

Assignment 12.4 Ai Assisted Coding

Htno:2303a51305

Btno:05

Task 1:

Bubble Sort for Ranking Exam

Scores

Scenario

You are working on a college result processing system where a small list of student scores needs to be sorted after every internal assessment.

Prompt:

Write a simple Python program for Bubble Sort to sort student exam scores. Add inline comments for comparisons and swaps, include early termination if already sorted, give time complexity (best, average, worst), and show sample input/output.

Code:

The screenshot shows the VS Code interface with the following details:

- File Explorer (Left):** Shows files in the "AI CODING" folder, including bubblesort.py, COMPARISON.md, contact_manager_array.py, contact_manager_linked_list.py, digits.py, discount.py, docstring.py, even number.py, evennumber.py, fibonacci.py, file_operations.py, number.py, prime_docstring.py, sample.txt, string.py, sum.py, task1.py, and threenumbers.py.
- Code Editor (Center):** The active file is bubblesort.py. The code implements both bubble sort and insertion sort, comparing their performance on nearly sorted data.
- Chat Panel (Right):** A chat titled "BUBBLE SORT IMPLEMENTATION..." is open, showing a history of messages:
 - ✓ Reviewed bubblesort.py
 - I'll create a comprehensive solution comparing Bubble Sort and Insertion Sort for nearly sorted student attendance roll numbers:
 - ✓ Created bubblesort.py
 - Let me read the file again:
 - ✓ Reviewed bubblesort.py
 - Let me try reading the entire file to see its exact format:
 - ✓ Reviewed bubblesort.py
- Bottom Status Bar:** Shows the line number (Ln 2), column (Col 65), spaces (Spaces: 4), encoding (UTF-8), and file type (base (3.13.5)).

The screenshot shows the VS Code interface with the following details:

- File Explorer (Left):** Shows files in the "AI CODING" folder, including bubblesort.py, COMPARISON.md, contact_manager_array.py, contact_manager_linked_list.py, digits.py, discount.py, docstring.py, even number.py, evennumber.py, fibonacci.py, file_operations.py, number.py, prime_docstring.py, sample.txt, string.py, sum.py, task1.py, and threenumbers.py.
- Code Editor (Center):** The active file is bubblesort.py. The code implements both bubble sort and insertion sort, comparing their performance on nearly sorted data.
- Chat Panel (Right):** A chat titled "BUBBLE SORT IMPLEMENTATION..." is open, showing a history of messages:
 - ✓ Reviewed bubblesort.py
 - I'll create a comprehensive solution comparing Bubble Sort and Insertion Sort for nearly sorted student attendance roll numbers:
 - ✓ Created bubblesort.py
 - Let me read the file again:
 - ✓ Reviewed bubblesort.py
 - Let me try reading the entire file to see its exact format:
 - ✓ Reviewed bubblesort.py
- Bottom Status Bar:** Shows the line number (Ln 2), column (Col 65), spaces (Spaces: 4), encoding (UTF-8), and file type (base (3.13.5)).

This screenshot shows the VS Code interface with the 'bubblesort.py' file open. The code implements a step-by-step demo of bubble sort. It iterates through the array from index i to $n-1$, comparing adjacent elements. If arr[j] > arr[j+1], it swaps them and increments swap_count. The process continues until no swaps are made, indicating early termination.

```
211 def step_by_step_demo():
212     n = len(arr)
213     comp_count = 0
214     shift_count = 0
215
216     print(f"\nInitial: {arr}\n")
217
218     for i in range(1, n):
219         key = arr[i]
220         j = i - 1
221         comparisons_in_iteration = 0
222         shifts_in_iteration = 0
223
224         print(f"Insert arr[{i}]={key} into sorted region {arr[:i]}:")
225
226         # Find correct position and shift elements
227         while j >= 0 and arr[j] > key:
228             comp_count += 1
229             comparisons_in_iteration += 1
230             print(f" Compare {arr[j]} > {key} → True, shift {arr[j]} right")
231             arr[j + 1] = arr[j]
232             shift_count += 1
233             shifts_in_iteration += 1
234             j -= 1
235
236     print(f"\nFinal: {arr}\n")
237
238     print(f"Total Comparisons: {comp_count}, Total Shifts: {shift_count}")
239
240
241
242
243
244
245
246
247
248
249
250
251
252
253
254
255
256
257
258
259
260
261
262
263
264
265
266
267
```

This screenshot shows the VS Code interface with the 'bubblesort.py' file open. The code implements a step-by-step demo of bubble sort. It iterates through the array from index i to $n-1$, comparing adjacent elements. If arr[j] > arr[j+1], it swaps them and increments swap_count. The process continues until no swaps are made, indicating early termination.

```
211 def step_by_step_demo():
212     n = len(arr)
213     comp_count = 0
214     shift_count = 0
215
216     print(f"\nInitial: {arr}\n")
217
218     for j in range(n - 1):
219         for i in range(n - 1 - j):
220             if arr[i] > arr[i + 1]:
221                 arr[i], arr[i + 1] = arr[i + 1], arr[i]
222                 comp_count += 1
223
224             print(f" Compare arr[{i}]={arr[i]} vs arr[{i+1}]={arr[i+1]}, end={arr[i+1]}")
225
226             if arr[i] == arr[i + 1]:
227                 print(f" → No swap")
228
229             else:
230                 print(f" → SWAP")
231
232     print(f"\nFinal: {arr}\n")
233
234     print(f"Total Comparisons: {comp_count}, Total Shifts: {shift_count}")
235
236
237
238
239
240
241
242
243
244
245
246
247
248
249
250
251
252
253
254
255
256
257
258
259
260
261
262
263
264
265
266
267
268
269
270
271
272
273
274
275
276
277
278
279
280
281
282
283
284
285
286
287
288
289
290
291
292
293
294
295
296
297
298
299
```

