

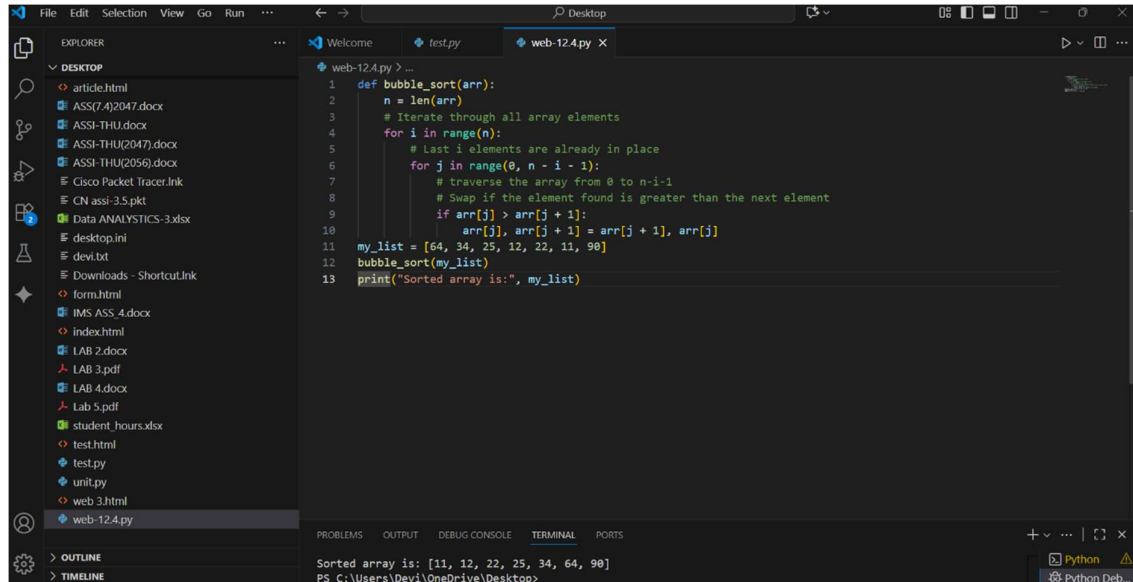
ASSIGNMENT-12.4

NAME:MOHANTH SIDDHARTHA.S

2403A52047

BATCH-03

TASK-1: Write a Python implementation of Bubble Sort

A screenshot of a Python IDE window. The left sidebar shows a file explorer with various files on the desktop. The main editor area displays a Python script named 'web-12.4.py'. The script implements a bubble sort function. The code is as follows:

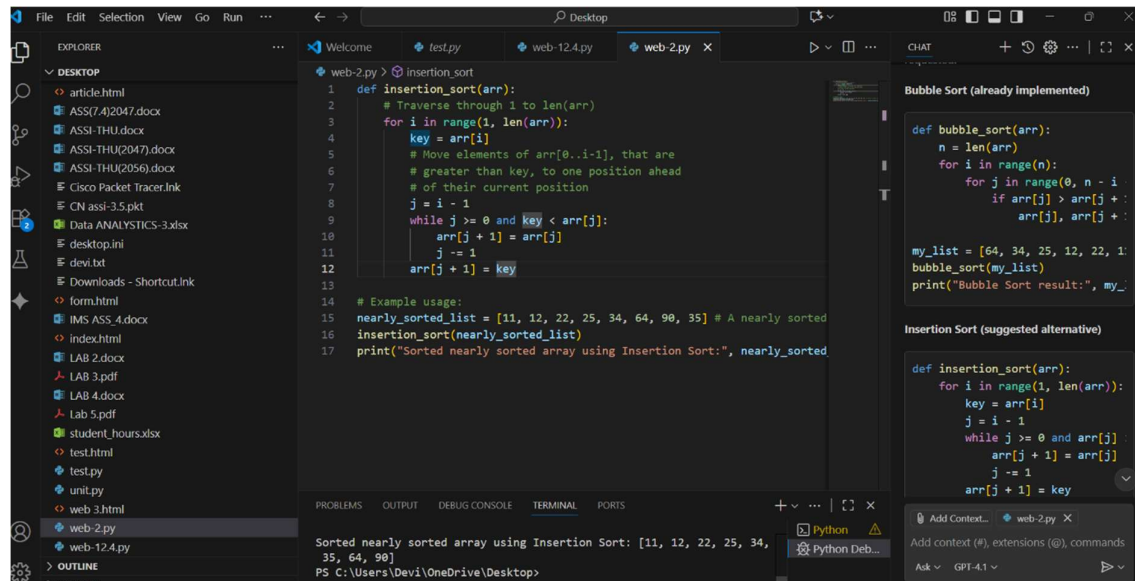
```
1 def bubble_sort(arr):
2     n = len(arr)
3     # Iterate through all array elements
4     for i in range(n):
5         # Last i elements are already in place
6         for j in range(0, n - i - 1):
7             # traverse the array from 0 to n-i-1
8             # Swap if the element found is greater than the next element
9             if arr[j] > arr[j + 1]:
10                 arr[j], arr[j + 1] = arr[j + 1], arr[j]
11 my_list = [64, 34, 25, 12, 22, 11, 90]
12 bubble_sort(my_list)
13 print("Sorted array is:", my_list)
```

