

AI ASS 12.4

Name: A.Harshita

H.T.NO: 2303A52211

Batch:43

Task1:

Write a Python program to implement Bubble Sort for sorting student exam scores.

Add inline comments explaining comparisons, swaps, and passes.

Include early termination using a swapped flag.

Provide sample input/output and briefly explain best, average, and worst-case time complexity.

```
OPEN EDITORS
  ▾ OPEN EDITORS
    ▾ AI ASS - 1
      ▾ AI 2.4.py
      ▾ AI 4.4.py
      ▾ AI 9.1.py
      ✘ AI 12.4.py
      ▾ AI 14.py
      ▾ AI 24.py
      ▾ AI 44.py
      ▾ AI 91.py
    ▾ AI ASS - 2
      ▾ AI 12.4.py
      ▾ AI Task1
      ▾ bubble sort...
      ▾ def bubble_sort(scores):
      ▾   n = len(scores)
      ▾   for i in range(n):
      ▾     swapped = False
      ▾     for j in range(0, n - i - 1):
      ▾       if scores[j] > scores[j + 1]:
      ▾         scores[j], scores[j + 1] = scores[j + 1], scores[j]
      ▾         swapped = True
      ▾     if not swapped:
      ▾       print("List became sorted after pass: ", i+1)
      ▾       break
      ▾   return scores
      student_scores = [78, 45, 89, 62, 55]
      sorted_scores = bubble_sort(student_scores)
      print("Sorted Scores: ", sorted_scores)

PROBLEMS  OUTPUT  DEBUG CONSOLE  TERMINAL  FODE
PS C:\Users\Samjal\OneDrive\Desktop\AI assistant coding-> c:\Users\Samjal\OneDrive\Desktop\AI assistant coding> python -m venv venv
c:\Users\Samjal\OneDrive\Desktop\AI assistant coding> venv\Scripts\python -m pip install -U pygments
c:\Users\Samjal\OneDrive\Desktop\AI assistant coding> venv\Scripts\pygmentize -f html -O style=vs -l python -f html -o venv\Scripts\pygments.css
c:\Users\Samjal\OneDrive\Desktop\AI assistant coding> venv\Scripts\pygmentize -f html -O style=vs -l python -f html -o venv\Scripts\pygments.css
List became sorted after pass 4
Sorted Scores: [45, 55, 62, 78, 89]
PS C:\Users\Samjal\OneDrive\Desktop\AI assistant coding>
```

Observation:

Bubble Sort was implemented to sort student scores after internal assessments.

The algorithm shows $O(n)$ time complexity in the best case when the list is already sorted due to early termination.

In the average case, when scores are randomly arranged, the time complexity is $O(n^2)$.

In the worst case, when scores are in reverse order, it also takes $O(n^2)$ time due to maximum comparisons and swaps.

Thus, Bubble Sort is suitable for small datasets but not efficient for large datasets.

Task2:

I have a nearly sorted list of student roll numbers (attendance records).

1. Implement Bubble Sort in Python.
2. Suggest a more suitable sorting algorithm for nearly sorted data.
3. Implement Insertion Sort in Python.
4. Explain why Insertion Sort performs better on nearly sorted datasets.
5. Compare their behavior on a nearly sorted input example.
6. Include time complexity analysis.

The screenshot shows the PyCharm IDE interface. The left sidebar displays a file tree with several Python files (Ass 2.4.py, Ass 4.4.py, etc.) and a 'MATH' folder containing 'Ass 12.4.py'. The main editor window contains two snippets of Python code: one for bubble sort and one for insertion sort. The terminal window at the bottom shows the command line running the code and printing the original and sorted lists for both methods.

```
1 #!/usr/bin/python3
2
3 def bubble_sort(arr):
4     n = len(arr)
5     for i in range(0, n - 1):
6         swapped = False
7         for j in range(0, n - i - 1):
8             if arr[j] > arr[j + 1]:
9                 arr[j], arr[j + 1] = arr[j + 1], arr[j]
10                swapped = True
11
12    return arr
13
14 def insertion_sort(arr):
15     for i in range(1, len(arr)):
16         key = arr[i]
17         j = i - 1
18         while j >= 0 and arr[j] > key:
19             arr[j + 1] = arr[j]
20             j -= 1
21         arr[j + 1] = key
22
23 attendance_marks = [10, 12, 14, 16, 18, 19]
24
25 # calling both functions separately gives us the result modifying same list
26 print("Original Attendance Records:", attendance_marks)
27 print("Bubble Sort Output:", bubble_sort(attendance_marks))
28 print("Insertion Sort Output:", insertion_sort(attendance_marks))
```

```
PS C:\Users\Sameer\Desktop\AI\assistant coding> cd 'C:\Users\Sameer\Desktop\AI\assistant coding'; &
  setenv VIRTUAL_ENV=C:\Users\Sameer\Desktop\AI\assistant coding
  set _OLD_VIRTUAL_PATH=%PATH%
  set PATH=%VIRTUAL_ENV%\Scripts;%PATH%
  set _OLD_VIRTUAL_PROMPT=%PROMPT%
  set PROMPT=$([System.IO.File]::GetLastWriteTime('C:\Windows\system32\cmd.exe').ToString('MM-dd-yyyy'))> 
original Attendance Records: [10, 12, 14, 16, 18, 19]
Bubble Sort Output: [10, 12, 14, 16, 18, 19]
Insertion Sort Output: [10, 12, 14, 16, 18, 19]
```