The screenshot shows the Visual Studio Code interface. The Explorer sidebar on the left lists files under 'AI CODING' such as bubblesort.py, COMPARISON.md, contact_manager_array.py, contact_manager_linked_list.py, digits.py, discount.py, docstring.py, even number.py, evennumber.py, fibonacci.py, file_operations.py, number.py, prime_docstring.py, sample.txt, string.py, sum.py, task1.py, and threenumbers.py. The 'OUTLINE' and 'TIMELINE' sections are also visible. The main editor area displays the contents of bubblesort.py:

```
392
393 # =====
394 # MAIN EXECUTION
395 # =====
396
397 if __name__ == "__main__":
398
399     # 1. Print explanation first
400     print_explanation()
401
402     # 2. Demonstrate step-by-step behavior
403     step_by_step_demo()
404
405     # 3. Performance comparison
406     detailed_comparison()
407
408     # 4. Final recommendation
409     print("\n" + "=" * 80)
410     print("FINAL RECOMMENDATION FOR ATTENDANCE ROLL NUMBER SORTING")
411     print("=" * 80)
412     print"""
413 For a college attendance system tracking student roll numbers:
```

The status bar at the bottom indicates the code is in Python mode, using base (3.13.5), with Ln 295, Col 39, and other settings like Spaces: 4, UTF-8, CRLF.

Output:

EXPLORER PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL POSTMAN CONSOLE

AI CODING

- bubblesort.py
- COMPARISON.md
- contact_manager_array.py
- contact_manager_linked_list.py
- digits.py
- discount.py
- docstring.py
- even number.py
- evennumber.py
- fibonacci.py
- file_operations.py
- number.py
- prime_docstring.py
- sample.txt
- string.py
- sum.py
- task1.py
- threenumbers.py

OUTLINE TIMELINE

TERMINAL

```
=====
WHY INSERTION SORT IS MORE EFFICIENT FOR NEARLY SORTED DATA
=====
```

1. EARLY LOOP TERMINATION

- Insertion Sort: Inner while loop terminates as soon as correct position found
- Bubble Sort: Must complete ALL comparisons in each pass, even if no swaps occur

For nearly sorted data:

- Insertion Sort breaks early (few elements out of place)
- Bubble Sort must pass through most of the list unnecessarily

2. ADAPTIVE ALGORITHM

- Insertion Sort adapts to the input: $O(n)$ when nearly sorted
- Bubble Sort requires $O(n^2)$ even with early termination optimization

Key insight: For a list with k out-of-place elements:

- Insertion Sort: $O(n + k \cdot n)$ complexity
- Bubble Sort: $O(n^2)$ always

Ln 295, Col 39 Spaces: 4 UTF-8 CRLF Python base (3.13.5) Go Live Go Live Go Live

EXPLORER PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL POSTMAN CONSOLE

AI CODING

- bubblesort.py
- COMPARISON.md
- contact_manager_array.py
- contact_manager_linked_list.py
- digits.py
- discount.py
- docstring.py
- even number.py
- evennumber.py
- fibonacci.py
- file_operations.py
- number.py
- prime_docstring.py
- sample.txt
- string.py
- sum.py
- task1.py
- threenumbers.py

OUTLINE TIMELINE

TERMINAL

```
=====
ALGORITHM SELECTION GUIDE:
=====
```

Use INSERTION SORT when:

- ✓ Data is nearly sorted (5-20% out of order)
- ✓ Small to medium datasets (< 5000 elements)
- ✓ Simplicity and cache efficiency matter
- ✓ Online/streaming sorting needed
- ✓ Stability is required (maintains relative order of equal elements)

Use BUBBLE SORT when:

- ✓ Teaching sorting concepts
- ✓ Educational purposes
- ✓ Already sorted data (with early termination)
- ✓ Very small datasets

Avoid both for large unsorted datasets - use Quicksort, Mergesort, or Heapsort instead.

```
=====
STEP-BY-STEP DEMONSTRATION ON SMALL NEARLY SORTED LIST
=====
```

Ln 295, Col 39 Spaces: 4 UTF-8 CRLF Python base (3.13.5) Go Live Go Live Go Live

Nearly Sorted Roll Numbers: [1, 2, 4, 3, 5, 6, 7, 8, 9, 10]
(Only 3 and 4 are out of order)

BUBBLE SORT - TRACE

Initial: [1, 2, 4, 3, 5, 6, 7, 8, 9, 10]

Pass 1:

```
Compare arr[0]=1 vs arr[1]=2 → No swap
Compare arr[1]=2 vs arr[2]=4 → No swap
Compare arr[2]=4 vs arr[3]=3 → SWAP
Compare arr[3]=4 vs arr[4]=5 → No swap
Compare arr[4]=5 vs arr[5]=6 → No swap
Compare arr[5]=6 vs arr[6]=7 → No swap
Compare arr[6]=7 vs arr[7]=8 → No swap
Compare arr[7]=8 vs arr[8]=9 → No swap
Compare arr[8]=9 vs arr[9]=10 → No swap
After Pass 1: [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
```

Pass 2:

```
Compare arr[0]=1 vs arr[1]=2 → No swap
```

Ln 295, Col 39 Spaces:4 UTF-8 CRLF Python base (3.13.5) ⚡ Go Live ⚡ Go Live ⚡ Go Live

After Pass 1: [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]

Pass 2:

```
Compare arr[0]=1 vs arr[1]=2 → No swap
Compare arr[1]=2 vs arr[2]=3 → No swap
Compare arr[2]=3 vs arr[3]=4 → No swap
Compare arr[3]=4 vs arr[4]=5 → No swap
Compare arr[4]=5 vs arr[5]=6 → No swap
Compare arr[5]=6 vs arr[6]=7 → No swap
Compare arr[6]=7 vs arr[7]=8 → No swap
Compare arr[7]=8 vs arr[8]=9 → No swap
Compare arr[8]=9 vs arr[9]=10 → No swap
After Pass 2: [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
✓ No swaps - EARLY TERMINATION at Pass 2
```

Bubble Sort Result: [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
Total Comparisons: 17, Total Swaps: 1

INSERTION SORT - TRACE

Initial: [1, 2, 4, 3, 5, 6, 7, 8, 9, 10]

Insert arr[1]=2 into sorted region [1]:

Ln 295, Col 39 Spaces:4 UTF-8 CRLF Python base (3.13.5) ⚡ Go Live ⚡ Go Live ⚡ Go Live