The bottom of the IDE shows the output of the program: 'Sorted array is: [11, 12, 22, 25, 34, 64, 90]'. The terminal also shows the file path: 'PS C:\Users\Dev\OneDrive\Desktop>'.

EXPLANATION:

- **def bubble_sort(arr):** This line defines a function named bubble_sort that takes one argument, arr, which is the list to be sorted.
- **n = len(arr):** This gets the number of elements in the input list and stores it in the variable n.
- **for i in range(n):** This is the outer loop. It iterates n times. In each iteration, the largest unsorted element "bubbles up" to its correct position at the end of the unsorted portion of the list.
- **for j in range(0, n - i - 1):** This is the inner loop. It traverses the unsorted portion of the array (from the beginning up to the i-th element from the end, which is already sorted).

TASK-2: Provide Bubble Sort code to AI and ask it to suggest a more efficient algorithm for partially sorted arrays



The screenshot shows a VS Code editor with a file explorer on the left, a code editor in the center, and a chat window on the right. The code editor contains the following Python code:

```
1 def insertion_sort(arr):
2     # Traverse through 1 to len(arr)
3     for i in range(1, len(arr)):
4         key = arr[i]
5         # Move elements of arr[0..i-1], that are
6         # greater than key, to one position ahead
7         # of their current position
8         j = i - 1
9         while j >= 0 and key < arr[j]:
10            arr[j + 1] = arr[j]
11            j -= 1
12        arr[j + 1] = key
13
14 # Example usage:
15 nearly_sorted_list = [11, 12, 22, 25, 34, 64, 90, 35] # A nearly sorted
16 insertion_sort(nearly_sorted_list)
17 print("Sorted nearly sorted array using Insertion Sort:", nearly_sorted
```

The chat window on the right shows the following text:

Bubble Sort (already implemented)

```
def bubble_sort(arr):
    n = len(arr)
    for i in range(n):
        for j in range(0, n - i):
            if arr[j] > arr[j + 1]:
                arr[j], arr[j + 1] = arr[j + 1], arr[j]

my_list = [64, 34, 25, 12, 22, 1]
bubble_sort(my_list)
print("Bubble Sort result:", my_list)
```

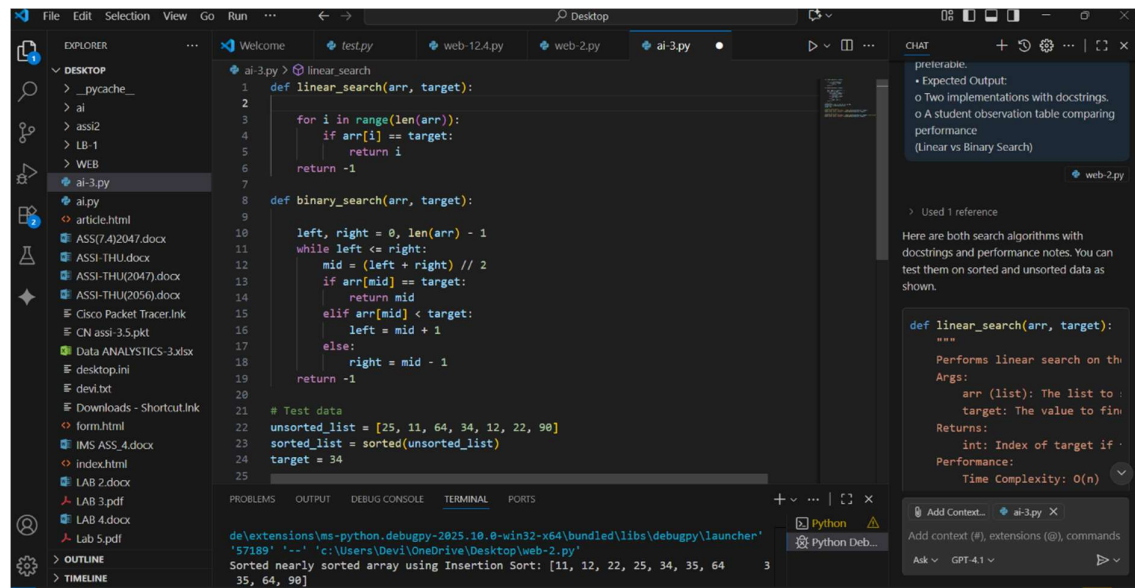
Insertion Sort (suggested alternative)

```
def insertion_sort(arr):
    for i in range(1, len(arr)):
        key = arr[i]
        j = i - 1
        while j >= 0 and arr[j] > key:
            arr[j + 1] = arr[j]
            j -= 1
        arr[j + 1] = key
```

