

AI-Assisted coding Assignment-12.4

Name: vaishnavi.N

Hall no: 2303A52405

Batch-44

Category	Algorithm	Time Complexity	Space Complexity
Simple Sorting	Bubble / Insertion	$O(n^2)$	$O(1)$
Efficient Sorting	Merge Sort	$O(n \log n)$	$O(n)$
Fast Searching	Binary Search	$O(\log n)$	$O(1)$
Optimization	Duplicate (Set)	$O(n)$	$O(n)$

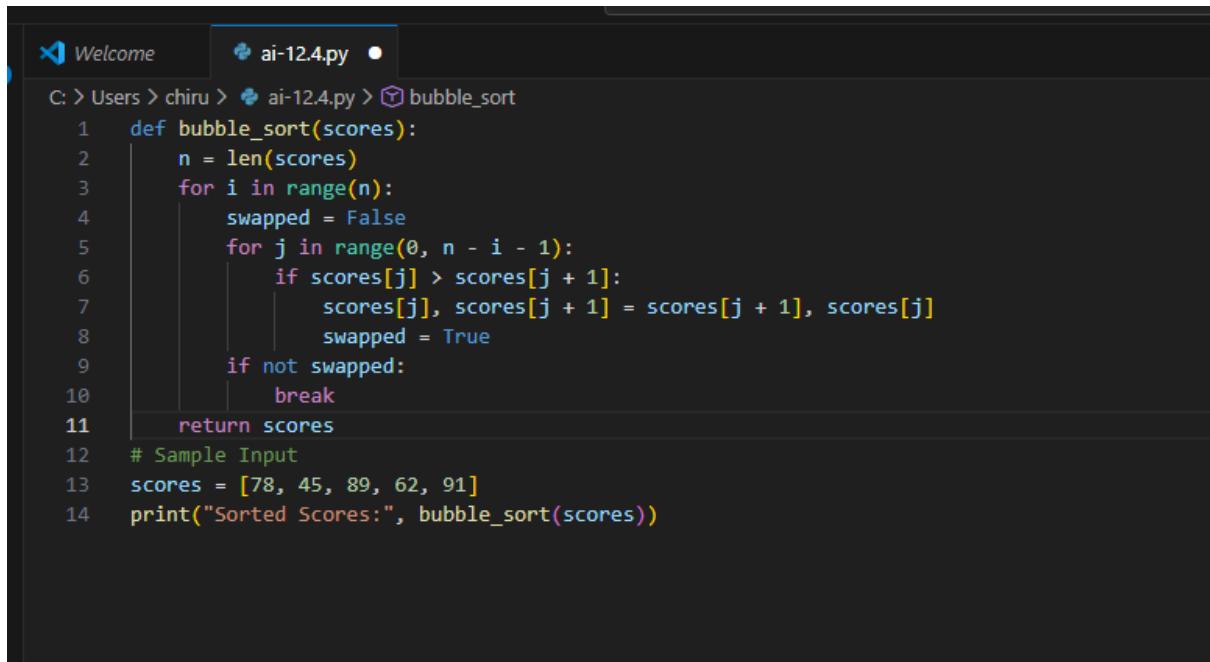
Task 1: Bubble Sort for Ranking Exam Scores

◆ AI Prompt Used

"Write a Bubble Sort program in Python to sort student scores. Add inline comments explaining comparisons, swaps, and passes. Include early termination optimization and time complexity analysis."