The screenshot shows the VS Code interface with the terminal window open, displaying the trace of an insertion sort algorithm. The terminal output is as follows:

```
INITIAL: [1, 2, 4, 3, 5, 6, 7, 8, 9, 10]
Insert arr[1]=2 into sorted region [1]:
  Compare 1 > 2 → False, position found
  Insert 2 at index 1
  Array now: [1, 2, 4, 3, 5, 6, 7, 8, 9, 10] (Comparisons: 1, Shifts: 0)

Insert arr[2]=4 into sorted region [1, 2]:
  Compare 2 > 4 → False, position found
  Insert 4 at index 2
  Array now: [1, 2, 4, 3, 5, 6, 7, 8, 9, 10] (Comparisons: 1, Shifts: 0)

Insert arr[3]=3 into sorted region [1, 2, 4]:
  Compare 4 > 3 → True, shift 4 right
  Compare 2 > 3 → False, position found
  Insert 3 at index 2
  Array now: [1, 2, 3, 4, 5, 6, 7, 8, 9, 10] (Comparisons: 2, Shifts: 1)

Insert arr[4]=5 into sorted region [1, 2, 3, 4]:
  Compare 4 > 5 → False, position found
  Insert 5 at index 3
  Array now: [1, 2, 3, 4, 5, 6, 7, 8, 9, 10] (Comparisons: 2, Shifts: 1)

Insert arr[5]=6 into sorted region [1, 2, 3, 4, 5]:
  Compare 5 > 6 → False, position found
  Insert 6 at index 4
  Array now: [1, 2, 3, 4, 5, 6, 7, 8, 9, 10] (Comparisons: 1, Shifts: 0)

Insert arr[6]=7 into sorted region [1, 2, 3, 4, 5, 6]:
  Compare 6 > 7 → False, position found
  Insert 7 at index 5
  Array now: [1, 2, 3, 4, 5, 6, 7, 8, 9, 10] (Comparisons: 1, Shifts: 0)

Insert arr[7]=8 into sorted region [1, 2, 3, 4, 5, 6, 7]:
  Compare 7 > 8 → False, position found
  Insert 8 at index 6
  Array now: [1, 2, 3, 4, 5, 6, 7, 8, 9, 10] (Comparisons: 1, Shifts: 0)

Insert arr[8]=9 into sorted region [1, 2, 3, 4, 5, 6, 7, 8]:
  Compare 8 > 9 → False, position found
  Insert 9 at index 7
  Array now: [1, 2, 3, 4, 5, 6, 7, 8, 9, 10] (Comparisons: 1, Shifts: 0)
```

The screenshot shows the VS Code interface with the terminal window open, displaying the continuation of the insertion sort algorithm's trace. The terminal output is as follows:

```
Insert arr[9]=10 into sorted region [1, 2, 3, 4, 5, 6, 7, 8]:
  Compare 9 > 10 → False, position found
  Insert 10 at index 8
  Array now: [1, 2, 3, 4, 5, 6, 7, 8, 9, 10] (Comparisons: 1, Shifts: 0)
```

The screenshot shows a Microsoft Visual Studio Code (VS Code) interface. The left sidebar has sections for EXPLORER, TERMINAL, OUTLINE, and TIMELINE. The EXPLORER section shows a folder named 'AI CODING' containing files like bubblesort.py, COMPARISON.md, contact_manager_array.py, contact_manager_linked_list.py, digits.py, discount.py, docstring.py, even_number.py, evennumber.py, fibonacci.py, file_operations.py, number.py, prime_docstring.py, sample.txt, string.py, sum.py, task1.py, and threenumbers.py. The TERMINAL tab is active, displaying the output of a script comparing sorting algorithms. The output shows results for a test with 100 student roll numbers, comparing Bubble Sort and Insertion Sort, and concludes with efficiency gains. The status bar at the bottom indicates the script is running in Python base (3.13.5).

```
COMPARISON: BUBBLE SORT vs INSERTION SORT FOR NEARLY SORTED DATA
=====
TEST WITH 100 STUDENT ROLL NUMBERS (Nearly Sorted - 5% disorder)
=====
→ BUBBLE SORT:
Comparisons: 4,935
Swaps: 535
Time: 1.0836 ms

→ INSERTION SORT:
Comparisons: 632
Shifts: 535
Time: 0.1805 ms

✓ EFFICIENCY GAIN (Insertion Sort vs Bubble Sort):
Comparisons Reduced: 87.2%
Time Improvement: 83.3%
Speed Ratio: 6.00x

TEST WITH 500 STUDENT ROLL NUMBERS (Nearly Sorted - 5% disorder)
```

Task 3:

Searching Student Records in a

Database

Scenario

You are developing a student information portal where users search for student records by roll number.

Prompt:

Implement Linear Search and Binary Search in Python to search student records by roll number. Add proper docstrings explaining parameters and return values. Explain when Binary Search is applicable and compare performance differences. Also provide time complexity and a short observation comparing results on sorted vs unsorted lists.

Code:

The screenshot shows a code editor interface with a dark theme. The left sidebar has a tree view under 'EXPLORER' labeled 'AI CODING' containing files like bubblesort.py, COMPARISON.md, contact_manager_array.py, contact_manager_linked_list.py, digits.py, discount.py, docstring.py, even_number.py, evennumber.py, fibonacci.py, file_operations.py, number.py, prime_docstring.py, sample.txt, string.py, sum.py, task1.py, and threenumbers.py. The main editor area is titled 'Welcome' and shows the content of 'bubblesort.py'. The code implements a bubble sort algorithm and a linear search function. A status bar at the bottom indicates the file is in Python mode, with spaces: 4, UTF-8 encoding, and line 12, column 1.

```
1 # Student record example
2 students_unsorted = [
3     {"roll": 103, "name": "Asha"}, 
4     {"roll": 101, "name": "Ravi"}, 
5     {"roll": 105, "name": "Sneha"}, 
6     {"roll": 102, "name": "Kiran"}, 
7     {"roll": 104, "name": "Anil"}]
8
9
10 students_sorted = sorted(students_unsorted, key=lambda x: x["roll"])
11
12 def linear_search(students, target_roll):
13     """
14         Perform Linear Search to find a student by roll number.
15
16     Parameters:
17         students (list): List of student dictionaries (unsorted or sorted).
18         target_roll (int): Roll number to search for.
19
20     Returns:
21         dict or None: Student record if found, otherwise None.
22
23 
```