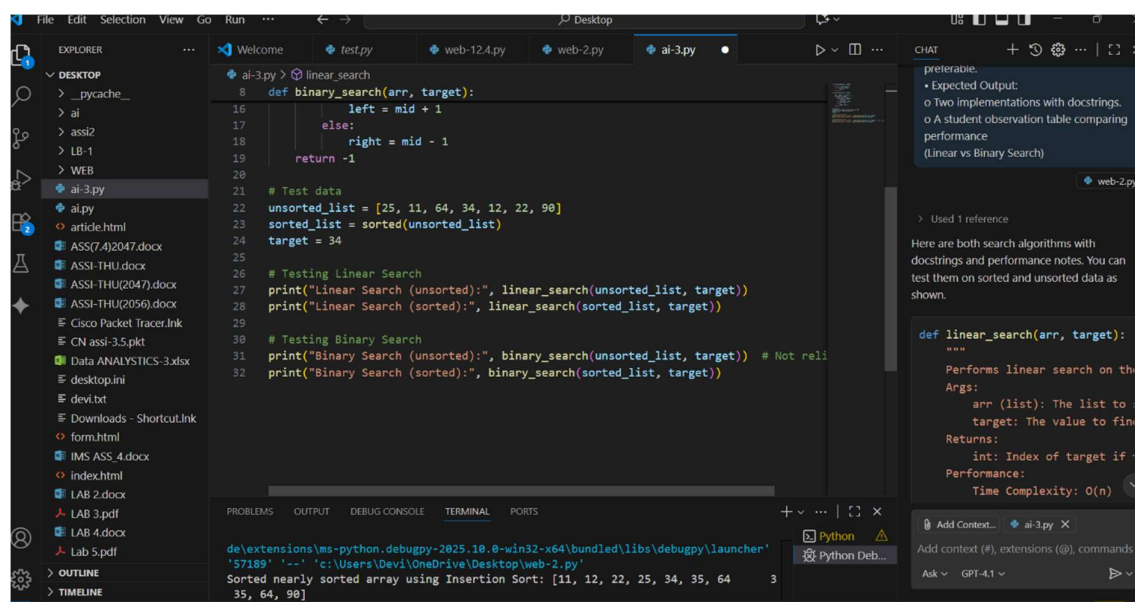
EXPLANATION:

- **def insertion_sort(arr):** This line defines a function named `insertion_sort` that takes one argument, `arr`, which is the list to be sorted.
- **for i in range(1, len(arr)):** This is the main loop. It iterates through the list starting from the second element ($i = 1$) up to the last element. It considers each element `arr[i]` as the "key" to be inserted into the already sorted portion of the array to its left.
- **key = arr[i]:** This line stores the current element being considered for insertion in the `key` variable.
- **j = i - 1:** This initializes a variable `j` to the index of the last element in the sorted portion of the array (to the left of the key).

TASK-3: Implement both Linear Search and Binary Search



```
1 def linear_search(arr, target):
2     for i in range(len(arr)):
3         if arr[i] == target:
4             return i
5     return -1
6
7 def binary_search(arr, target):
8
9     left, right = 0, len(arr) - 1
10    while left <= right:
11        mid = (left + right) // 2
12        if arr[mid] == target:
13            return mid
14        elif arr[mid] < target:
15            left = mid + 1
16        else:
17            right = mid - 1
18    return -1
19
20 # Test data
21 unsorted_list = [25, 11, 64, 34, 12, 22, 90]
22 sorted_list = sorted(unsorted_list)
23 target = 34
24
25
26 de\extensions\ms-python.debugpy-2025.10.0-win32-x64\bundle\libs\debugpy\launcher
27 '57189' '--' 'c:\Users\Devi\OneDrive\Desktop\web-2.py'
28 Sorted nearly sorted array using Insertion Sort: [11, 12, 22, 25, 34, 35, 64
29 35, 64, 90]
```



```
8 def binary_search(arr, target):
9     left, right = 0, len(arr) - 1
10    while left <= right:
11        mid = (left + right) // 2
12        if arr[mid] == target:
13            return mid
14        elif arr[mid] < target:
15            left = mid + 1
16        else:
17            right = mid - 1
18    return -1
19
20 # Test data
21 unsorted_list = [25, 11, 64, 34, 12, 22, 90]
22 sorted_list = sorted(unsorted_list)
23 target = 34
24
25 # Testing Linear Search
26 print("Linear Search (unsorted):", linear_search(unsorted_list, target))
27 print("Linear Search (sorted):", linear_search(sorted_list, target))
28
29 # Testing Binary Search
30 print("Binary Search (unsorted):", binary_search(unsorted_list, target)) # Not reli
31 print("Binary Search (sorted):", binary_search(sorted_list, target))
32
```

EXPLANATION:

- **def linear_search(arr, target):**: This line defines a function named linear_search that takes two arguments: arr (the list to search within) and target (the value to search for).
- **""" ... """**: This is a docstring, which explains what the function does, its arguments (Args), and what it returns (Returns).
- **for index, element in enumerate(arr):**: This loop iterates through each element in the input list arr. The enumerate() function provides both the index and the value of each element.

- **if element == target::** This condition checks if the current element is equal to the target value.

TASK-4: Quick Sort and Merge Sort Comparison

The screenshot shows a VS Code editor with a file explorer on the left, a code editor in the center, a chat window on the right, and a terminal at the bottom.

Code Editor: The file `ai-4.py` contains the following code:

```

28 def merge(left, right):
29     i = j = 0
30     while i < len(left) and j < len(right):
31         if left[i] <= right[j]:
32             result.append(left[i])
33             i += 1
34         else:
35             result.append(right[j])
36             j += 1
37     result.extend(left[i:])
38     result.extend(right[j:])
39     return result
40
41 # Test lists
42 import random
43 random_list = random.sample(range(1, 100), 10)
44 sorted_list = sorted(random_list)
45 reverse_sorted_list = sorted(random_list, reverse=True)
46
47 print("Quick Sort (random):", quick_sort(random_list))
48 print("Quick Sort (sorted):", quick_sort(sorted_list))
49 print("Quick Sort (reverse):", quick_sort(reverse_sorted_list))
50
51 print("Merge Sort (random):", merge_sort(random_list))
52 print("Merge Sort (sorted):", merge_sort(sorted_list))
53 print("Merge Sort (reverse):", merge_sort(reverse_sorted_list))

```

Chat Window: The chat window on the right shows a reference for the `quick_sort` function, explaining its recursive nature, arguments, and performance characteristics.

Terminal: The terminal at the bottom shows the output of the program:

```

PS C:\Users\Devi\OneDrive\Desktop> python ai-4.py
Quick Sort (random): [18, 26, 33, 39, 63, 65, 66, 76, 84, 88]
Quick Sort (sorted): [18, 26, 33, 39, 63, 65, 66, 76, 84, 88]
Quick Sort (reverse): [18, 26, 33, 39, 63, 65, 66, 76, 84, 88]
Merge Sort (random): [18, 26, 33, 39, 63, 65, 66, 76, 84, 88]
Merge Sort (sorted): [18, 26, 33, 39, 63, 65, 66, 76, 84, 88]
Merge Sort (reverse): [18, 26, 33, 39, 63, 65, 66, 76, 84, 88]

```

EXPLANATION: `def quick_sort(arr)::` This defines the recursive function `quick_sort` that takes a list `arr` as input.

`""" ... """`: This is a docstring explaining the function's purpose, arguments, and return value.

if len(arr) <= 1: This is the base case for the recursion. If the list has 0 or 1 element, it's already sorted, so the function simply returns the list.

pivot = arr[0]: This selects the first element of the list as the pivot. *Note: Different pivot selection strategies exist, and the choice of pivot significantly impacts performance.*

TASK-5: Give AI a naive algorithm (e.g., $O(n^2)$ duplicate search).

```
1 import time
2 import random
3
4 # --- Brute-Force Algorithm (O(n^2)) ---
5 def find_duplicates_brute_force(arr):
6     """
7     Finds duplicates using a brute-force O(n^2) approach with nested loops.
8     """
9     n = len(arr)
10    duplicates = set()
11    for i in range(n):
12        for j in range(i + 1, n):
13            if arr[i] == arr[j]:
14                duplicates.add(arr[i])
15    return list(duplicates)
16
17 # --- Optimized Algorithm (O(n)) ---
18 def find_duplicates_optimized(arr):
19     """
20     Finds duplicates using an optimized O(n) approach with a hash set.
21     """
22    seen = set()
23    duplicates = set()
24    for item in arr:
25        if item in seen:
26            duplicates.add(item)
27        else:
28            seen.add(item)
29    return list(duplicates)
30
31 # --- Performance Comparison ---
32 def run_performance_test(size=10000):
33     """
34     Generates a large list and compares the execution times of both algorithms.
35     """
36    # Create a large list with some duplicates
37    data = [random.randint(1, size // 2) for _ in range(size)]
38
39    print(f"Testing algorithms with a list of size {size}...")
40    print("-" * 40)
41
42    # Test Brute-Force
43    start_time = time.time()
44    brute_force_duplicates = find_duplicates_brute_force(data)
45    end_time = time.time()
46    brute_force_time = end_time - start_time
47    print(f"Brute-Force Algorithm completed in: {brute_force_time:.4f} seconds")
48
49    # Test Optimized
50    start_time = time.time()
51    optimized_duplicates = find_duplicates_optimized(data)
52    end_time = time.time()
53    optimized_time = end_time - start_time
54    print(f"Optimized Algorithm completed in: {optimized_time:.4f} seconds")
55
56    print("-" * 40)
57    print(f"Found {len(brute_force_duplicates)} duplicates using brute-force.")
58    print(f"Found {len(optimized_duplicates)} duplicates using optimized method.")
```


The screenshot shows the VS Code editor with the file `ai-5.py` open. The file contains a `run_performance_test` function that tests two duplicate-finding algorithms: a brute-force algorithm and an optimized algorithm using sets. The chat window on the right displays two versions of a duplicate-finder algorithm: 1. Brute Force ($O(n^2)$) and 2. Optimized ($O(n)$ using set).

```

32 def run_performance_test(size=10000):
33     # Test Brute Force
34     brute_force_duplicates = find_duplicates_brute_force(data)
35     brute_force_time = time.time() - start_time
36     print(f"Brute-Force Algorithm completed in: {brute_force_time:.4f} seconds")
37
38     # Test Optimized
39     start_time = time.time()
40     optimized_duplicates = find_duplicates_optimized(data)
41     end_time = time.time()
42     optimized_time = end_time - start_time
43     print(f"Optimized Algorithm completed in: {optimized_time:.4f} seconds")
44
45     print("\n" * 40)
46     print(f"Found {len(brute_force_duplicates)} duplicates using brute-force.")
47     print(f"Found {len(optimized_duplicates)} duplicates using optimized method.")
48
49     # Show the speedup factor
50     if optimized_time > 0:
51         speedup = brute_force_time / optimized_time
52         print(f"\nOptimization provided a speedup of approximately {speedup:.2f}x.")
53
54     # Run the test
55     if __name__ == "__main__":
56         run_performance_test()

```

The screenshot shows the VS Code terminal with the output of the `ai-5.py` script. The output displays the execution times for both the brute-force and optimized algorithms, showing that the optimized algorithm is significantly faster (approximately 1755.19x speedup).

```

'49866' '--' 'c:\Users\Devi\OneDrive\Desktop\ai-5.py'
Testing algorithms with a list of size 10,000...
-----
Brute-Force Algorithm completed in: 1.7592 seconds
Testing algorithms with a list of size 10,000...
-----
Brute-Force Algorithm completed in: 1.7592 seconds
-----
Brute-Force Algorithm completed in: 1.7592 seconds
Brute-Force Algorithm completed in: 1.7592 seconds
Optimized Algorithm completed in:  0.0010 seconds
Optimized Algorithm completed in:  0.0010 seconds
-----
Found 2987 duplicates using brute-force.
Found 2987 duplicates using optimized method.
Found 2987 duplicates using optimized method.

Optimization provided a speedup of approximately 1755.19x.
PS C:\Users\Devi\OneDrive\Desktop>

```

EXPLANATION:

- **performance_data = []:** This initializes an empty list called `performance_data`. This list will be used to store dictionaries, where each dictionary will represent a row in the final performance table.
- **for size, data_types in execution_times.items():** This is the outer loop that iterates through the `execution_times` dictionary. `execution_times` likely contains the measured execution times for different list sizes.
- **for data_type, algorithms in data_types.items():** This inner loop iterates through the data types within each list size (e.g., 'random', 'sorted', 'reverse-sorted').