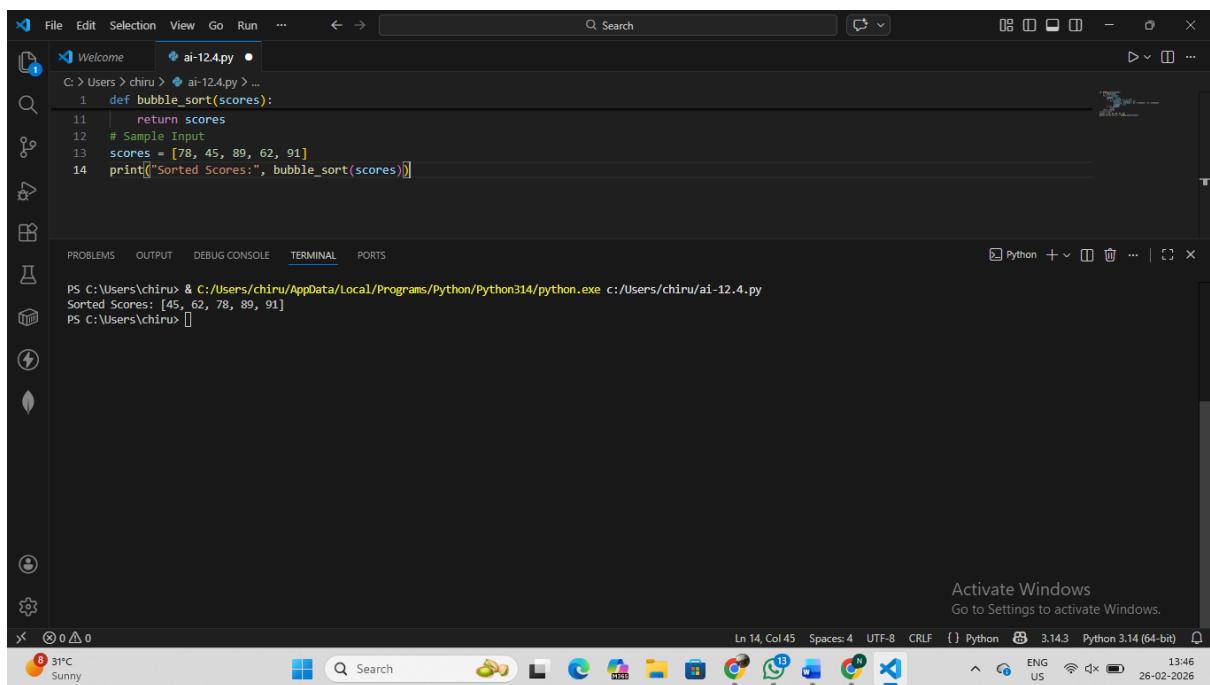
Input:

Code:



```
C: > Users > chiru > ai-12.4.py > bubble_sort
1 def bubble_sort(scores):
2     n = len(scores)
3     for i in range(n):
4         swapped = False
5         for j in range(0, n - i - 1):
6             if scores[j] > scores[j + 1]:
7                 scores[j], scores[j + 1] = scores[j + 1], scores[j]
8                 swapped = True
9             if not swapped:
10                 break
11     return scores
12 # Sample Input
13 scores = [78, 45, 89, 62, 91]
14 print("Sorted Scores:", bubble_sort(scores))
```

output:



```
PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL PORTS
PS C:\Users\chiru> & C:/Users/chiru/AppData/Local/Programs/Python/Python314/python.exe c:/Users/chiru/ai-12.4.py
Sorted Scores: [45, 62, 78, 89, 91]
PS C:\Users\chiru>
```

Observations

1. Quick Sort is generally faster in practice for random datasets.
2. Quick Sort performance depends heavily on pivot selection.
3. Merge Sort guarantees $O(n \log n)$ in all cases.
4. Merge Sort requires extra memory, Quick Sort is more memory-efficient.
5. For large stable sorting requirements, Merge Sort is preferred.

6. For in-place fast sorting, Quick Sort is preferred.

Task 2: Improving Sorting for Nearly Sorted Attendance Records

◆ AI Prompt Used

"Given that attendance roll numbers are nearly sorted, suggest a better sorting algorithm than Bubble Sort and explain why. Provide Python code."