This screenshot shows the same code editor interface as the first one, but the main editor area now displays a Python script for binary search. The code defines a 'binary_search' function that takes a list of student records and a target roll number. It performs a binary search to find the student record. A status bar at the bottom indicates the file is in Python mode, with spaces: 4, UTF-8 encoding, and line 12, column 1.

```
30
31
32 def binary_search(students, target_roll):
33     """
34         Perform Binary Search to find a student by roll number.
35
36     Parameters:
37         students (list): Sorted list of student dictionaries (MUST be sorted by roll)
38         target_roll (int): Roll number to search for.
39
40     Returns:
41         dict or None: Student record if found, otherwise None.
42
43     Time Complexity: O(log n) - divides search space in half.
44     Applicable only on SORTED lists. Much faster than linear search.
45
46     low = 0
47     high = len(students) - 1
48
49     while low <= high:
50         mid = (low + high) // 2
51         if students[mid]["roll"] == target_roll:
52             return students[mid]
```

```
# Example usage and comparison
if __name__ == "__main__":
    print("Unsorted List:", students_unsorted)
    print("Sorted List:", students_sorted)
    print()

    # Search examples
    target = 102
    linear_result = linear_search(students_unsorted, target)
    binary_result = binary_search(students_sorted, target)

    print(f"Linear Search for roll {target}: {linear_result}")
    print(f"Binary Search for roll {target}: {binary_result}")
    print()

    # Observation
    print("Observations:")
    print("- Linear Search works on unsorted lists")
    print("- Binary Search requires sorted input")
    print("- For small lists (<100 elements), difference is negligible")
    print("- For large lists (>10,000 elements), Binary Search is significantly faster than Linear Search")
```

```
print(f"Linear Search for roll {target}: {linear_result}")
print(f"Binary Search for roll {target}: {binary_result}")
print()

# Observation
print("Observations:")
print("- Linear Search works on unsorted lists but requires O(n) time")
print("- Binary Search requires sorted input but is O(log n) - much faster for large lists")
print("- For small lists (<100 elements), difference is negligible")
print("- For large lists (>10,000 elements), Binary Search is significantly faster than Linear Search")
```

Output:

Explanation:

- **Linear Search** checks each student record one by one, so it works even if the data is unsorted.
- **Binary Search** divides the list into halves repeatedly, but it only works when the student records are sorted by roll number.
- Linear Search has **$O(n)$ time complexity**, meaning it may take longer as the number of records increases.
- Binary Search has **$O(\log n)$ time complexity**, making it much faster for large datasets.
- Use Linear Search for small or unsorted data, and Binary Search for large sorted databases where quick searching is needed.

Task 4:

Choosing Between Quick Sort and

Merge Sort for Data Processing

Scenario: You are part of a data analytics team that needs to sort large datasets received from different sources (random order, already sorted, and reverse sorted).

Prompt:

Complete recursive Quick Sort and Merge Sort functions with docstrings. Explain how recursion works, test on random, sorted, and reverse-sorted data, and compare time complexities with practical use cases.

Code:

The screenshot shows two instances of the VS Code code editor. Both instances have the same file structure in the Explorer sidebar, including files like bubblesort.py, COMPARISON.md, contact_manager_array.py, contact_manager_linked_list.py, digits.py, discount.py, docstring.py, even number.py, evennumber.py, fibonacci.py, file_operations.py, number.py, prime_docstring.py, sample.txt, string.py, sum.py, task1.py, and threenumbers.py.

Top Instance (Initial State):

```
1 import random
2 import time
3 from typing import List
4
5 """
6 Sorting Algorithms: Quick Sort and Merge Sort
7 Demonstrates recursive sorting with complexity analysis
8 """
9
10
11 def quick_sort(arr: List[int], low: int = 0, high: int = None) -> List[int]:
12     """
13         Quick Sort using divide-and-conquer recursion.
14
15         How recursion works:
16             1. Base case: if low >= high, array is sorted
17             2. Partition: choose pivot, split array into smaller/larger elements
18             3. Recursive case: recursively sort left and right partitions
19
20         Time Complexity:
21             - Best/Average: O(n log n)
22             - Worst: O(n^2) when pivot is always smallest/largest
23
24
25
26
27
28
29
30
31
32
33
34
35
36
37
38
39
40
41
42
43
44
45
46
47
48
```

Bottom Instance (Refactored State):

```
12 def quick_sort(arr: List[int], low: int = 0, high: int = None) -> List[int]:
13     """
14         if high is None:
15             high = len(arr) - 1
16
17         if low < high:
18             # Partition and get pivot index
19             pivot_index = _partition(arr, low, high)
20
21             # Recursively sort left partition
22             quick_sort(arr, low, pivot_index - 1)
23
24             # Recursively sort right partition
25             quick_sort(arr, pivot_index + 1, high)
26
27
28
29
30
31
32
33
34
35
36
37
38
39
40
41
42
43
44
45
46
47
48
```

In the bottom instance, the `_partition` function has been moved to its own file, `COMPARISON.md`, which now contains the following content:

```
"""
Helper function to partition array for Quick Sort."""
pivot = arr[high]
i = low - 1
for i in range(low, high):
```

The screenshot shows the VS Code interface with the following details:

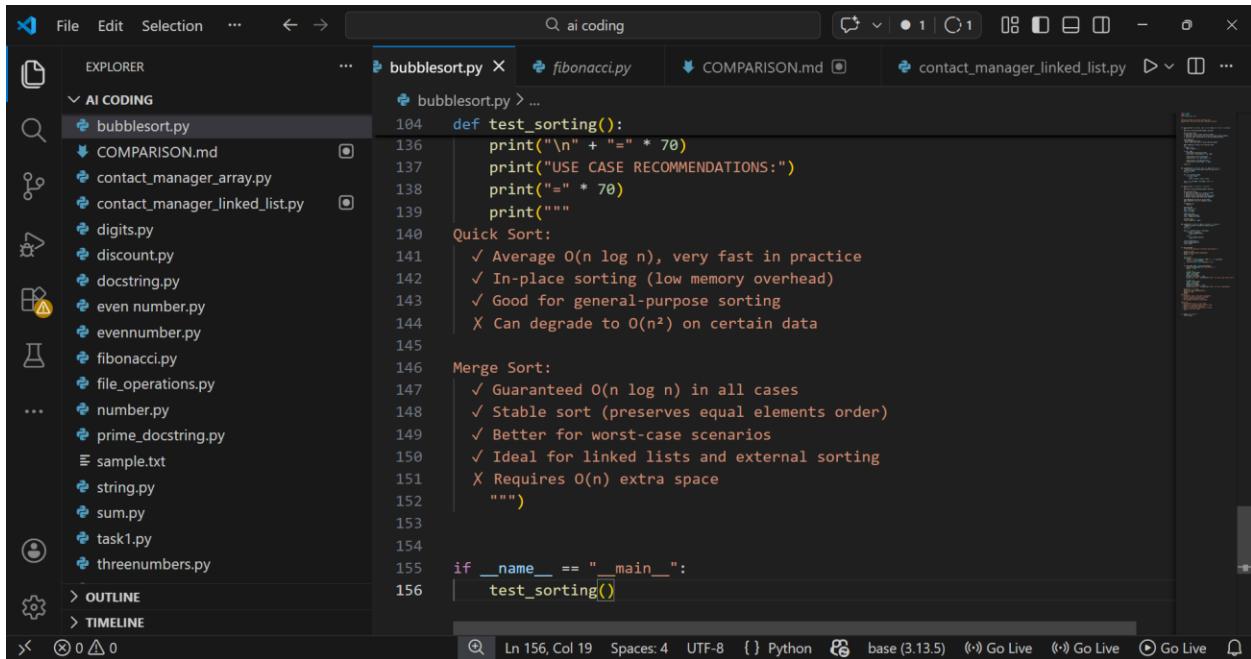
- File Explorer:** Shows files in the "AI CODING" folder, including bubblesort.py, COMPARISON.md, contact_manager_array.py, contact_manager_linked_list.py, digits.py, discount.py, even_number.py, evennumber.py, fibonacci.py, file_operations.py, number.py, prime_docstring.py, sample.txt, string.py, sum.py, task1.py, and threenumbers.py.
- Code Editor:** The active file is bubblesort.py. The code implements the Merge Sort algorithm with the following structure:

```
def merge_sort(arr: List[int]) -> List[int]:  
    """  
    Merge Sort using divide-and-conquer recursion.  
  
    How recursion works:  
    1. Base case: if length <= 1, array is sorted  
    2. Divide: split array into two halves  
    3. Recursive case: recursively sort both halves  
    4. Conquer: merge sorted halves back together  
  
    Time Complexity: O(n log n) in all cases  
    Space Complexity: O(n) for temporary arrays  
    """  
  
    if len(arr) <= 1:  
        return arr  
  
    # Divide step  
    mid = len(arr) // 2  
    left = arr[:mid]  
    right = arr[mid:]  
  
    # Recursive step
```
- Status Bar:** Shows Ln 156, Col 19, Spaces: 4, UTF-8, Python, base (3.13.5), and three Go Live buttons.

The screenshot shows the VS Code interface with the following details:

- File Explorer:** Shows files in the "AI CODING" folder, including bubblesort.py, COMPARISON.md, contact_manager_array.py, contact_manager_linked_list.py, digits.py, discount.py, even_number.py, evennumber.py, fibonacci.py, file_operations.py, number.py, prime_docstring.py, sample.txt, string.py, sum.py, task1.py, and threenumbers.py.
- Code Editor:** The active file is bubblesort.py. The code now includes a test function:

```
def test_sorting():  
    """Test both algorithms on different data patterns."""  
  
    print("=" * 70)  
    print("SORTING ALGORITHM COMPARISON")  
    print("=" * 70)  
  
    # Test cases  
    test_cases = {  
        "Random": [random.randint(1, 1000) for _ in range(100)],  
        "Sorted": list(range(100)),  
        "Reverse-Sorted": list(range(100, 0, -1))  
    }  
  
    for test_name, data in test_cases.items():  
        print(f"\n{test_name} Data (100 elements):")  
        print("-" * 70)  
  
        # Quick Sort  
        qs_data = data.copy()  
        start = time.time()  
        quick_sort(qs_data)
```
- Status Bar:** Shows Ln 156, Col 19, Spaces: 4, UTF-8, Python, base (3.13.5), and three Go Live buttons.



The screenshot shows a dark-themed instance of Visual Studio Code. The left sidebar has a 'File Explorer' section titled 'AI CODING' containing files like bubblesort.py, COMPARISON.md, contact_manager_array.py, contact_manager_linked_list.py, digits.py, discount.py, docstring.py, even number.py, evennumber.py, fibonacci.py, file_operations.py, number.py, prime_docstring.py, sample.txt, string.py, sum.py, task1.py, and threenumbers.py. Below this are sections for 'OUTLINE' and 'TIMELINE'. The main editor area is titled 'bubblesort.py X' and contains the following Python code:

```
104     def test_sorting():
105         print("\n" + "=" * 70)
106         print("USE CASE RECOMMENDATIONS:")
107         print("=" * 70)
108         print("")
109
110         Quick Sort:
111             ✓ Average O(n log n), very fast in practice
112             ✓ In-place sorting (low memory overhead)
113             ✓ Good for general-purpose sorting
114             ✗ Can degrade to O(n2) on certain data
115
116         Merge Sort:
117             ✓ Guaranteed O(n log n) in all cases
118             ✓ Stable sort (preserves equal elements order)
119             ✓ Better for worst-case scenarios
120             ✓ Ideal for linked lists and external sorting
121             ✗ Requires O(n) extra space
122             """
123
124
125     if __name__ == "__main__":
126         test_sorting()
```

The status bar at the bottom indicates the code is in 'Ln 156, Col 19' with 'Spaces: 4' and 'UTF-8'. It also shows the Python extension is version 3.13.5 and there are three 'Go Live' buttons.

Output:

The image shows two side-by-side terminal sessions in VS Code, both titled "ai coding".

