 25
Selection Sort Result: [11, 12, 22, 25
```

Both algorithms sort the same list correctly and the script matches the assignment. No linter issues reported.