Observation:

- 1.The attendance roll numbers were nearly sorted with only a few misplaced elements.
- 2.Bubble Sort performed multiple comparisons even when only small corrections were needed.
- 3.Insertion Sort shifted only the misplaced elements to their correct positions.
- 4.For nearly sorted data, Insertion Sort behaved close to **O(n)** time complexity.
5. Bubble Sort still required more passes, making it comparatively slower.
- 6.Therefore, Insertion Sort is more efficient and suitable for partially sorted attendance records.

Task3:

Building a student information portal.

1. Implement Linear Search for unsorted student roll numbers.
2. Implement Binary Search for sorted roll numbers.
3. Add proper Python docstrings explaining parameters and return values.
4. Explain when Binary Search can be used.
5. Compare time complexity and performance differences.
6. Provide sample input and output.
7. Add a short observation comparing results on sorted vs unsorted data.

The screenshot shows a Visual Studio Code interface with the following details:

- File Explorer:** Shows files in the current workspace, including `Ass 2.4.py`, `Ass 4.4.py`, `Ass 8.1.py`, `Ass 12.4.py`, `Ass 1.4.py`, `Ass 2.4.py`, `Ass 4.4.py`, `Ass 9.1.py`, and `Ass 12.4.ipynb`.
- Editor:** The main editor area displays Python code for two search algorithms:

```
def linear_search(student_rolls, target):
    # Traverse each element one by one
    for index in range(len(student_rolls)):
        if student_rolls[index] == target:
            return index
    return -1

def binary_search(student_rolls, target):
    low = 0
    high = len(student_rolls) - 1
    while low <= high:
        mid = (low + high) // 2
        if student_rolls[mid] == target:
            return mid
        elif student_rolls[mid] < target:
            low = mid + 1
        else:
            high = mid - 1
    return -1

unsorted_rolls = [9, 2, 8, 10, 2]
sorted_rolls = sorted(unsorted_rolls)
target_roll = 10

print("Linear search result (unsorted data):",
      linear_search(unsorted_rolls, target_roll))
print("Binary search result (unsorted data):",
      binary_search(unsorted_rolls, target_roll))

print("Linear search result (sorted data):",
      linear_search(sorted_rolls, target_roll))
print("Binary search result (sorted data):",
      binary_search(sorted_rolls, target_roll))
```
- Terminal:** The bottom terminal window shows the output of running the script:

```
PS C:\Users\Suraj\OneDrive\Desktop\AI assistant coding> cd "C:\Users\Suraj\OneDrive\Desktop\AI assistant coding\b"
PS C:\Users\Suraj\OneDrive\Desktop\AI assistant coding\b> python Ass 12.4.py
Linear Search Result (Unsorted Data): 3
Binary Search Result (Unsorted Data): 3
Linear Search Result (Sorted Data): 4
Binary Search Result (Sorted Data): 4
```

Observation:

Linear Search checks each element one by one, so its time complexity is $O(n)$ in average and worst cases. It works on both sorted and unsorted lists.

Binary Search divides the sorted list into halves repeatedly, so its time complexity is $O(\log n)$. It works only on sorted data.

For unsorted lists, Linear Search is used.

For large sorted lists, Binary Search is faster and more efficient.

Task4:

work with large datasets that may be random, already sorted, or reverse sorted.

1. Complete recursive implementations of Quick Sort and Merge Sort in Python.
 2. Add proper docstrings explaining parameters and return values.

3. Explain how recursion works in both algorithms.

4. Test both algorithms on:

- Random data

- Sorted data

- Reverse sorted data

5. Compare best, average, and worst-case time complexity.

6. Explain practical scenarios where one is preferred over the other.

The screenshot shows the VS Code interface with several files open in the Explorer sidebar. The active file is `Ans 12.4.py`, which contains the implementation of both quick sort and merge sort. The code is as follows:

```
def quick_sort(arr):
    if len(arr) <= 1:
        return arr
    pivot = arr[-1]
    left = [x for x in arr[:-1] if x <= pivot]
    right = [x for x in arr[:-1] if x > pivot]
    return quick_sort(left) + [pivot] + quick_sort(right)

def merge_sort(arr):
    if len(arr) <= 1:
        return arr
    mid = len(arr) // 2
    left_half = merge_sort(arr[:mid])
    right_half = merge_sort(arr[mid:])
    return merge(left_half, right_half)

def merge(left, right):
    result = []
    i = j = 0
    while i < len(left) and j < len(right):
        if left[i] < right[j]:
            result.append(left[i])
            i += 1
        else:
            result.append(right[j])
            j += 1
    result.extend(left[i:])
    result.extend(right[j:])
    return result
```

Below the code, the terminal window shows the execution of the sorting functions with random, sorted, and reverse sorted data points.

```
Quick Sort - Random: [1, 12, 23, 34, 45, 56, 67]
Quick Sort - Sorted: [1, 12, 23, 34, 45, 56, 67]
Quick Sort - Reversed: [1, 12, 23, 34, 45, 56, 67]
Merge Sort - Random: [1, 12, 23, 34, 45, 56, 67]
Merge Sort - Sorted: [1, 12, 23, 34, 45, 56, 67]
Merge Sort - Reversed: [1, 12, 23, 34, 45, 56, 67]
```

The screenshot shows the VS Code interface with several files open in the Explorer sidebar. The active file is `Ans 12.4.py`, which contains the implementation of merge sort and tests for different data types. The code is as follows:

```
def merge(left, right):
    result = []
    while i < len(left) and j < len(right):
        if left[i] < right[j]:
            result.append(left[i])
            i += 1
        else:
            result.append(right[j])
            j += 1
    result.extend(left[i:])
    result.extend(right[j:])
    return result

random_data = [45, 12, 37, 34, 23, 67, 1]
already_sorted_data = sorted(random_data)
sorted_data = sorted(random_data)
reverse_data = sorted(random_data, reverse=True)

print("Quick Sort - Random:", quick_sort(random_data))
print("Quick Sort - Sorted:", quick_sort(sorted_data))
print("Quick Sort - Reversed:", quick_sort(reverse_data))

print("Merge Sort - Random:", merge_sort(random_data))
print("Merge Sort - Sorted:", merge_sort(sorted_data))
print("Merge Sort - Reversed:", merge_sort(reverse_data))
```

Below the code, the terminal window shows the execution of merge sort with random, sorted, and reverse sorted data points.

```
Quick Sort - Random: [1, 12, 23, 34, 45, 56, 67]
Quick Sort - Sorted: [1, 12, 23, 34, 45, 56, 67]
Quick Sort - Reversed: [1, 12, 23, 34, 45, 56, 67]
Merge Sort - Random: [1, 12, 23, 34, 45, 56, 67]
Merge Sort - Sorted: [1, 12, 23, 34, 45, 56, 67]
Merge Sort - Reversed: [1, 12, 23, 34, 45, 56, 67]
```

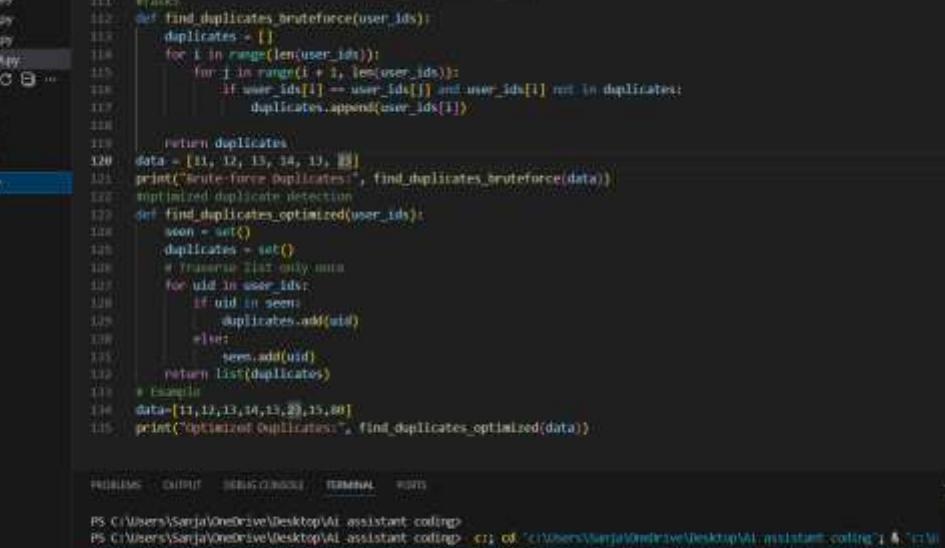
Observation:

1. Quick Sort has best and average time complexity of $O(n \log n)$.
2. Quick Sort worst case is $O(n^2)$ when pivot selection is poor.
3. Merge Sort has $O(n \log n)$ time complexity in best, average, and worst cases.
4. Quick Sort uses less memory (in-place), while Merge Sort requires extra memory.
5. Quick Sort is usually faster for random datasets.
6. Merge Sort is preferred when guaranteed performance and stable sorting are required.

Task5:

Building a data validation module to detect duplicate user IDs.

1. Write a brute-force duplicate detection algorithm using nested loops.
2. Analyze its time complexity.
3. Suggest an optimized approach using a set or dictionary.
4. Rewrite the algorithm with improved efficiency.
5. Compare performance conceptually for large datasets.
6. Keep explanation simple and clear.



The screenshot shows a Visual Studio Code interface with the following details:

- File Explorer:** Shows files: Ass 2.4.py, Ass 4.4.py, Ass 9.1.py, Ass 12.4.py (highlighted), Ass 12.4.pyc, Ass 12.4.pyw.
- Open Editors:** Shows the content of Ass 12.4.py.
- Code Editor:** Displays Python code for finding duplicates. It includes two implementations: a brute-force approach and an optimized set-based approach. The optimized version is annotated with comments explaining its performance characteristics.
- Terminal:** Shows command-line output for both implementations, demonstrating they produce the same results (duplicates: [13]).

```
111 #!/usr/bin/python
112 def find_duplicates_bruteforce(user_ids):
113     duplicates = []
114     for i in range(len(user_ids)):
115         for j in range(i+1, len(user_ids)):
116             if user_ids[i] == user_ids[j] and user_ids[i] not in duplicates:
117                 duplicates.append(user_ids[i])
118
119     return duplicates
120
121 data = [11, 12, 13, 14, 13, 15]
122 print("Brute-force duplicates:", find_duplicates_bruteforce(data))
123
124 optimized duplicate detection
125
126 def find_duplicates_optimized(user_ids):
127     seen = set()
128     duplicates = set()
129     # Traverse list only once
130     for uid in user_ids:
131         if uid in seen:
132             duplicates.add(uid)
133         else:
134             seen.add(uid)
135     return list(duplicates)
136
137 # example
138 data=[11,12,13,14,13,15,16]
139 print("Optimized duplicates:", find_duplicates_optimized(data))
```

PS C:\Users\Sanja\OneDrive\Desktop\AI\assistant\coding>

PS C:\Users\Sanja\OneDrive\Desktop\AI\assistant\coding> cd "C:\Users\Sanja\OneDrive\Desktop\AI\assistant\coding" & "C:\Windows\Python\Python37\python.exe" "C:\Users\Sanja\vscode\extensions\ms-python.debug-2025.10.2\src\server\extension\lib\util\MyPyLanceServer.py" "unit" "C:\Users\Sanja\OneDrive\Desktop\AI\assistant\coding\Ass 12.4.py"

Brute-force duplicates: [13]

Optimized duplicates: [13]

PS C:\Users\Sanja\OneDrive\Desktop\AI\assistant\coding>

Observation:

- 1.The brute-force algorithm uses nested loops to compare every element with all others, resulting in $O(n^2)$ time complexity.
 - 2.It performs a large number of unnecessary comparisons, especially for large datasets.
 - 3.The optimized algorithm uses a set (or dictionary) to store already seen elements.
 - 4.Since set lookup takes $O(1)$ time, the entire list is processed in a single loop, giving $O(n)$ time complexity.
 - 5.The performance improved because repeated comparisons were replaced with fast lookups.
 - 6.Therefore, the optimized approach is much faster and more suitable for large datasets.