Top Terminal Session:

```

=====
SORTING ALGORITHM COMPARISON
=====

Random Data (100 elements):
-----
Quick Sort: 0.3142ms - O(n log n) avg, O(n²) worst
Merge Sort: 0.6332ms - O(n log n) guaranteed

Sorted Data (100 elements):
-----
Quick Sort: 1.0798ms - O(n log n) avg, O(n²) worst
Merge Sort: 0.4272ms - O(n log n) guaranteed

Reverse-Sorted Data (100 elements):
-----
Quick Sort: 0.6008ms - O(n log n) avg, O(n²) worst
Merge Sort: 0.2117ms - O(n log n) guaranteed

=====
USE CASE RECOMMENDATIONS:
=====

Quick Sort:
    ✓ Average O(n log n), very fast in practice
    ✓ In-place sorting (low memory overhead)
    ✓ Good for general-purpose sorting
    ✗ Can degrade to O(n²) on certain data

Merge Sort:
    ✓ Guaranteed O(n log n) in all cases
    ✓ Stable sort (preserves equal elements order)
    ✓ Better for worst-case scenarios
    ✓ Ideal for linked lists and external sorting
    ✗ Requires O(n) extra space
  
```

Bottom Terminal Session:

```

=====
Reverse-Sorted Data (100 elements):
-----
Quick Sort: 0.6008ms - O(n log n) avg, O(n²) worst
Merge Sort: 0.2117ms - O(n log n) guaranteed

=====
USE CASE RECOMMENDATIONS:
=====

Quick Sort:
    ✓ Average O(n log n), very fast in practice
    ✓ In-place sorting (low memory overhead)
    ✓ Good for general-purpose sorting
    ✗ Can degrade to O(n²) on certain data

Merge Sort:
    ✓ Guaranteed O(n log n) in all cases
    ✓ Stable sort (preserves equal elements order)
    ✓ Better for worst-case scenarios
    ✓ Ideal for linked lists and external sorting
    ✗ Requires O(n) extra space
  
```

The terminals show the execution of a script named "bubblesort.py" which compares the performance of Quick Sort and Merge Sort on random, sorted, and reverse-sorted data sets. The bottom terminal session provides a detailed list of use cases and pros/cons for each algorithm.

Explanation:

- **Quick Sort** uses a pivot to divide the dataset into smaller parts and sorts them recursively, making it fast for random data.
- **Merge Sort** divides the dataset into halves and merges them after sorting, ensuring stable and consistent performance.
- Recursion works by repeatedly breaking the problem into smaller subproblems until they become simple to solve.

- Quick Sort is usually faster in practice but can perform poorly on already sorted or reverse-sorted data (worst case).
- Merge Sort guarantees **O(n log n)** performance and is preferred when stable sorting or predictable speed is required.

Task 5:

Optimizing a Duplicate Detection Algorithm

Scenario

You are building a data validation module that must detect duplicate user IDs in a large dataset before importing it into a system.

Prompt:

Implement a brute-force duplicate detection algorithm using nested loops, analyze its time complexity, then optimize it using a set or dictionary and compare performance for large datasets.

Code:

The screenshot shows the VS Code interface with the title bar "ai coding". The Explorer sidebar on the left shows files like bubblesort.py, COMPARISON.md, contact_manager_array.py, contact_manager_linked_list.py, digits.py, discount.py, even number.py, evennumber.py, fibonacci.py, file_operations.py, number.py, prime_docstring.py, sample.txt, string.py, sum.py, task1.py, and threenumbers.py. The bubblesort.py tab is active, displaying the following code:

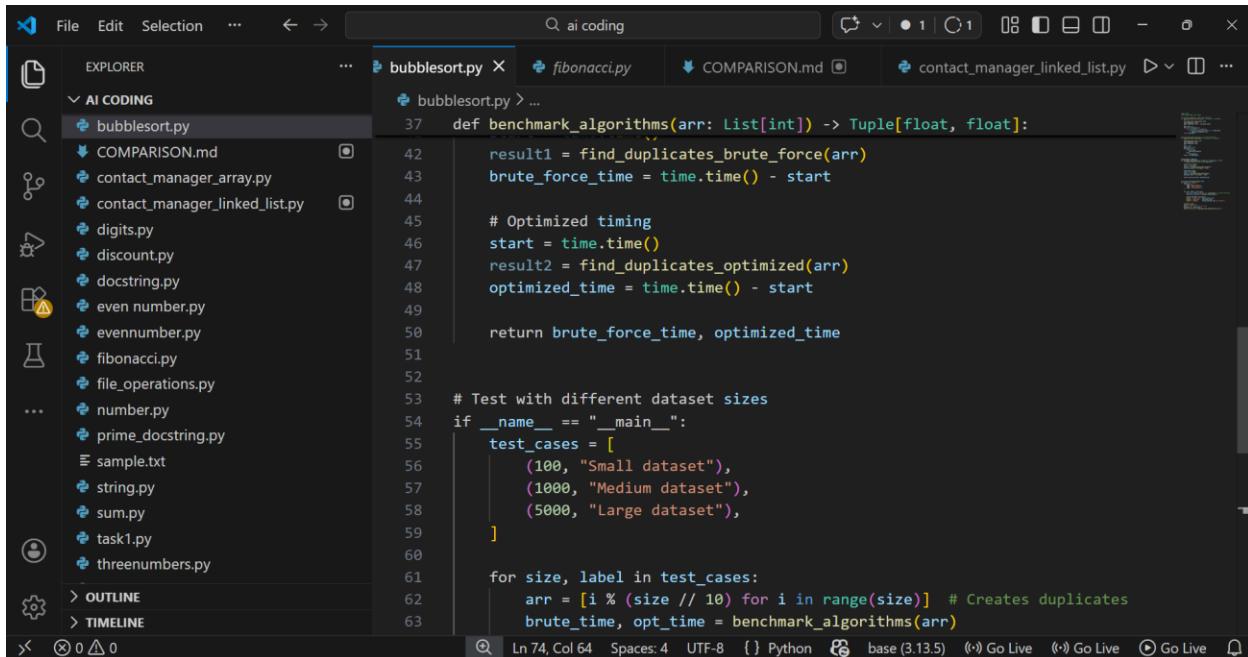
```
1 import time
2 from typing import List, Tuple
3
4 # Brute-force approach - O(n2) time complexity
5 def find_duplicates_brute_force(arr: List[int]) -> List[int]:
6     """
7         Find duplicates using nested loops.
8         Time Complexity: O(n2)
9         Space Complexity: O(1) - excluding output
10    """
11    duplicates = []
12    for i in range(len(arr)):
13        for j in range(i + 1, len(arr)):
14            if arr[i] == arr[j] and arr[i] not in duplicates:
15                duplicates.append(arr[i])
16    return duplicates
17
18
19 # Optimized approach using set - O(n) time complexity
20 def find_duplicates_optimized(arr: List[int]) -> List[int]:
21     """
22         Find duplicates using a set.
23         Time Complexity: O(n)
24     """
25
26
27
28
29
30
31
32
33
34
35
36
37
38
39
40
41
```