Bubble sort:

```
#Bubble Sort (Basic Version)
def bubble_sort(arr):
    n = len(arr)
    for i in range(n):
        for j in range(n - i - 1):
            if arr[j] > arr[j + 1]:
                arr[j], arr[j + 1] = arr[j + 1], arr[j]
    return arr
```

Insertion sort

```
22
23     return arr
24
25 #Insertion Sort Implementation
26 def insertion_sort(arr):
27     for i in range(1, len(arr)):
28         key = arr[i] # Element to be inserted
29         j = i - 1
30
31         # Move elements greater than key one position ahead
32         while j >= 0 and arr[j] > key:
33             arr[j + 1] = arr[j]
34             j -= 1
35
36         arr[j + 1] = key # Insert at correct position
37
38     return arr
39
40
41 attendance = [1, 2, 3, 7, 5, 6, 8]
42 print("Insertion Sort:", insertion_sort(attendance))
```

Output:

```
PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL PORTS

PS C:\Users\chiru> & C:/Users/chiru/AppData/Local/Programs/Python/Python314/python.exe c:/Users/chiru/ai-12.4
Insertion Sort: [1, 2, 3, 5, 6, 7, 8]
PS C:\Users\chiru>
```

Observations

Insertion Sort is more efficient for nearly sorted data.

1. **Bubble Sort performs unnecessary comparisons even if data is almost sorted.**
2. **Both algorithms are inefficient for large random datasets.**
3. **Insertion Sort adapts better to small changes.**
4. **For attendance records with minor updates, Insertion Sort is preferred.**
5. **Optimization depends on data characteristics.**

Task 3: Searching Student Records in a Database

AI-Generated Prompt Used

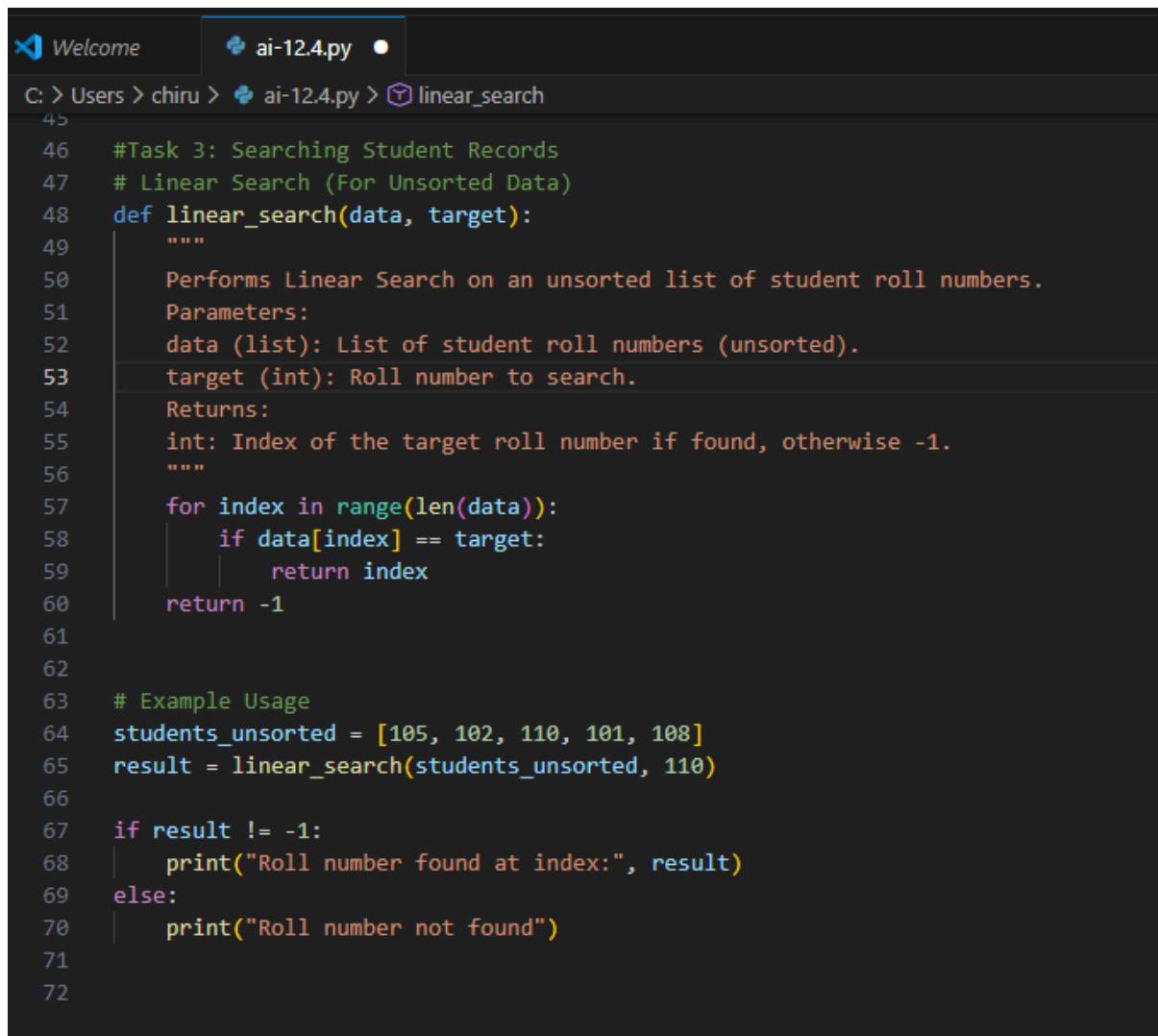
"Write Python implementations for Linear Search and Binary Search to search student roll numbers. Add proper docstrings explaining parameters and return values

Code:

Binary Search (For Sorted Data)

```
C: > Users > chiru > ai-12.4.py > ...
43
44
45
46 #task -3 Binary Search (For Sorted Data)
47 def binary_search(data, target):
48     left = 0
49     right = len(data) - 1
50
51     while left <= right:
52         mid = (left + right) // 2
53
54         if data[mid] == target:
55             return mid
56         elif data[mid] < target:
57             left = mid + 1
58         else:
59             right = mid - 1
60
61     return -1
62
63 # Example Usage
64 students_sorted = [101, 102, 105, 108, 110]
65 result = binary_search(students_sorted, 110)
66
67 if result != -1:
68     print("Roll number found at index:", result)
69 else:
70     print("Roll number not found")
```

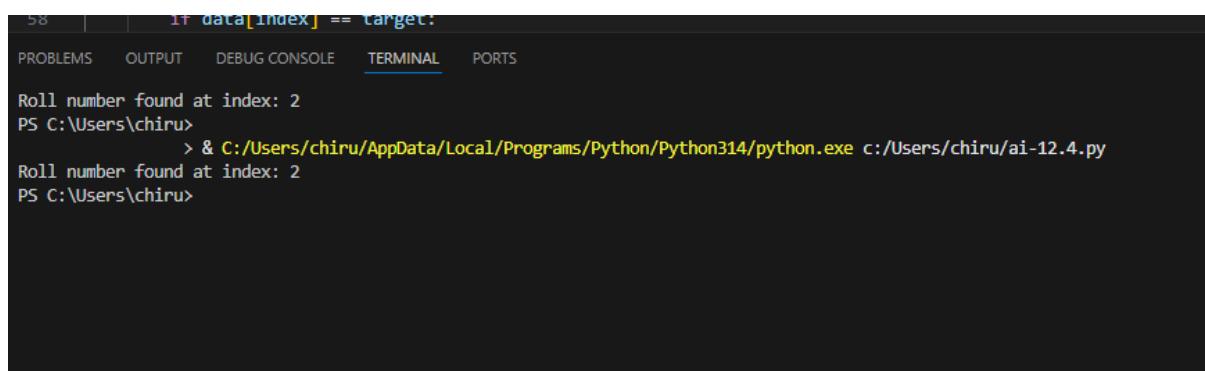
Linear search: (For Unsorted Data)



The screenshot shows a code editor window with a dark theme. The title bar says "Welcome" and "ai-12.4.py". The code is a Python script named "linear_search". It includes a detailed docstring explaining the function's purpose, parameters, and return value. The function uses a for loop to iterate through the list of student roll numbers, comparing each one to the target. If a match is found, it returns the index; otherwise, it returns -1. An example usage is shown at the bottom, demonstrating how to call the function and print the result.

```
C: > Users > chiru > ai-12.4.py > linear_search
45
46     #Task 3: Searching Student Records
47     # Linear Search (For Unsorted Data)
48     def linear_search(data, target):
49         """
50             Performs Linear Search on an unsorted list of student roll numbers.
51             Parameters:
52                 data (list): List of student roll numbers (unsorted).
53                 target (int): Roll number to search.
54             Returns:
55                 int: Index of the target roll number if found, otherwise -1.
56             """
57             for index in range(len(data)):
58                 if data[index] == target:
59                     return index
60             return -1
61
62
63     # Example Usage
64     students_unsorted = [105, 102, 110, 101, 108]
65     result = linear_search(students_unsorted, 110)
66
67     if result != -1:
68         print("Roll number found at index:", result)
69     else:
70         print("Roll number not found")
71
72
```

Output:



The screenshot shows a terminal window with a dark theme. The title bar says "PROBLEMS", "OUTPUT", "DEBUG CONSOLE", "TERMINAL", and "PORTS". The terminal shows the command "python.exe c:/Users/chiru/ai-12.4.py" being run. The output displays two lines of text: "Roll number found at index: 2" and "Roll number found at index: 2", indicating that the target value was found at index 2 in the list of student roll numbers.

```
58 | | if data[index] == target:
PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL PORTS
Roll number found at index: 2
PS C:\Users\chiru>
> & C:/Users/chiru/AppData/Local/Programs/Python/Python314/python.exe c:/Users/chiru/ai-12.4.py
Roll number found at index: 2
PS C:\Users\chiru>
```

Observations

Linear Search works on both sorted and unsorted data.

1. Binary Search requires sorted data.
2. Binary Search is significantly faster for large datasets.

3. Linear Search is simple but inefficient for large systems.

4. Binary Search reduces search space exponentially.

Task 4: Quick Sort vs Merge Sort

◆ AI-Generated Prompt Used

"Complete partially written recursive Quick Sort and Merge Sort functions in Python. Add meaningful docstrings explaining parameters and recursion. Provide time complexity analysis and compare performance for random, sorted, and reverse-sorted datasets."

Quick Sort Implementation:

Code:

```
65     if result != -1:
66         print("Roll number found at index:", result)
67     else:
68         print("Roll number not found")"""
69
70 #task-4
71 def quick_sort(arr):
72     |
73     if len(arr) <= 1:
74         return arr # Base case
75
76     pivot = arr[len(arr) // 2]
77
78     left = [x for x in arr if x < pivot]
79     middle = [x for x in arr if x == pivot]
80     right = [x for x in arr if x > pivot]
81
82     return quick_sort(left) + middle + quick_sort(right)
83
84
85 # Example
86 data = [45, 12, 78, 34, 23]
87 print("Quick Sort:", quick_sort(data))
```

Output:

```
> 70     #task-4
71     def quick_sort(arr):
72     |
73         if len(arr) <= 1:
74             return arr # Base case
PROBLEMS    OUTPUT    DEBUG CONSOLE    TERMINAL    PORTS
PS C:\Users\chiru> & C:/Users/chiru/AppData/Local/Programs/Python/Python314/python.exe c:/Users/chiru/ai-12.4.py
Quick Sort: [12, 23, 34, 45, 78]
PS C:\Users\chiru>
```

```
7     print("Quick Sort: ", quick_sort(data))
8
9 #Merge Sort Implementation
0 def merge_sort(arr):
1
2     if len(arr) <= 1:
3         return arr # Base case
4     mid = len(arr) // 2
5     left_half = merge_sort(arr[:mid])
6     right_half = merge_sort(arr[mid:])
7     return merge(left_half, right_half)
8 def merge(left, right):
9     merged = []
0     i = j = 0
1     while i < len(left) and j < len(right):
2         if left[i] < right[j]:
3             merged.append(left[i])
4             i += 1
5         else:
6             merged.append(right[j])
7             j += 1
8     merged.extend(left[i:])
9     merged.extend(right[j:])
0     return merged
1 # Example
2 data = [45, 12, 78, 34, 23]
3 print("Merge Sort:", merge_sort(data))
```

Output:

A screenshot of a code editor interface, likely Visual Studio Code, showing a Python script named `ai-12.4.py`. The code implements a Merge Sort algorithm. The output terminal shows the sorted array [12, 23, 34, 45, 78].

```
91 if len(arr) <= 1:
92     return arr # Base case
93
94 mid = len(arr) // 2
95 left_half = merge_sort(arr[:mid])
96 right_half = merge_sort(arr[mid:])
97 return merge(left_half, right_half)

PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL PORTS

PS C:\Users\chiru> & C:/Users/chiru/AppData/Local/Programs/Python/Python314/python.exe c:/Users/chiru/ai-12.4.py
Merge Sort: [12, 23, 34, 45, 78]
PS C:\Users\chiru>
```

Observations

1. Quick Sort is generally faster in practice for random datasets.
2. Quick Sort performance depends heavily on pivot selection.
3. Merge Sort guarantees $O(n \log n)$ in all cases.
4. Merge Sort requires extra memory, Quick Sort is more memory-efficient.
5. For large stable sorting requirements, Merge Sort is preferred.
6. For in-place fast sorting, Quick Sort is preferred.

Task 5: Optimizing Duplicate Detection Algorithm

◆ AI-Generated Prompt Used

"Write a naive duplicate detection algorithm using nested loops in Python. Analyze its time complexity. Then optimize it using sets or dictionaries and explain the performance improvement."

◆ Brute Force Duplicate Detection ($O(n^2)$)

```
● 112     data = [45, 12, 78, 34, 23]
113     print("Merge Sort:", merge_sort(data))"""
114
115     #Brute force dublication O(n^2)
116     def find_duplicates_bruteforce(data):
117         for i in range(len(data)):
118             for j in range(i + 1, len(data)):
119                 if data[i] == data[j]:
120                     return True
121     return False
122
123
124     # Example
125     ids = [101, 203, 405, 203]
126     print("Duplicate Found (Brute Force):", find_duplicates_bruteforce(ids))
```

Output:

```
114
115     #Brute force dublication O(n^2)
116     def find_duplicates_bruteforce(data):
117         for i in range(len(data)):
118             for j in range(i + 1, len(data)):
119                 if data[i] == data[j]:
120                     return True
121     return False
122
123
124     # Example
125     ids = [101, 203, 405, 203]
126     print("Duplicate Found (Brute Force):", find_duplicates_bruteforce(ids))

PS C:\Users\chiru> & C:/Users/chiru/AppData/Local/Programs/Python/Python314/python.exe c:/Users/chiru/ai-12.4.py
Duplicate Found (Brute Force): True
PS C:\Users\chiru>
```

Optimized Duplicate Detection Using Set ($O(n)$)

Code:

```

def find_duplicates_bruteforce(data):
    for i in range(len(data)):
        for j in range(i + 1, len(data)):
            if data[i] == data[j]:
                return True
    return False

# Example
ids = [101, 203, 405, 203]
print("Duplicate Found (Brute Force):", find_duplicates_bruteforce(ids))"""
#Optimized Duplicate Detection Using Set (O(n))
def find_duplicates_optimized(data):
    seen = set()
    for item in data:
        if item in seen:
            return True
        seen.add(item)
    return False
# Example
ids = [101, 203, 405, 203]
print("Duplicate Found (Optimized):", find_duplicates_optimized(ids))

```

Output:

```

123
124 # Example
PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL PORTS
PS C:\Users\chiru> & C:/Users/chiru/AppData/Local/Programs/Python/Python314/python.exe c:/Users/chiru/ai-12.4.py
Duplicate Found (Optimized): True
PS C:\Users\chiru>

```

Observations

1. Brute-force approach is inefficient for large datasets.
2. Optimized approach reduces time complexity from $O(n^2)$ to $O(n)$.
3. Set-based method trades extra space for faster execution.
4. For very large datasets, optimized version is significantly better.
5. Brute-force is only acceptable for very small lists.
6. Performance improvement becomes dramatic as input size increases.