The status bar at the bottom indicates "Ln 74, Col 64 Spaces: 4 UTF-8 {} Python base (3.13.5) (↻) Go Live (↻) Go Live (↻) Go Live".

The screenshot shows the VS Code interface with the title bar "ai coding". The Explorer sidebar on the left shows the same files as the first screenshot. The bubblesort.py tab is active, displaying the following code:

```
19 # Optimized approach using set - O(n) time complexity
20 def find_duplicates_optimized(arr: List[int]) -> List[int]:
21     """
22         Find duplicates using a set.
23         Time Complexity: O(n)
24         Space Complexity: O(n)
25     """
26     seen = set()
27     duplicates = set()
28     for num in arr:
29         if num in seen:
30             duplicates.add(num)
31         else:
32             seen.add(num)
33     return list(duplicates)
34
35
36
37
38
39
40
41
```

The status bar at the bottom indicates "Ln 74, Col 64 Spaces: 4 UTF-8 {} Python base (3.13.5) (↻) Go Live (↻) Go Live (↻) Go Live".



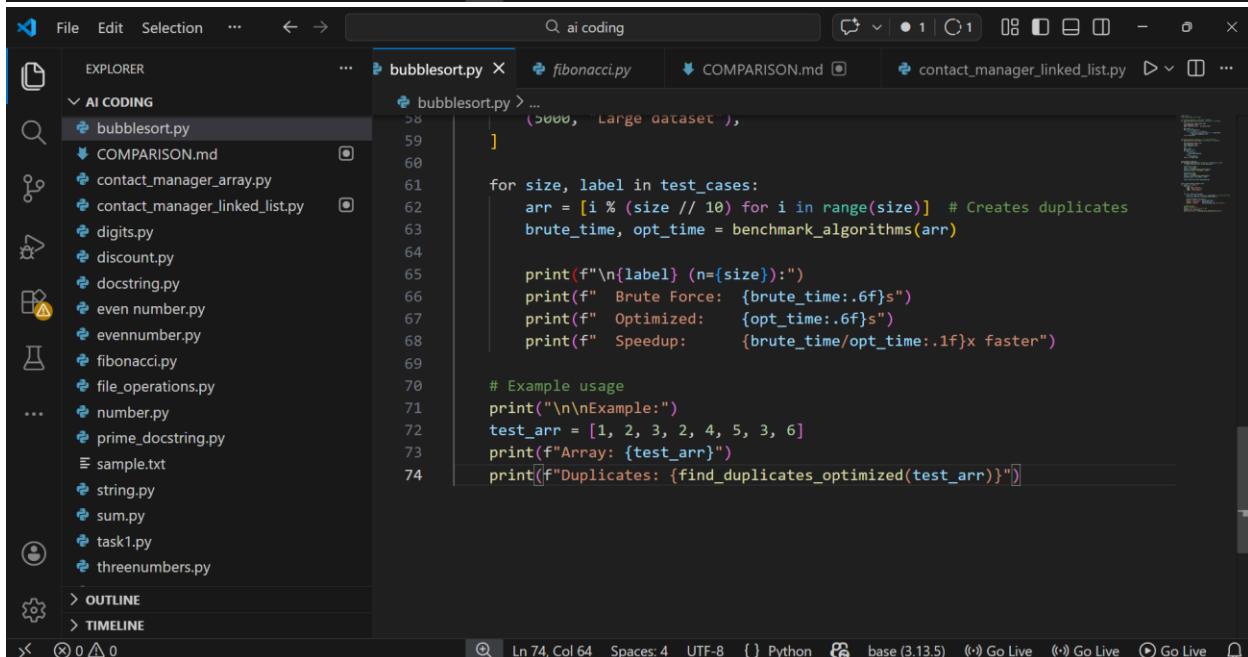
```
def benchmark_algorithms(arr: List[int]) -> Tuple[float, float]:
    result1 = find_duplicates_brute_force(arr)
    brute_force_time = time.time() - start

    # Optimized timing
    start = time.time()
    result2 = find_duplicates_optimized(arr)
    optimized_time = time.time() - start

    return brute_force_time, optimized_time

# Test with different dataset sizes
if __name__ == "__main__":
    test_cases = [
        (100, "Small dataset"),
        (1000, "Medium dataset"),
        (5000, "Large dataset"),
    ]

    for size, label in test_cases:
        arr = [i % (size // 10) for i in range(size)] # Creates duplicates
        brute_time, opt_time = benchmark_algorithms(arr)
```



```
(5000, "Large dataset"),
]

for size, label in test_cases:
    arr = [i % (size // 10) for i in range(size)] # Creates duplicates
    brute_time, opt_time = benchmark_algorithms(arr)

    print(f"\n{label} ({size}):")
    print(f"  Brute Force: {brute_time:.6f}s")
    print(f"  Optimized:   {opt_time:.6f}s")
    print(f"  Speedup:     {brute_time/opt_time:.1f}x faster")

# Example usage
print("\nExample:")
test_arr = [1, 2, 3, 2, 4, 5, 3, 6]
print(f"Array: {test_arr}")
print(f"Duplicates: {find_duplicates_optimized(test_arr)}")
```

Output:

The screenshot shows the Visual Studio Code interface. The Explorer sidebar on the left displays a folder named 'AI CODING' containing several Python files: bubblesort.py, COMPARISON.md, contact_manager_array.py, contact_manager_linked_list.py, digits.py, discount.py, docstring.py, even number.py, evennumber.py, fibonacci.py, file_operations.py, number.py, prime_docstring.py, sample.txt, string.py, sum.py, task1.py, and threenumbers.py. The 'bubblesort.py' file is selected. The Terminal tab at the top has the title 'TERMINAL'. The terminal window shows the output of running the 'bubblesort.py' script. It compares the execution time of a brute-force method against an optimized one for three dataset sizes: small (n=100), medium (n=1000), and large (n=5000). The output indicates significant speedups for the optimized version, especially for larger datasets. The status bar at the bottom shows the current file path as 'C:\Users\sneha\OneDrive\Documents\Desktop\ai coding>', the line and column numbers as 'Ln 74, Col 64', and the encoding as 'UTF-8'. It also shows icons for Python, base (3.13.5), and Go Live.

```
PS C:\Users\sneha\OneDrive\Documents\Desktop\ai coding> cd "c:\Users\sneha\OneDrive\Documents\Desktop\ai coding" ; python bubblesort.py

Small dataset (n=100):
Brute Force: 0.000991s
Optimized: 0.00029s
Speedup: 33.8x faster

Medium dataset (n=1000):
Brute Force: 0.064839s
Optimized: 0.000146s
Speedup: 444.4x faster

Large dataset (n=5000):
Brute Force: 1.103027s
Optimized: 0.000587s
Speedup: 1879.9x faster

Example:
Array: [1, 2, 3, 2, 4, 5, 3, 6]
Duplicates: [2, 3]
```

Explanation:

- The brute-force method uses nested loops to compare each user ID with every other ID, resulting in $O(n^2)$ time complexity.
- This approach becomes slow for large datasets because the number of comparisons grows rapidly.
- The optimized method uses a **set** or **dictionary** to track seen IDs, reducing the time complexity to $O(n)$.
- Performance improves because each ID is checked only once instead of repeatedly comparing with all others.
- For large inputs, the optimized approach runs much faster and is preferred for real-world data validation systems.

The screenshot shows the VS Code interface with the following details:

- EXPLORER** sidebar: Shows files like bubblesort.py, COMPARISON.md, contact_manager_array.py, contact_manager_linked_list.py, digits.py, discount.py, docstring.py, even_number.py, evennumber.py, fibonacci.py, file_operations.py, number.py, prime_docstring.py, sample.txt, string.py, sum.py, task1.py, and threenumbers.py.
- WORKBENCH** tab bar: Welcome, bubblesort.py (active), COMPARISON.md.
- bubblesort.py** code editor:

```
1 # Bubble Sort for Student Exam Scores
2 # Time Complexity: Best O(n), Average O(n2), Worst O(n2)
3
4 def bubble_sort(scores):
    """
    Sorts exam scores in ascending order using bubble sort algorithm.

    Args:
        scores: List of exam scores

    Returns:
        Sorted list of scores
    """
    n = len(scores)

    # Traverse through all array elements
    for i in range(n):
        swapped = False # Flag for early termination

        # Last i elements are already in place
        for j in range(0, n - i - 1):
            # Compare adjacent elements
            if scores[j] > scores[j + 1]:
```
- CHAT** panel: A prompt for "BUBBLE SORT IMPLEMENTATION..." with numbered steps:
 - Add inline comments explaining key steps such as comparisons, swaps, and each iteration pass.
 - Include an early termination condition to stop the algorithm if the list is already sorted.
 - Provide a brief time complexity analysis explaining best, average, and worst cases.
 - Show a sample input list of scores and the corresponding sorted output.A note below says: "Make the code simple and suitable for a college result processing scenario."
- COMMANDS** panel: Shows a list of recent files and build options.

The screenshot shows the VS Code interface with the following details:

- EXPLORER** sidebar: Same as the first screenshot.
- WORKBENCH** tab bar: Welcome, bubblesort.py (active), COMPARISON.md.
- bubblesort.py** code editor:

```
4 def bubble_sort(scores):
    ...
    # If no swaps occurred, array is already sorted
    if not swapped:
        break

    return scores

# Sample Input and Output
if __name__ == "__main__":
    # Input: Student exam scores
    exam_scores = [85, 92, 78, 95, 88, 76, 91, 83]

    print("Original Scores:", exam_scores)
    sorted_scores = bubble_sort(exam_scores)
    print("Sorted Scores:", sorted_scores)
```
- CHAT** panel: Same as the first screenshot.
- COMMANDS** panel: Same as the first screenshot.

The screenshot shows the VS Code interface with the AI Coding extension active. The Explorer sidebar is open, showing files under 'AI CODING' such as bubblesort.py, COMPARISON.md, contact_manager_array.py, and others. The main editor window displays Python code for bubble sort, with line numbers 35 to 49 visible. The Chat panel on the right contains a list of steps for implementing bubble sort, followed by a summary: "Make the code simple and suitable for a college result processing scenario." The status bar at the bottom indicates the file is base (3.13.5).

Output:

The screenshot shows the VS Code interface with the AI Coding extension active. The Explorer sidebar is open, showing files under 'AI CODING' such as bubblesort.py, COMPARISON.md, contact_manager_array.py, and others. The TERMINAL tab is selected, showing the command 'python bubblesort.py' being run in the terminal, followed by the output: 'Original Scores: [85, 92, 78, 95, 88, 76, 91, 83] Sorted Scores: [76, 78, 83, 85, 88, 91, 92, 95]' and '--- Early Termination Example ---'. The Chat panel on the right contains a list of steps for implementing bubble sort, followed by a summary: "Make the code simple and suitable for a college result processing scenario." The status bar at the bottom indicates the file is base (3.13.5).

Explanation:

- Bubble Sort compares adjacent exam scores and swaps them if they are in the wrong order.
- It repeats multiple passes until all scores are arranged correctly.
- After each pass, the largest score moves to its correct position (“bubbles up”).
- If no swaps happen in a pass, the algorithm stops early because the list is already sorted.
- Time complexity: Best — $O(n)$, Average — $O(n^2)$, Worst — $O(n^2)$.

Task 2:

Improving Sorting for Nearly Sorted

Attendance Records

Scenario

You are maintaining an attendance system where student roll numbers are already almost sorted, with only a few late updates.

Prompt: