AI ASSISTED CODING

LAB ASSIGNMENT: 12.4

NAME: T.Shirisha

H.NO : 2403A52077

BATCH NO : 04

TASK 1:

Implementing Bubble Sort with AI Comments

- Task: Write a Python implementation of Bubble Sort.
- Instructions:

o Students implement Bubble Sort normally.

o Ask AI to generate inline comments explaining key logic (like swapping, passes, and termination).

o Request AI to provide time complexity analysis.

PROMPT:

Generate python code and implementing Bubble Sort with AI Comments

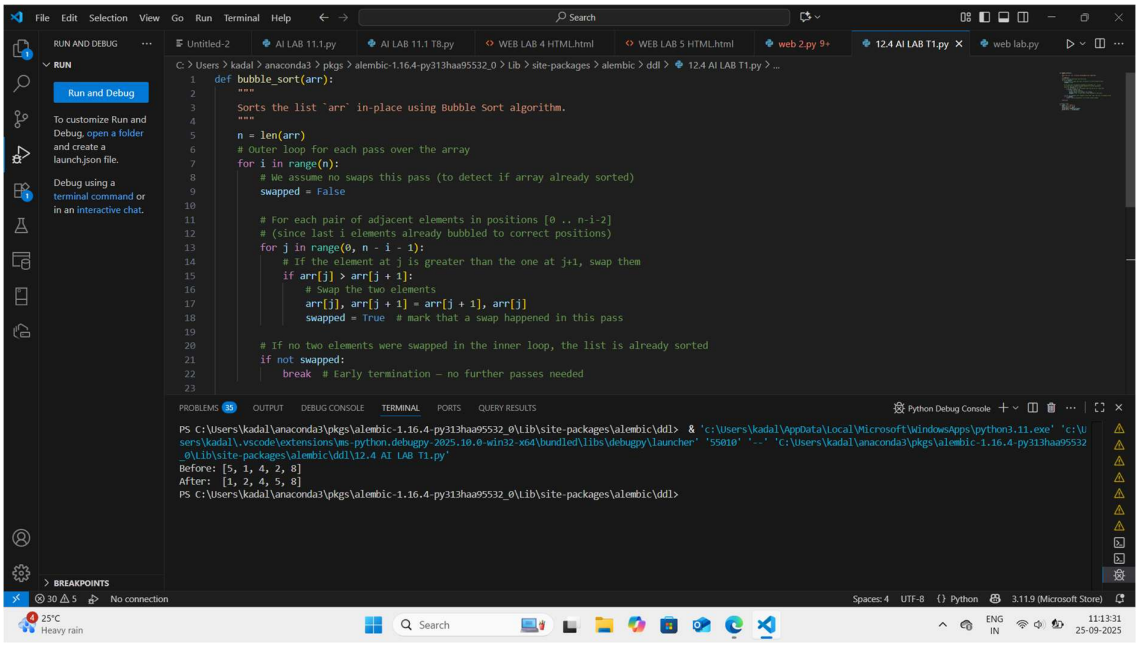
- Task: Write a Python implementation of Bubble Sort.
- Instructions:

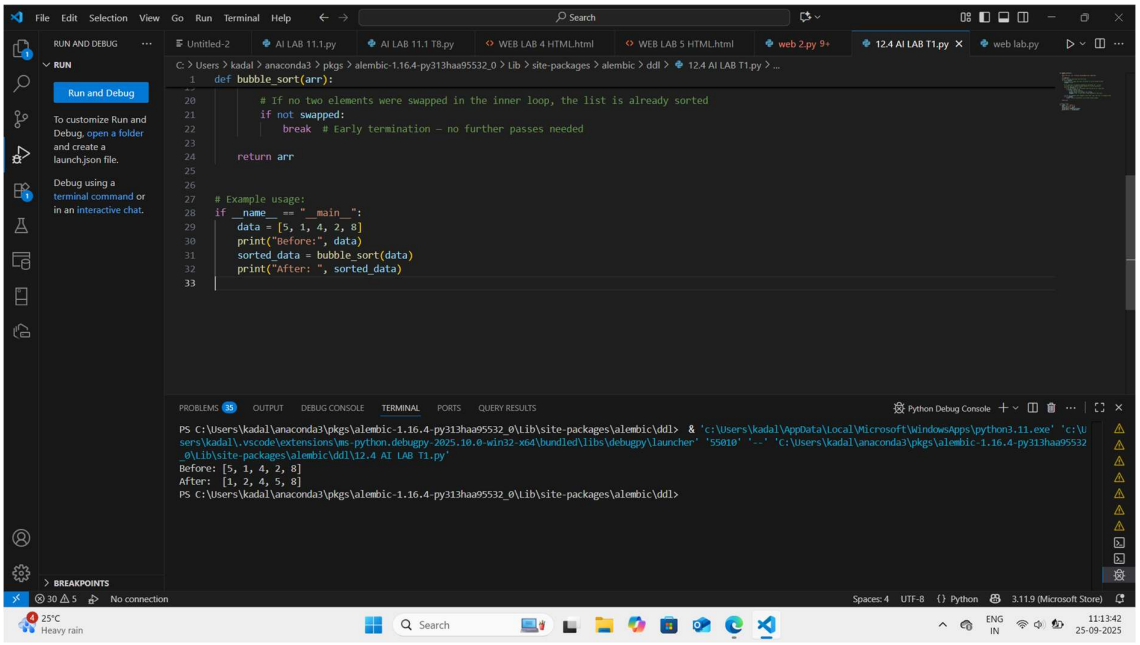
o Students implement Bubble Sort normally.

o Ask AI to generate inline comments explaining key

logic (like swapping, passes, and termination).
o Request AI to provide time complexity analysis.

CODE & OUTPUT:





EXPLAINATION:

- **Outer loop** (`for i in range(n)`): Each pass moves the next largest (or smallest, depending on sort direction) element into its correct position at the end. After `i` passes, the last `i` elements are already sorted and in place.
- **Inner loop** (`for j in range(0, n - i - 1)`): Compares adjacent pairs `arr[j]` and `arr[j + 1]`, and swaps if they are out of order. We don't go all the way to the end every pass because the tail end is already sorted.
- **Swapping logic**: `arr[j], arr[j + 1] = arr[j + 1], arr[j]` performs the exchange in one line.
- **Early termination check** (`if not swapped: break`): If in a full pass we made no swaps, the list is already sorted — so we break to avoid unnecessary passes (optimization).

Time Complexity Analysis

- **Worst-case time complexity**:
  In the worst case (the input is in reverse order), the algorithm will perform all possible comparisons and swaps in every pass. That leads to

$$\sum_{i=0}^{n-1}(n-i-1) = \frac{n(n-1)}{2} = O(n^2)$$

  So **O(n²)**.
- **Best-case time complexity**:
  If the array is already sorted, then in the first pass no swaps will be made, so `swapped` remains `False` and we break out early. The inner loop runs once with ~n comparisons, so it is **O(n)** in the best case (with the early termination optimization).

- **Best-case time complexity**:
  If the array is already sorted, then in the first pass no swaps will be made, so `swapped` remains `False` and we break out early. The inner loop runs once with ~n comparisons, so it is **O(n)** in the best case (with the early termination optimization).

- **Average-case time complexity**:
  Generally also **O(n²)**.
- **Space complexity**:
  Bubble Sort is in-place (only a constant amount of extra memory for indices and flag), so **O(1)** additional space.

TASK 2:

Optimizing Bubble Sort → Insertion Sort

• Task: Provide Bubble Sort code to AI and ask it to suggest a

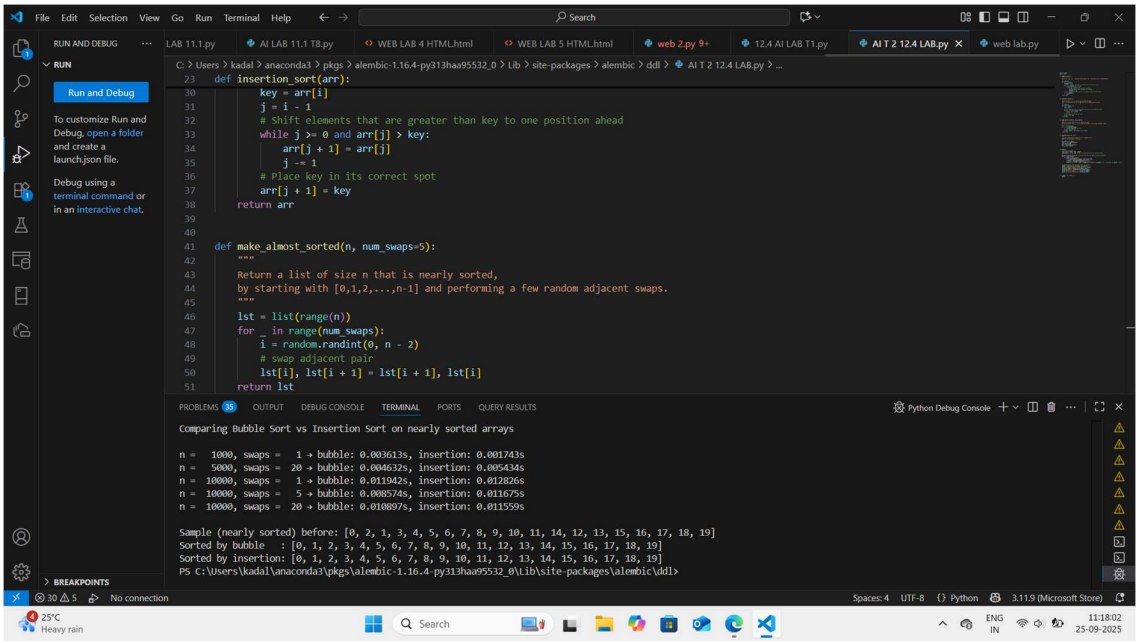
more efficient algorithm for partially sorted arrays.

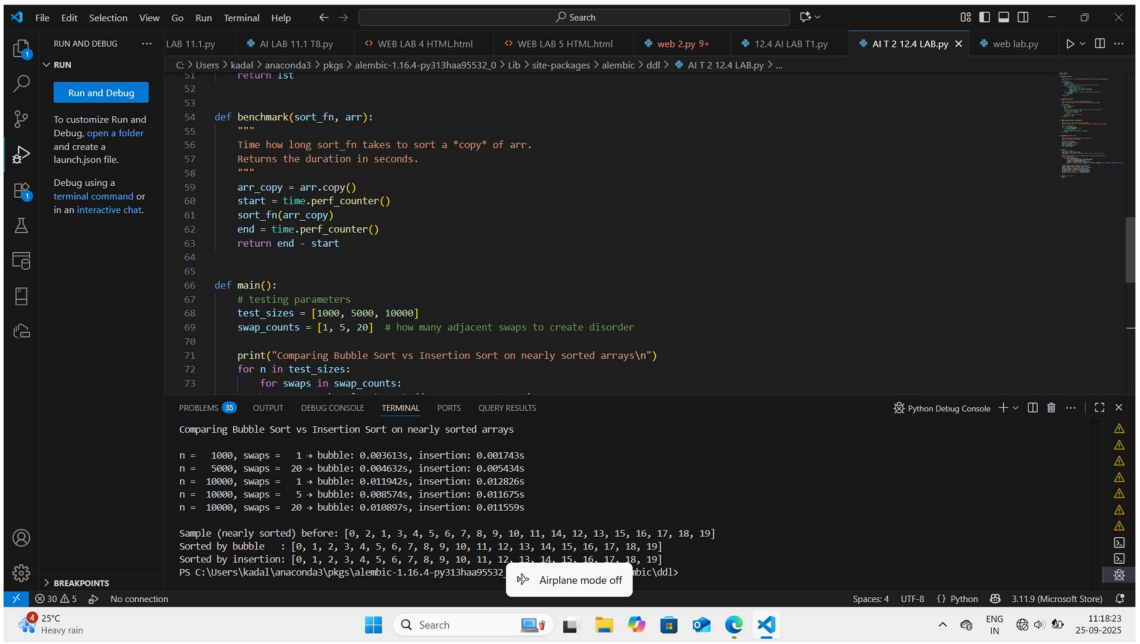
• Instructions:
o Students implement Bubble Sort first.
o Ask AI to suggest an alternative (Insertion Sort).
o Compare performance on nearly sorted input.

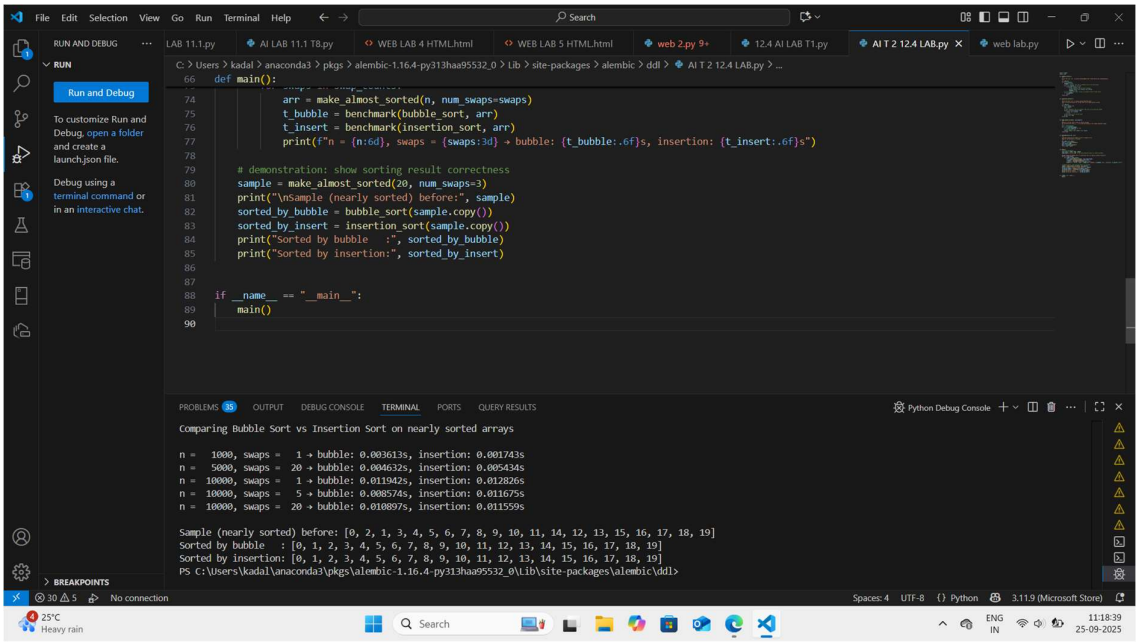
PROMPT:

Generate python code and optimizing Bubble Sort → Insertion Sort
• Task: Provide Bubble Sort code to AI and ask it to suggest a
more efficient algorithm for partially sorted arrays.
• Instructions:
o Students implement Bubble Sort first.
o Ask AI to suggest an alternative (Insertion Sort).
o Compare performance on nearly sorted input.

CODE & OUTPUT:


def insertion_sort(arr):
key = arr[i]
j = i - 1
# Shift elements that are greater than key to one position ahead
while j >= 0 and arr[j] > key:
arr[j + 1] = arr[j]
j -= 1
Place key in its correct spot
arr[j + 1] = key
return arr
def make_almost_sorted(n, num_swaps=5):
"""
Return a list of size n that is nearly sorted,
by starting with [0,1,2,...,n-1] and performing a few random adjacent swaps.
"""
lst = list(range(n))
for _ in range(num_swaps):
i = random.randint(0, n - 2)
# swap adjacent pair
lst[i], lst[i + 1] = lst[i + 1], lst[i]
return lst
Comparing Bubble Sort vs Insertion Sort on nearly sorted arrays
n = 1000, swaps = 1 → bubble: 0.003613s, insertion: 0.001743s
n = 5000, swaps = 20 → bubble: 0.004632s, insertion: 0.005434s
n = 10000, swaps = 1 → bubble: 0.011942s, insertion: 0.012826s
n = 10000, swaps = 5 → bubble: 0.008574s, insertion: 0.011675s
n = 10000, swaps = 20 → bubble: 0.010897s, insertion: 0.011559s
Sample (nearly sorted) before: [0, 2, 1, 3, 4, 5, 6, 7, 8, 9, 10, 11, 14, 12, 13, 15, 16, 17, 18, 19]
Sorted by bubble : [0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19]
Sorted by insertion: [0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19]
PS C:\Users\kadal\anaconda3\pkgs\alembic-1.16.4-py313haa95532_0\Lib\site-packages\alembic\ddl>


def benchmark(sort_fn, arr):
"""
Time how long sort_fn takes to sort a *copy* of arr.
Returns the duration in seconds.
"""
arr_copy = arr.copy()
start = time.perf_counter()
sort_fn(arr_copy)
end = time.perf_counter()
return end - start
def main():
# testing parameters
test_sizes = [1000, 5000, 10000]
swap_counts = [1, 5, 20] # how many adjacent swaps to create disorder
print("Comparing Bubble Sort vs Insertion Sort on nearly sorted arrays\n")
for n in test_sizes:
for swaps in swap_counts:
Comparing Bubble Sort vs Insertion Sort on nearly sorted arrays
n = 1000, swaps = 1 → bubble: 0.003613s, insertion: 0.001743s
n = 5000, swaps = 20 → bubble: 0.004632s, insertion: 0.005434s
n = 10000, swaps = 1 → bubble: 0.011942s, insertion: 0.012826s
n = 10000, swaps = 5 → bubble: 0.008574s, insertion: 0.011675s
n = 10000, swaps = 20 → bubble: 0.010897s, insertion: 0.011559s
Sample (nearly sorted) before: [0, 2, 1, 3, 4, 5, 6, 7, 8, 9, 10, 11, 14, 12, 13, 15, 16, 17, 18, 19]
Sorted by bubble : [0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19]
Sorted by insertion: [0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19]
PS C:\Users\kadal\anaconda3\pkgs\alembic-1.16.4-py313haa95532
Airplane mode off



EXPLAINATION:

- **Bubble Sort** is implemented with the usual double loop and an early-exit `swapped` flag to stop if no swaps occur in a pass.
- **Insertion Sort** is used as the alternative, especially effective when the array is already "almost sorted."
- `make_almost_sorted(n, num_swaps)` creates test data that is nearly sorted by applying a few adjacent swaps to a sorted list of size `n`.
- In `main()`, we run benchmarks for various sizes and disorder levels, and print timings that show where insertion sort overtakes bubble sort.

You can run this script and observe that:

- For large `n` and **very few swaps**, **insertion sort** tends to be *much faster* than bubble sort.
- For more disorder (lots of swaps), both may degrade toward $O(n^2)$ behavior, but insertion sort's constant factors are often better in practice for partially sorted data.

If you like, I can also generate a **detailed log / step-by-step** version (printing each swap or pass) for teaching purposes, or visualize the time complexity difference (e.g. a plot). Do you want me to produce that?

TASK 3:

Binary Search vs Linear Search

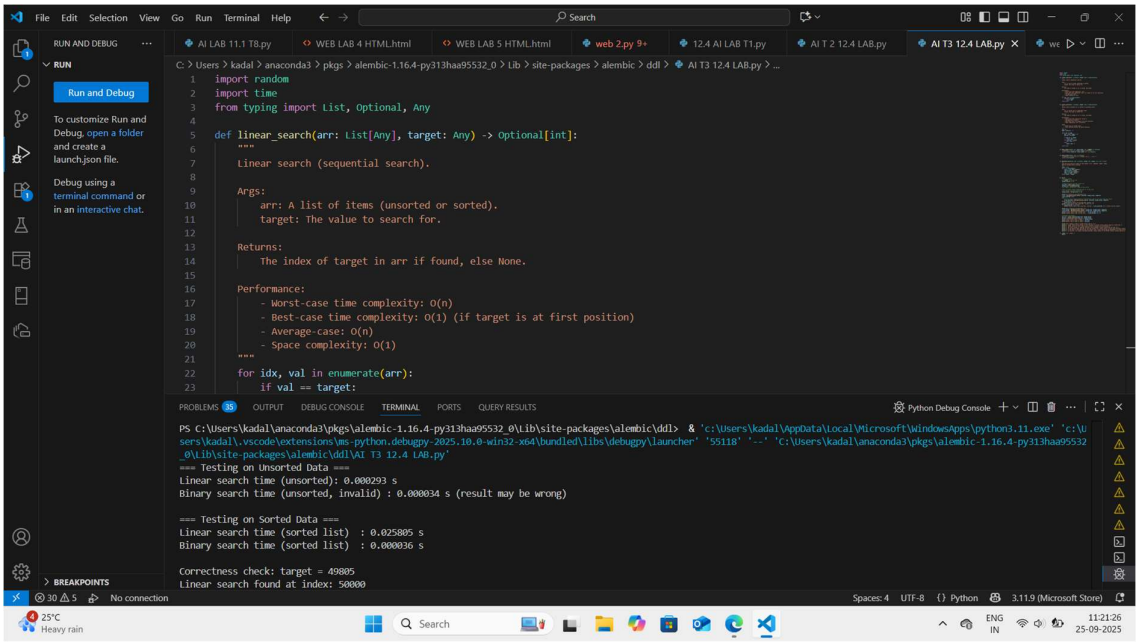
• Task: Implement both Linear Search and Binary Search.

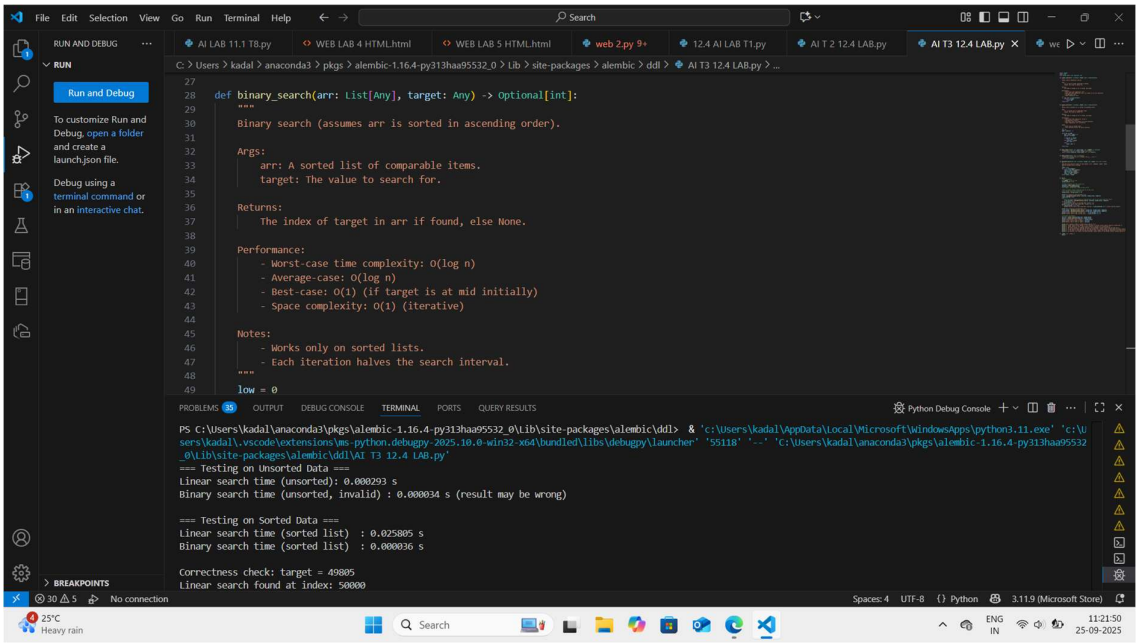
• Instructions:
o Use AI to generate docstrings and performance notes.
o Test both algorithms on sorted and unsorted data.
o Ask AI to explain when Binary Search is preferable.

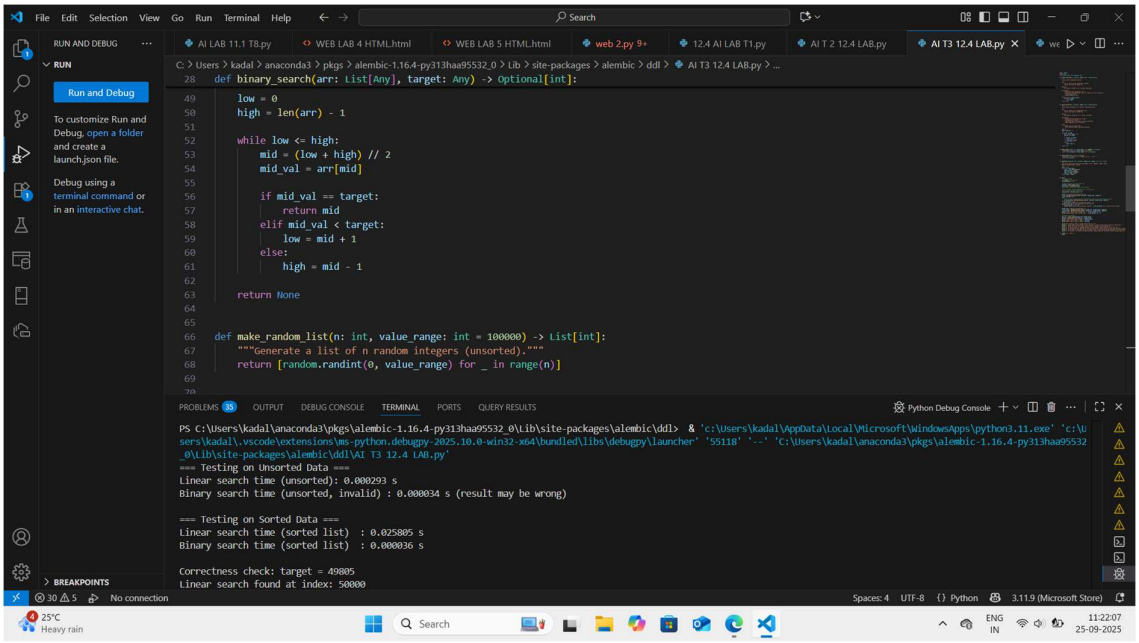
PROMPT:

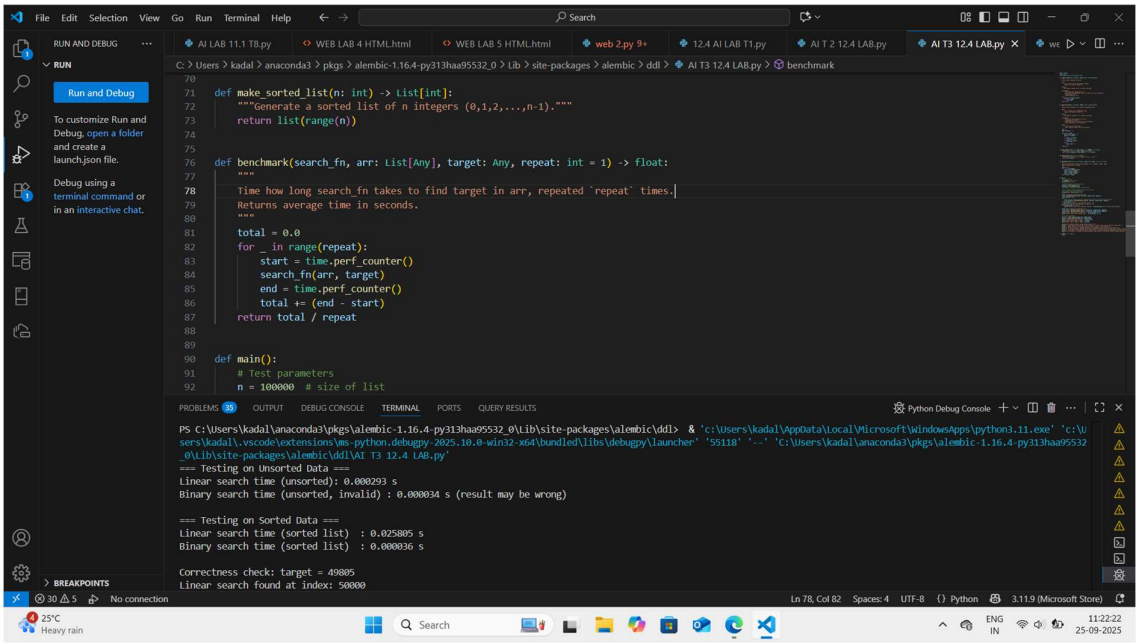
Generate python code and binary Search vs Linear Search
• Task: Implement both Linear Search and Binary Search.
• Instructions:
o Use AI to generate docstrings and performance notes.
o Test both algorithms on sorted and unsorted data.
o Ask AI to explain when Binary Search is preferable.

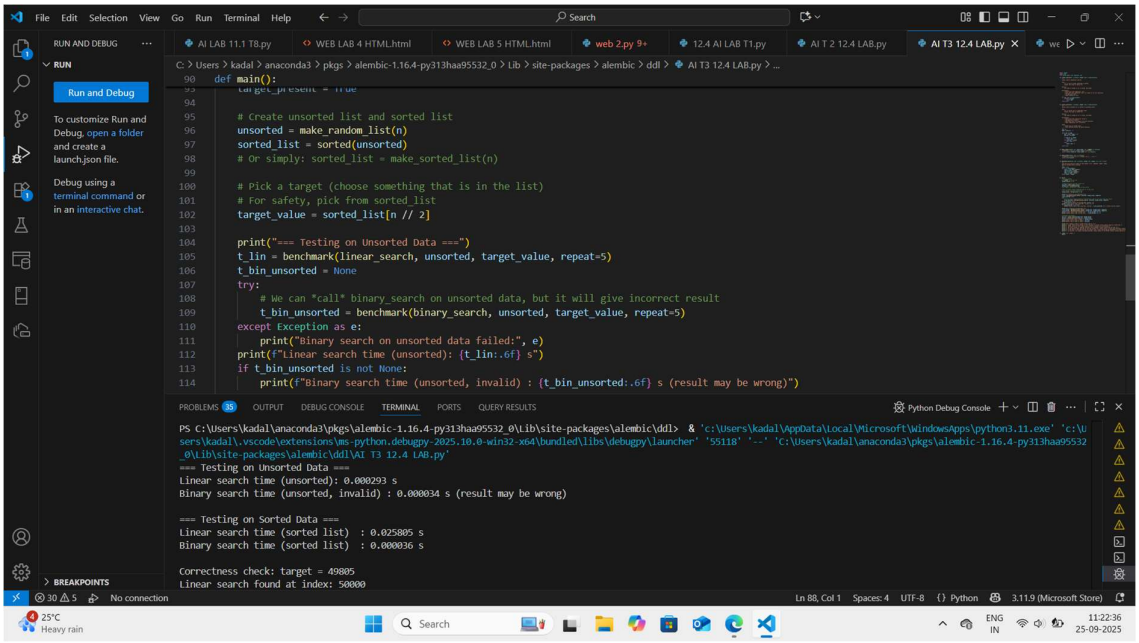
CODE & OUTPUT:

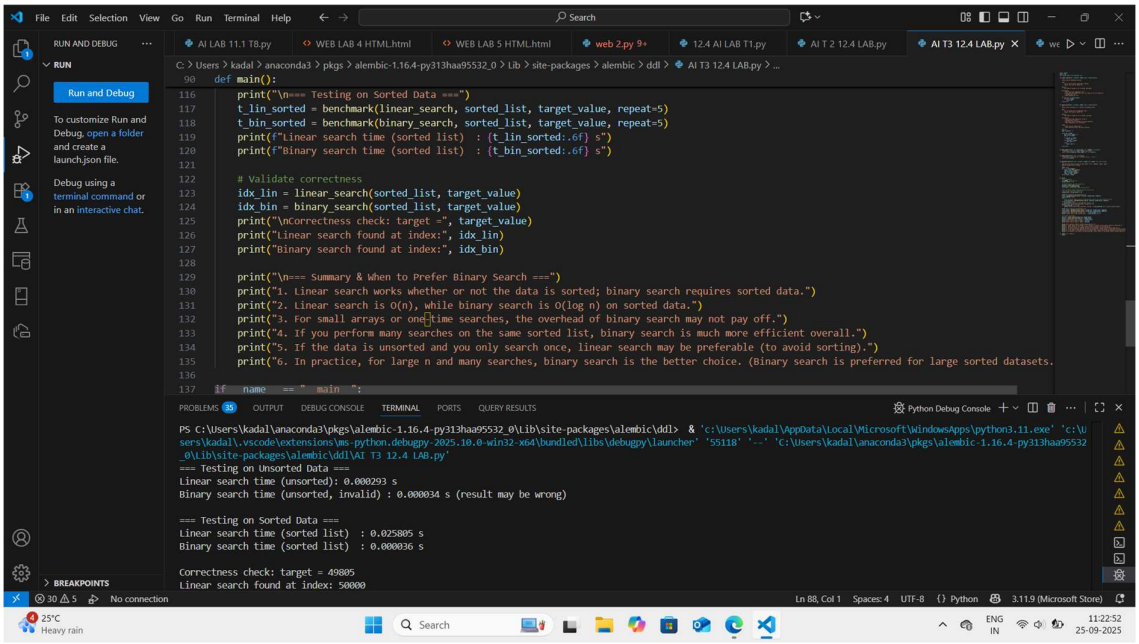

File Edit Selection View Go Run Terminal Help
Search
RUN AND DEBUG
RUN
Run and Debug
To customize Run and Debug, open a folder and create a launch.json file.
Debug using a terminal command or in an interactive chat.
AI LAB 11.1 T8.py
WEB LAB 4 HTML.html
WEB LAB 5 HTML.html
web 2.py 9+
12.4 AI LAB T1.py
AI T 2 12.4 LAB.py
AI T3 12.4 LAB.py
C: > Users > kadal > anaconda3 > pkgs > alembic-1.16.4-py313haa95532_0 > Lib > site-packages > alembic > ddl > AI T3 12.4 LAB.py > ...
import random
import time
from typing import List, Optional, Any
def linear_search(arr: List[Any], target: Any) -> Optional[int]:
"""
Linear search (sequential search).
Args:
arr: A list of items (unsorted or sorted).
target: The value to search for.
Returns:
The index of target in arr if found, else None.
Performance:
- Worst-case time complexity: O(n)
- Best-case time complexity: O(1) (if target is at first position)
- Average-case: O(n)
- Space complexity: O(1)
"""
for idx, val in enumerate(arr):
if val == target:
PROBLEMS 35
OUTPUT
DEBUG CONSOLE
TERMINAL
PORTS
QUERY RESULTS
Python Debug Console
PS C:\Users\kadal\anaconda3\pkgs\alembic-1.16.4-py313haa95532_0\Lib\site-packages\alembic\ddl> & 'c:\Users\kadal\AppData\Local\Microsoft\WindowsApps\python3.11.exe' 'c:\Users\kadal\.vscode\extensions\ms-python.debugpy-2025.10.0-win32-x64\bundled\libs\debugpy\launcher' '55118' '--' 'C:\Users\kadal\anaconda3\pkgs\alembic-1.16.4-py313haa95532_0\Lib\site-packages\alembic\ddl\AI T3 12.4 LAB.py'
=== Testing on Unsorted Data ===
Linear search time (unsorted): 0.000293 s
Binary search time (unsorted, invalid) : 0.000034 s (result may be wrong)
=== Testing on Sorted Data ===
Linear search time (sorted list) : 0.025805 s
Binary search time (sorted list) : 0.000036 s
Correctness check: target = 49805
Linear search found at index: 50000
BREAKPOINTS
30 5
No connection
Spaces: 4
UTF-8
Python
3.11.9 (Microsoft Store)
25°C
Heavy rain
Search
ENG
IN
11:21:26
25-09-2025


File Edit Selection View Go Run Terminal Help
Search
RUN AND DEBUG
RUN
Run and Debug
To customize Run and Debug, open a folder and create a launch.json file.
Debug using a terminal command or in an interactive chat.
AI LAB 11.1 T8.py
WEB LAB 4 HTML.html
WEB LAB 5 HTML.html
web 2.py 9+
12.4 AI LAB T1.py
AI T 2 12.4 LAB.py
AI T3 12.4 LAB.py
C: > Users > kadal > anaconda3 > pkgs > alembic-1.16.4-py313haa95532_0 > Lib > site-packages > alembic > ddl > AI T3 12.4 LAB.py > ...
def binary_search(arr: List[Any], target: Any) -> Optional[int]:
"""
Binary search (assumes arr is sorted in ascending order).
Args:
arr: A sorted list of comparable items.
target: The value to search for.
Returns:
The index of target in arr if found, else None.
Performance:
- Worst-case time complexity: O(log n)
- Average-case: O(log n)
- Best-case: O(1) (if target is at mid initially)
- Space complexity: O(1) (iterative)
Notes:
- Works only on sorted lists.
- Each iteration halves the search interval.
"""
low = 0
PROBLEMS 35
OUTPUT
DEBUG CONSOLE
TERMINAL
PORTS
QUERY RESULTS
Python Debug Console
PS C:\Users\kadal\anaconda3\pkgs\alembic-1.16.4-py313haa95532_0\Lib\site-packages\alembic\ddl> & 'c:\Users\kadal\AppData\Local\Microsoft\WindowsApps\python3.11.exe' 'c:\Users\kadal\.vscode\extensions\ms-python.debugpy-2025.10.0-win32-x64\bundled\libs\debugpy\launcher' '55118' '--' 'C:\Users\kadal\anaconda3\pkgs\alembic-1.16.4-py313haa95532_0\Lib\site-packages\alembic\ddl\AI T3 12.4 LAB.py'
=== Testing on Unsorted Data ===
Linear search time (unsorted): 0.000293 s
Binary search time (unsorted, invalid) : 0.000034 s (result may be wrong)
=== Testing on Sorted Data ===
Linear search time (sorted list) : 0.025805 s
Binary search time (sorted list) : 0.000036 s
Correctness check: target = 49805
Linear search found at index: 50000
BREAKPOINTS
30 5
No connection
Spaces: 4
UTF-8
Python
3.11.9 (Microsoft Store)
25°C
Heavy rain
Search
ENG
IN
11:21:50
25-09-2025


File Edit Selection View Go Run Terminal Help
Search
RUN AND DEBUG
RUN
Run and Debug
To customize Run and Debug, open a folder and create a launch.json file.
Debug using a terminal command or in an interactive chat.
AI LAB 11.1 T8.py
WEB LAB 4 HTML.html
WEB LAB 5 HTML.html
web 2.py 9+
12.4 AI LAB T1.py
AI T 2 12.4 LAB.py
AI T3 12.4 LAB.py
C: > Users > kadal > anaconda3 > pkgs > alembic-1.16.4-py313haa95532_0 > Lib > site-packages > alembic > ddl > AI T3 12.4 LAB.py > ...
def binary_search(arr: List[Any], target: Any) -> Optional[int]:
low = 0
high = len(arr) - 1
while low <= high:
mid = (low + high) // 2
mid_val = arr[mid]
if mid_val == target:
return mid
elif mid_val < target:
low = mid + 1
else:
high = mid - 1
return None
def make_random_list(n: int, value_range: int = 100000) -> List[int]:
"""Generate a list of n random integers (unsorted)."""
return [random.randint(0, value_range) for _ in range(n)]
PROBLEMS 35
OUTPUT
DEBUG CONSOLE
TERMINAL
PORTS
QUERY RESULTS
Python Debug Console
PS C:\Users\kadal\anaconda3\pkgs\alembic-1.16.4-py313haa95532_0\Lib\site-packages\alembic\ddl> & 'c:\Users\kadal\AppData\Local\Microsoft\WindowsApps\python3.11.exe' 'c:\Users\kadal\.vscode\extensions\ms-python.debugpy-2025.10.0-win32-x64\bundled\libs\debugpy\launcher' '55118' '--' 'C:\Users\kadal\anaconda3\pkgs\alembic-1.16.4-py313haa95532_0\Lib\site-packages\alembic\ddl\AI T3 12.4 LAB.py'
=== Testing on Unsorted Data ===
Linear search time (unsorted): 0.000293 s
Binary search time (unsorted, invalid) : 0.000034 s (result may be wrong)
=== Testing on Sorted Data ===
Linear search time (sorted list) : 0.025805 s
Binary search time (sorted list) : 0.000036 s
Correctness check: target = 49805
Linear search found at index: 50000
BREAKPOINTS
30 5
No connection
Spaces: 4
UTF-8
Python
3.11.9 (Microsoft Store)
25°C
Heavy rain
Search
ENG
IN
11:22:07
25-09-2025


File Edit Selection View Go Run Terminal Help
Search
RUN AND DEBUG
RUN
Run and Debug
To customize Run and Debug, open a folder and create a launch.json file.
Debug using a terminal command or in an interactive chat.
AI LAB 11.1 T8.py
WEB LAB 4 HTML.html
WEB LAB 5 HTML.html
web 2.py 9+
12.4 AI LAB T1.py
AI T 2 12.4 LAB.py
AI T3 12.4 LAB.py
C: > Users > kadal > anaconda3 > pkgs > alembic-1.16.4-py313haa95532_0 > Lib > site-packages > alembic > ddl > AI T3 12.4 LAB.py > benchmark
def make_sorted_list(n: int) -> List[int]:
"""Generate a sorted list of n integers (0,1,2,...,n-1)."""
return list(range(n))
def benchmark(search_fn, arr: List[Any], target: Any, repeat: int = 1) -> float:
"""
Time how long search_fn takes to find target in arr, repeated `repeat` times.
Returns average time in seconds.
"""
total = 0.0
for _ in range(repeat):
start = time.perf_counter()
search_fn(arr, target)
end = time.perf_counter()
total += (end - start)
return total / repeat
def main():
# Test parameters
n = 100000 # size of list
PROBLEMS 35
OUTPUT
DEBUG CONSOLE
TERMINAL
PORTS
QUERY RESULTS
Python Debug Console
PS C:\Users\kadal\anaconda3\pkgs\alembic-1.16.4-py313haa95532_0\Lib\site-packages\alembic\ddl> & 'c:\Users\kadal\AppData\Local\Microsoft\WindowsApps\python3.11.exe' 'c:\Users\kadal\.vscode\extensions\ms-python.debugpy-2025.10.0-win32-x64\bundled\libs\debugpy\launcher' '55118' '--' 'C:\Users\kadal\anaconda3\pkgs\alembic-1.16.4-py313haa95532_0\Lib\site-packages\alembic\ddl\AI T3 12.4 LAB.py'
=== Testing on Unsorted Data ===
Linear search time (unsorted): 0.000293 s
Binary search time (unsorted, invalid) : 0.000034 s (result may be wrong)
=== Testing on Sorted Data ===
Linear search time (sorted list) : 0.025805 s
Binary search time (sorted list) : 0.000036 s
Correctness check: target = 49805
Linear search found at index: 50000
BREAKPOINTS
30 5
No connection
Ln 78, Col 82
Spaces: 4
UTF-8
Python
3.11.9 (Microsoft Store)
25°C
Heavy rain
Search
ENG
IN
11:22:22
25-09-2025






```
Correctness check: target = 49805
Linear search found at index: 50000
Binary search found at index: 50000

=== Summary & When to Prefer Binary Search ===
1. Linear search works whether or not the data is sorted; binary search requires sorted data.
2. Linear search is O(n), while binary search is O(log n) on sorted data.
3. For small arrays or one-time searches, the overhead of binary search may not pay off.
4. If you perform many searches on the same sorted list, binary search is much more efficient overall.
5. If the data is unsorted and you only search once, linear search may be preferable (to avoid sorting).
6. In practice, for large n and many searches, binary search is the better choice. (Binary search is preferred for large sorted datasets.)
```

EXPLAINATION:

Linear Search is simple and works on **any** list (sorted or unsorted).

- It scans each element until it finds the target or reaches the end.
- Time complexity is **O(n)** in the worst and average cases. Wikipedia
- It has very low overhead, which may make it competitive for small arrays or simple uses.

Binary Search is much faster **if** the array is sorted.

- It repeatedly divides the search space in half. Wikipedia
- Time complexity is **O(log n)** in worst and average cases. Wikipedia +1
- But it **requires** the data to be sorted first.
- If you have to sort the list just to use binary search once, sorting time (O(n log n)) may dominate.

When Binary Search is preferable:

1. Your data is already sorted (or you can afford keeping it sorted).
2. You will perform **many** searches on the same dataset (then the one-time sorting cost is amortized).
3. The dataset is large, so the logarithmic reduction in comparisons is meaningful.
4. For small arrays or one-off searches, linear search's simplicity and minimal overhead may beat binary search in practice. (Some benchmarks show that for small n, linear search can be faster because binary search has more branch logic, etc.) dirtyhandscodin... +1

TASK 4:

Quick Sort and Merge Sort Comparison

• Task: Implement Quick Sort and Merge Sort using recursion.

• Instructions:

o Provide AI with partially completed functions for recursion.

o Ask AI to complete the missing logic and add docstrings.

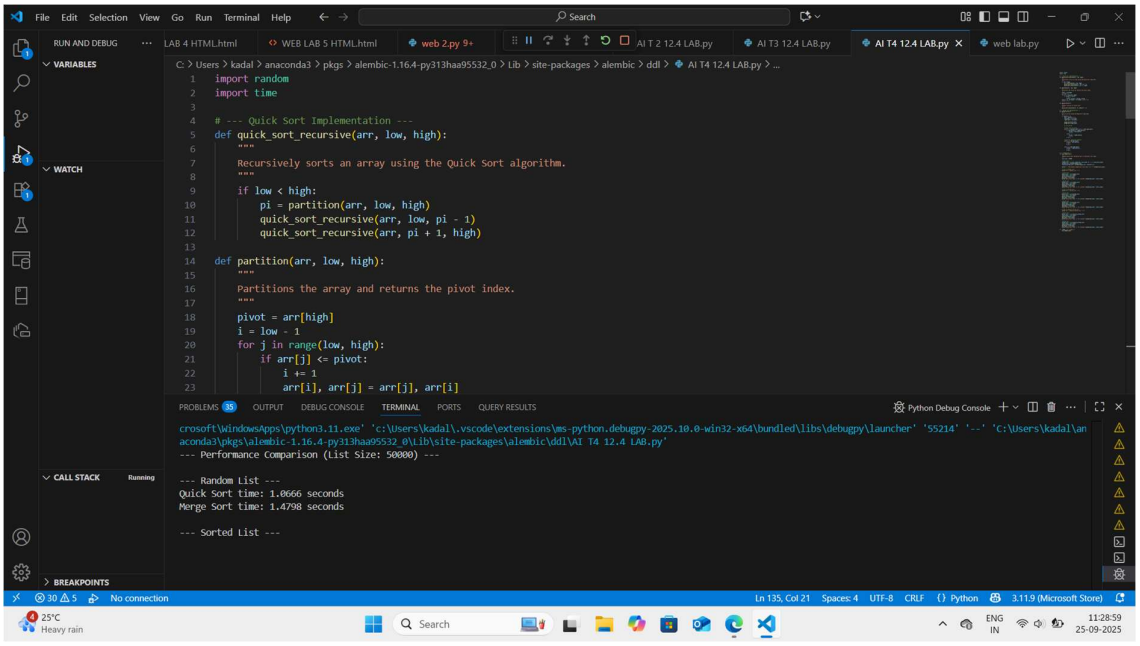
o Compare both algorithms on random, sorted, and reverse-
sorted lists.

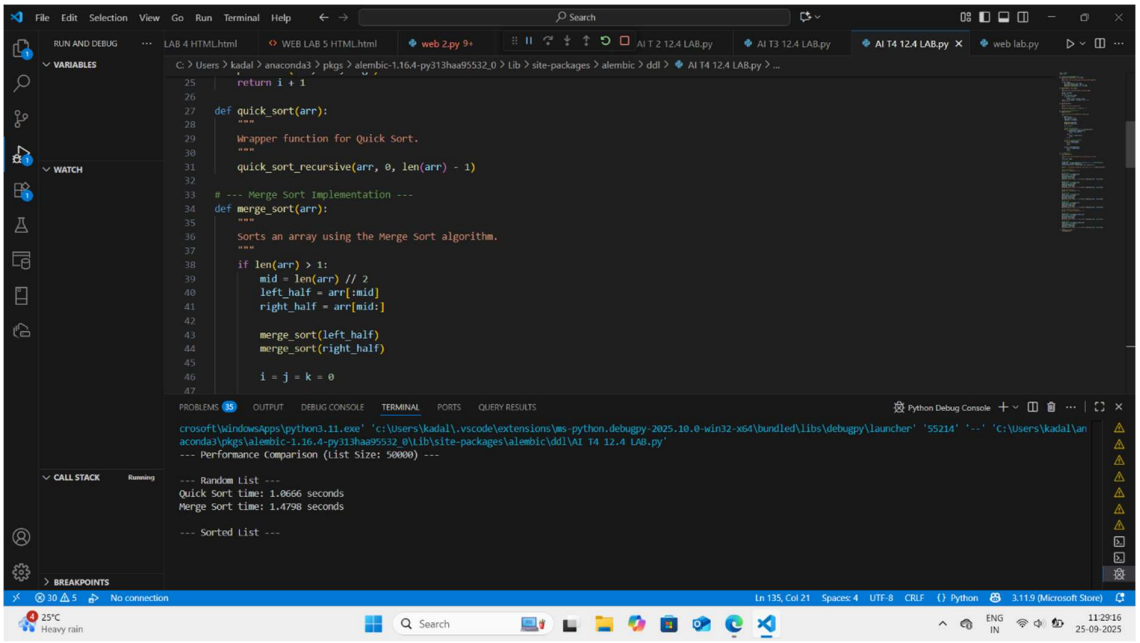
PROMPT:

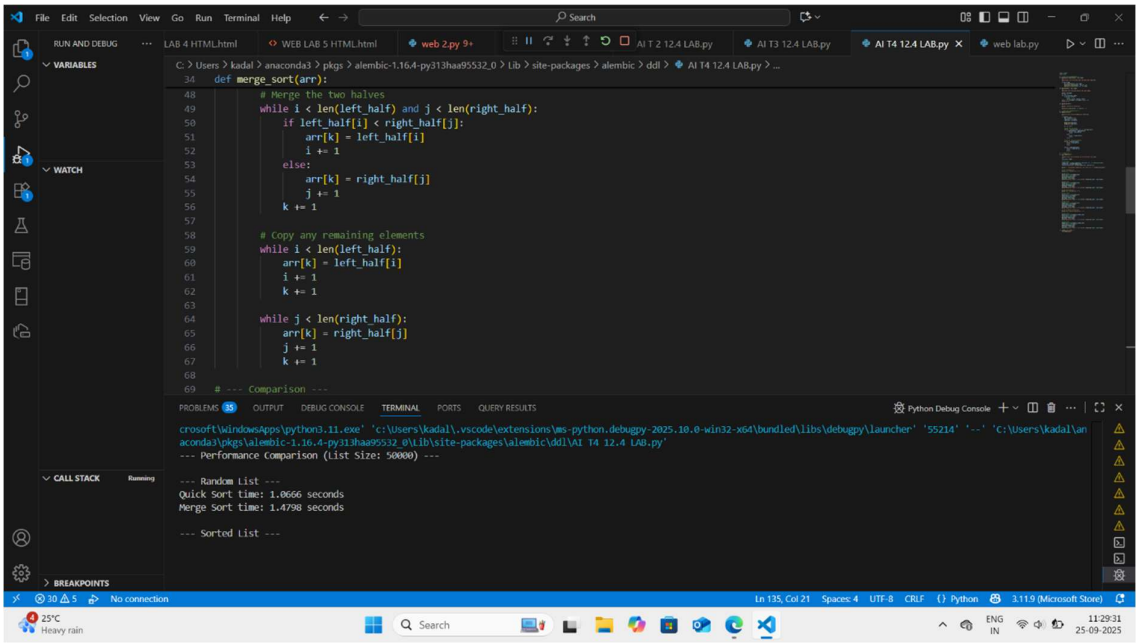
Generate python code and quick Sort and Merge Sort Comparison

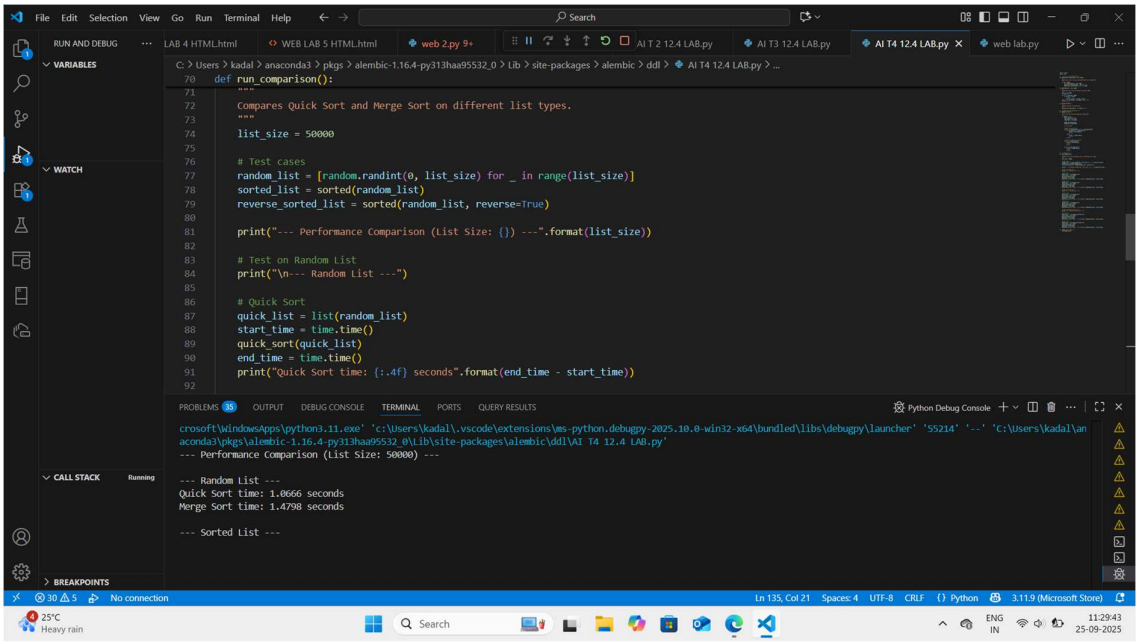
• Task: Implement Quick Sort and Merge Sort using recursion.
• Instructions:
o Provide AI with partially completed functions for recursion.
o Ask AI to complete the missing logic and add docstrings.
o Compare both algorithms on random, sorted, and reverse-
sorted lists.

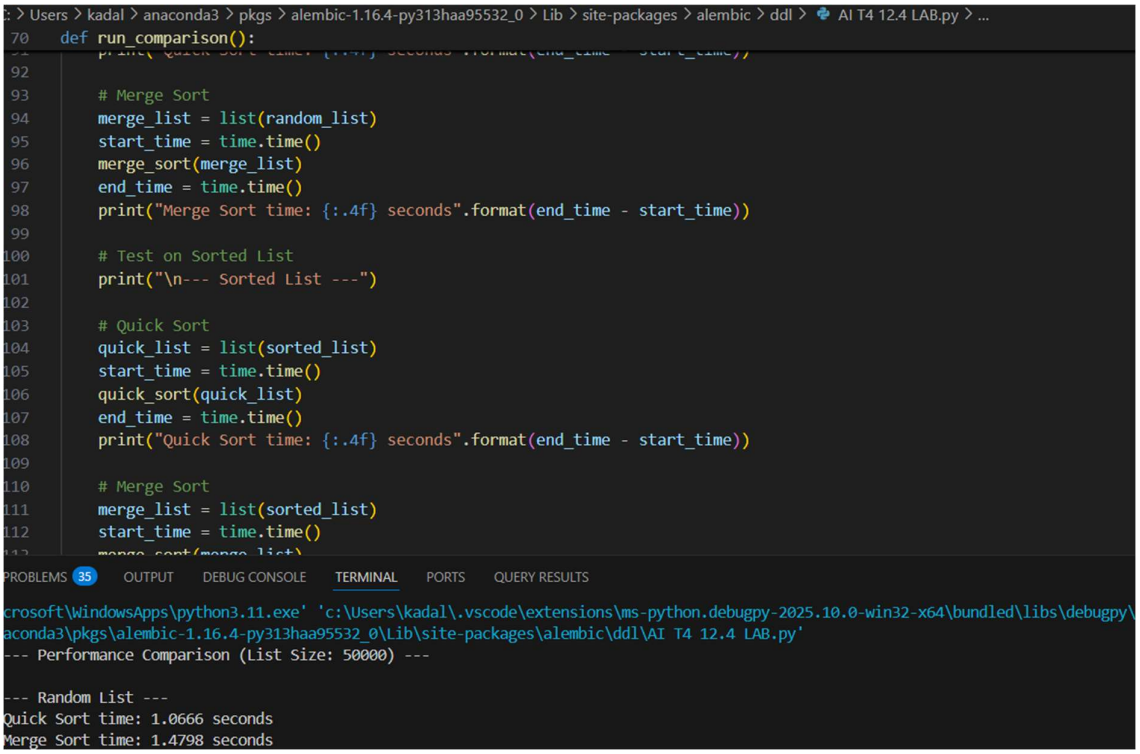
CODE & OUTPUT:




File Edit Selection View Go Run Terminal Help
Search
RUN AND DEBUG
LAB 4 HTML.html
WEB LAB 5 HTML.html
web 2.py 9+
AI T 2 12.4 LAB.py
AI T3 12.4 LAB.py
AI T4 12.4 LAB.py
web lab.py
VARIABLES
WATCH
C: > Users > kadal > anaconda3 > pkgs > alembic-1.16.4-py313haa95532_0 > Lib > site-packages > alembic > ddl > AI T4 12.4 LAB.py > ...
25 return i + 1
27 def quick_sort(arr):
28 """
29 Wrapper function for Quick Sort.
30 """
31 quick_sort_recursive(arr, 0, len(arr) - 1)
33 # --- Merge Sort Implementation ---
34 def merge_sort(arr):
35 """
36 Sorts an array using the Merge Sort algorithm.
37 """
38 if len(arr) > 1:
39 mid = len(arr) // 2
40 left_half = arr[:mid]
41 right_half = arr[mid:]
43 merge_sort(left_half)
44 merge_sort(right_half)
46 i = j = k = 0
PROBLEMS 35
OUTPUT
DEBUG CONSOLE
TERMINAL
PORTS
QUERY RESULTS
Python Debug Console
crosoft\WindowsApps\python3.11.exe' 'c:\Users\kadal\.vscode\extensions\ms-python.debugpy-2025.10.0-win32-x64\bundled\libs\debugpy\launcher' '55214' '--' 'C:\Users\kadal\anaconda3\pkgs\alembic-1.16.4-py313haa95532_0\Lib\site-packages\alembic\ddl\AI T4 12.4 LAB.py'
--- Performance Comparison (List Size: 50000) ---
--- Random List ---
Quick Sort time: 1.0666 seconds
Merge Sort time: 1.4798 seconds
--- Sorted List ---
CALL STACK
Running
BREAKPOINTS
30 5
No connection
Ln 135, Col 21
Spaces: 4
UTF-8
CRLF
Python
3.11.9 (Microsoft Store)
25°C
Heavy rain
Search
ENG
IN
11:29:16
25-09-2025


File Edit Selection View Go Run Terminal Help
Search
RUN AND DEBUG
LAB 4 HTML.html
WEB LAB 5 HTML.html
web 2.py 9+
AI T 2 12.4 LAB.py
AI T3 12.4 LAB.py
AI T4 12.4 LAB.py
web lab.py
VARIABLES
WATCH
C: > Users > kadal > anaconda3 > pkgs > alembic-1.16.4-py313haa95532_0 > Lib > site-packages > alembic > ddl > AI T4 12.4 LAB.py > ...
34 def merge_sort(arr):
48 # Merge the two halves
49 while i < len(left_half) and j < len(right_half):
50 if left_half[i] < right_half[j]:
51 arr[k] = left_half[i]
52 i += 1
53 else:
54 arr[k] = right_half[j]
55 j += 1
56 k += 1
58 # Copy any remaining elements
59 while i < len(left_half):
60 arr[k] = left_half[i]
61 i += 1
62 k += 1
64 while j < len(right_half):
65 arr[k] = right_half[j]
66 j += 1
67 k += 1
69 # --- Comparison ---
PROBLEMS 35
OUTPUT
DEBUG CONSOLE
TERMINAL
PORTS
QUERY RESULTS
Python Debug Console
crosoft\WindowsApps\python3.11.exe' 'c:\Users\kadal\.vscode\extensions\ms-python.debugpy-2025.10.0-win32-x64\bundled\libs\debugpy\launcher' '55214' '--' 'C:\Users\kadal\anaconda3\pkgs\alembic-1.16.4-py313haa95532_0\Lib\site-packages\alembic\ddl\AI T4 12.4 LAB.py'
--- Performance Comparison (List Size: 50000) ---
--- Random List ---
Quick Sort time: 1.0666 seconds
Merge Sort time: 1.4798 seconds
--- Sorted List ---
CALL STACK
Running
BREAKPOINTS
30 5
No connection
Ln 135, Col 21
Spaces: 4
UTF-8
CRLF
Python
3.11.9 (Microsoft Store)
25°C
Heavy rain
Search
ENG
IN
11:29:31
25-09-2025


File Edit Selection View Go Run Terminal Help
Search
RUN AND DEBUG
LAB 4 HTML.html
WEB LAB 5 HTML.html
web 2.py 9+
AI T 2 12.4 LAB.py
AI T3 12.4 LAB.py
AI T4 12.4 LAB.py
web lab.py
VARIABLES
WATCH
C: > Users > kadal > anaconda3 > pkgs > alembic-1.16.4-py313haa95532_0 > Lib > site-packages > alembic > ddl > AI T4 12.4 LAB.py > ...
def run_comparison():
"""
Compares Quick Sort and Merge Sort on different list types.
"""
list_size = 50000
# Test cases
random_list = [random.randint(0, list_size) for _ in range(list_size)]
sorted_list = sorted(random_list)
reverse_sorted_list = sorted(random_list, reverse=True)
print("--- Performance Comparison (List Size: {}) ---".format(list_size))
# Test on Random List
print("\n--- Random List ---")
# Quick Sort
quick_list = list(random_list)
start_time = time.time()
quick_sort(quick_list)
end_time = time.time()
print("Quick Sort time: {:.4f} seconds".format(end_time - start_time))
PROBLEMS 35
OUTPUT
DEBUG CONSOLE
TERMINAL
PORTS
QUERY RESULTS
Python Debug Console
crosoft\WindowsApps\python3.11.exe' 'c:\Users\kadal\.vscode\extensions\ms-python.debugpy-2025.10.0-win32-x64\bundled\libs\debugpy\launcher' '55214' '--' 'C:\Users\kadal\anaconda3\pkgs\alembic-1.16.4-py313haa95532_0\Lib\site-packages\alembic\ddl\AI T4 12.4 LAB.py'
--- Performance Comparison (List Size: 50000) ---
--- Random List ---
Quick Sort time: 1.0666 seconds
Merge Sort time: 1.4798 seconds
--- Sorted List ---
CALL STACK
Running
BREAKPOINTS
No connection
Ln 135, Col 21
Spaces: 4
UTF-8
CRLF
Python
3.11.9 (Microsoft Store)
25°C
Heavy rain
Search
ENG
IN
11:29:43
25-09-2025


: > Users > kadal > anaconda3 > pkgs > alembic-1.16.4-py313haa95532_0 > Lib > site-packages > alembic > ddl > AI T4 12.4 LAB.py > ...
def run_comparison():
# Merge Sort
merge_list = list(random_list)
start_time = time.time()
merge_sort(merge_list)
end_time = time.time()
print("Merge Sort time: {:.4f} seconds".format(end_time - start_time))
# Test on Sorted List
print("\n--- Sorted List ---")
# Quick Sort
quick_list = list(sorted_list)
start_time = time.time()
quick_sort(quick_list)
end_time = time.time()
print("Quick Sort time: {:.4f} seconds".format(end_time - start_time))
# Merge Sort
merge_list = list(sorted_list)
start_time = time.time()
PROBLEMS 35
OUTPUT
DEBUG CONSOLE
TERMINAL
PORTS
QUERY RESULTS
crosoft\WindowsApps\python3.11.exe' 'c:\Users\kadal\.vscode\extensions\ms-python.debugpy-2025.10.0-win32-x64\bundled\libs\debugpy\
aconda3\pkgs\alembic-1.16.4-py313haa95532_0\Lib\site-packages\alembic\ddl\AI T4 12.4 LAB.py'
--- Performance Comparison (List Size: 50000) ---
--- Random List ---
Quick Sort time: 1.0666 seconds
Merge Sort time: 1.4798 seconds

```python
def run_comparison():
    print("Merge Sort time: {:.4f} seconds".format(end_time - start_time))

    # Test on Reverse Sorted List
    print("\n--- Reverse Sorted List ---")

    # Quick Sort
    quick_list = list(reverse_sorted_list)
    start_time = time.time()
    quick_sort(quick_list)
    end_time = time.time()
    print("Quick Sort time: {:.4f} seconds".format(end_time - start_time))

    # Merge Sort
    merge_list = list(reverse_sorted_list)
    start_time = time.time()
    merge_sort(merge_list)
    end_time = time.time()
    print("Merge Sort time: {:.4f} seconds".format(end_time - start_time))

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

PROBLEMS 35 OUTPUT DEBUG CONSOLE TERMINAL PORTS QUERY RESULTS

```
crosoft\WindowsApps\python3.11.exe' 'c:\Users\kadal\.vscode\extensions\ms-python.debugpy-2025.10.0-win32-x64\bundled\libs\debu
aconda3\pkgs\alembic-1.16.4-py313haa95532_0\Lib\site-packages\alembic\ddl\AI T4 12.4 LAB.py'
--- Performance Comparison (List Size: 50000) ---

--- Random List ---
Quick Sort time: 1.0666 seconds
Merge Sort time: 1.4798 seconds
```

EXPLANATION:

Time Complexity (average / worst / best)

- **Quick Sort**: average-case **O(n log n)**, but worst-case **O(n^2)** (e.g. if pivot choices are poor). Wikipedia +2
- **Merge Sort**: **O(n log n)** in all cases (best, average, worst). Wikipedia +2

Space Complexity

- Quick Sort (this version) uses extra lists for partitioning, so not truly in-place; but an in-place variant (Lomuto / Hoare partition) can be done with O(log n) stack usage. Wikipedia +1
- Merge Sort needs O(n) extra space for merging (the merged list) plus recursion overhead. Wikipedia +1

Stability

- Merge Sort is stable (does not reorder equal elements). Wikipedia +1
- Quick Sort is generally **not stable** (depends on partition scheme). GeeksforGeeks +1

Partition Behavior vs Splitting

- Quick Sort partitions around a pivot; the partitions may be unbalanced, introducing risk of worst-case. GeeksforGeeks +2
- Merge Sort always splits evenly (midpoint), so its recursion depth is predictable. Wikipedia +2

Practical Performance & Use Cases

- Quick Sort often outperforms Merge Sort for in-memory arrays because of lower constant factors, better cache locality, and no large temporary arrays. Stack Overflow +2
- Merge Sort is preferable when you require stability, or when worst-case guarantees matter (e.g. real-time systems).
- Merge Sort is often used in external sorting (on disk) because its merging approach accesses data sequentially. GeeksforGeeks +1
- For linked lists, Merge Sort is often a better choice since you can merge without needing additional arrays and random access is expensive. GeeksforGeeks +1

TASK 5:

AI-Suggested Algorithm Optimization

• Task: Give AI a naive algorithm (e.g., $O(n^2)$ duplicate search).

• Instructions:

o Students write a brute force duplicate-finder.

o Ask AI to optimize it (e.g., by using sets/dictionaries with

O(n) time).

o Compare execution times with large input sizes.

PROMPT:

Generate python code and AI-Suggested Algorithm Optimization

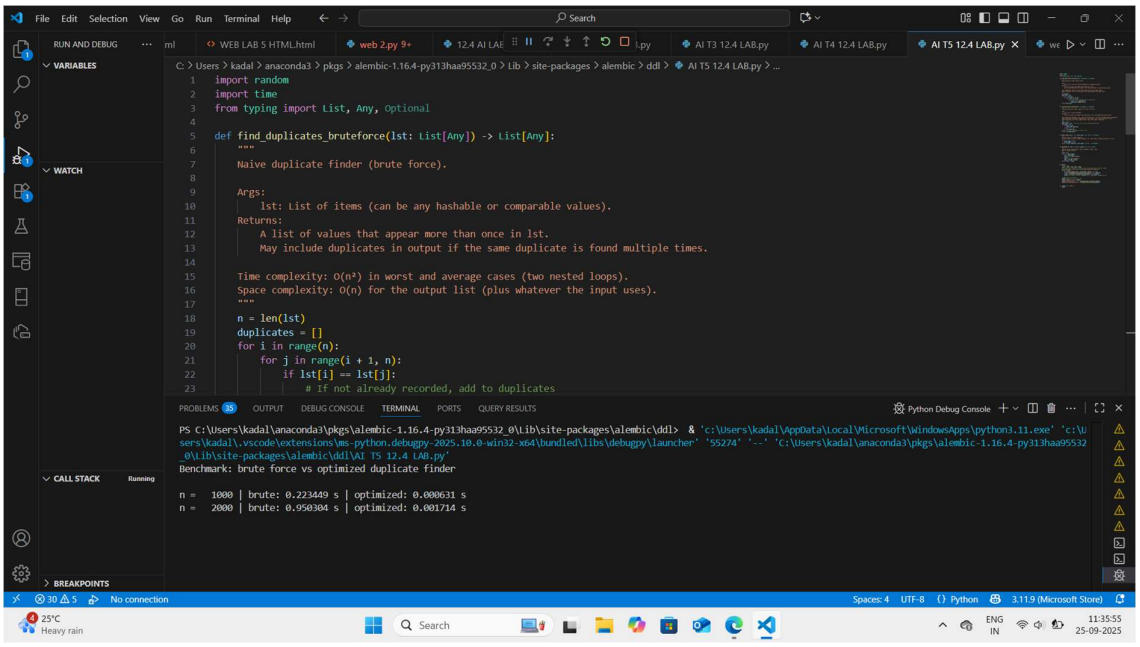
• Task: Give AI a naive algorithm (e.g., $O(n^2)$ duplicate search).

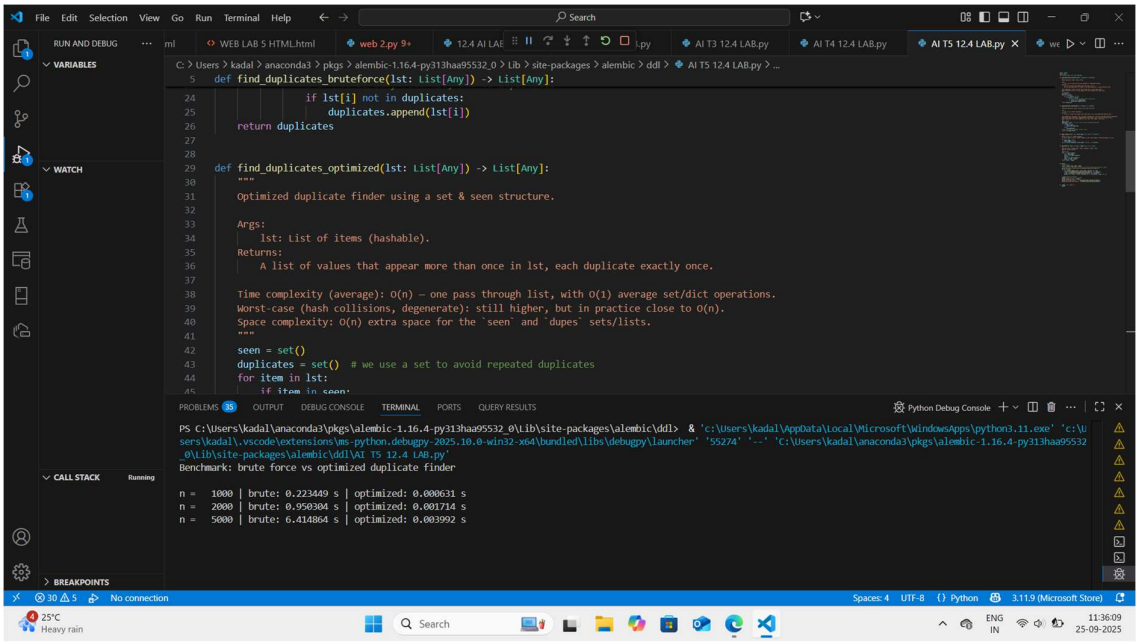
• Instructions:

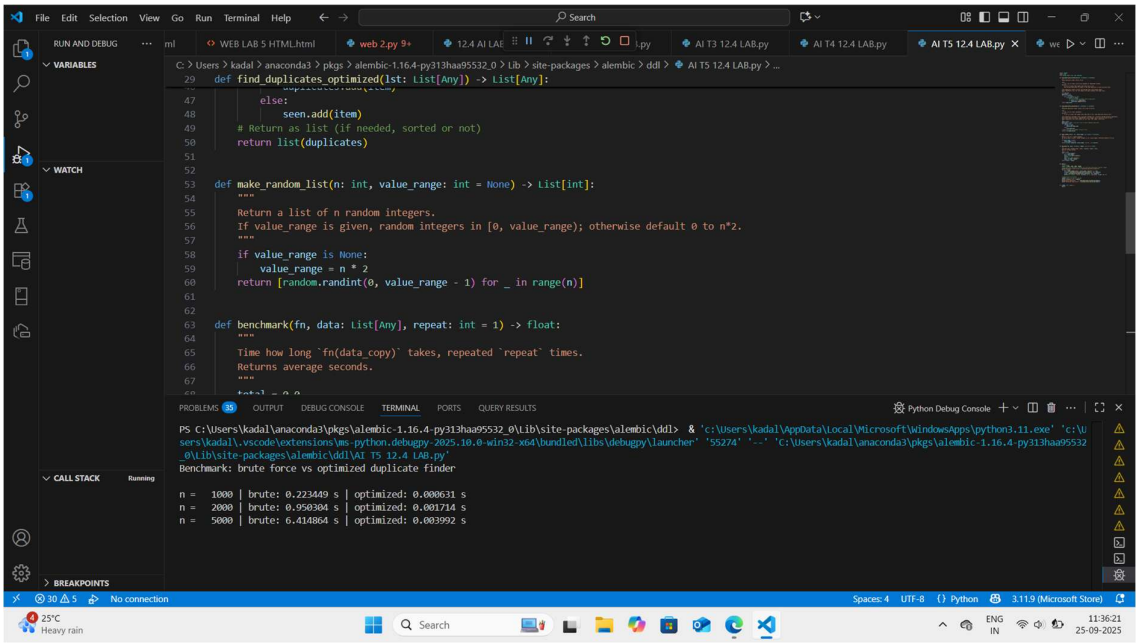
o Students write a brute force duplicate-finder.

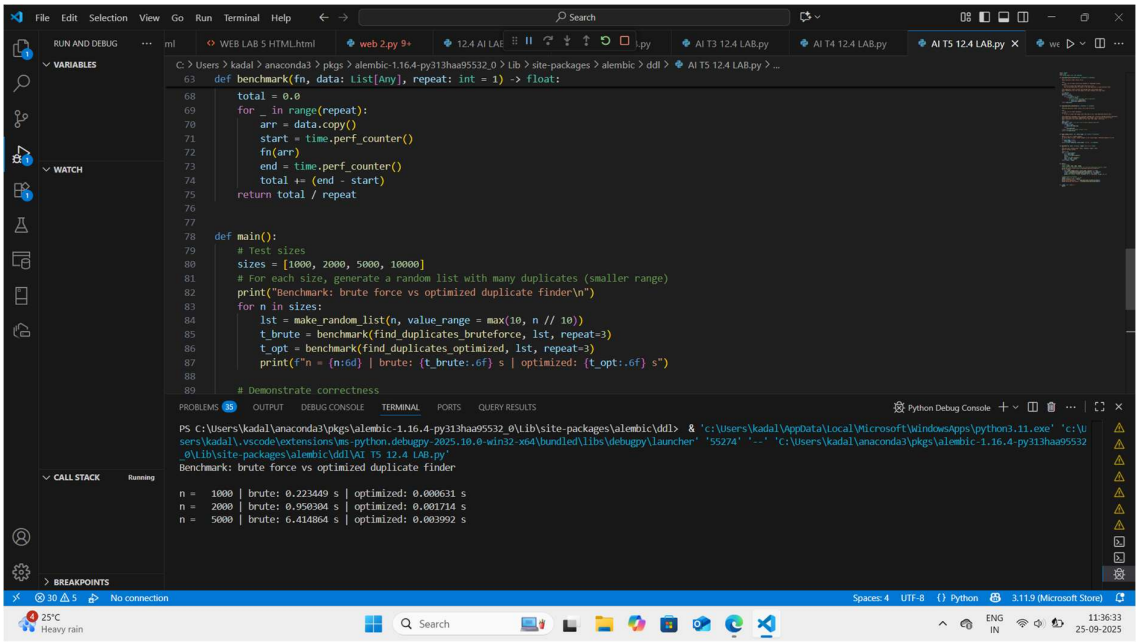
o Ask AI to optimize it (e.g., by using sets/dictionaries with
O(n) time).
o Compare execution times with large input sizes.

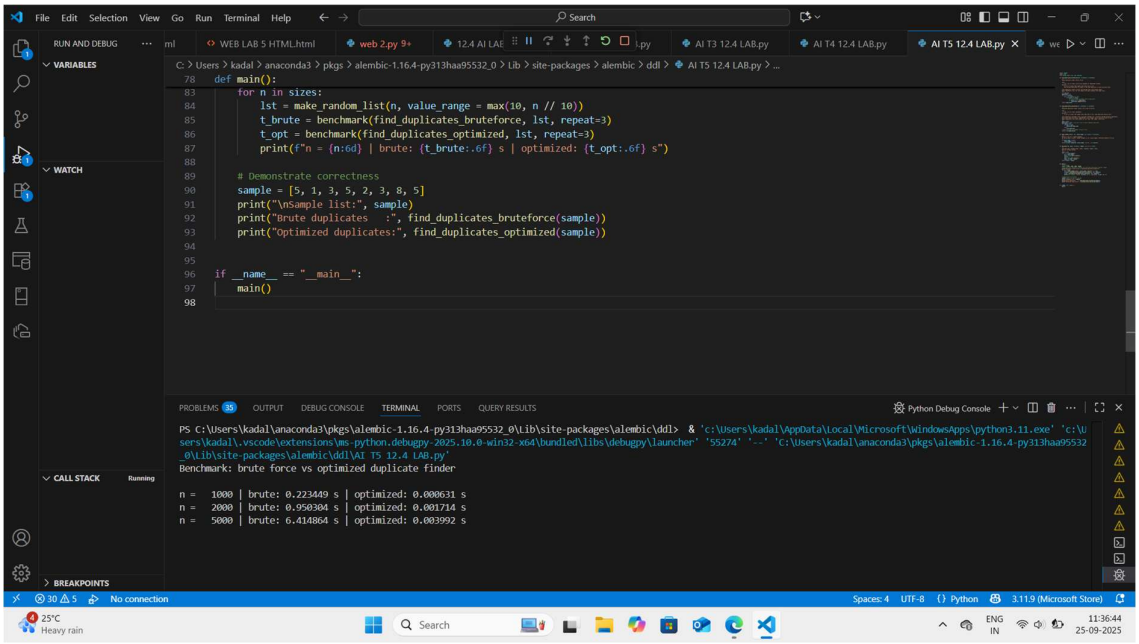
CODE & OUTPUT:




def find_duplicates_bruteforce(lst: List[Any]) -> List[Any]:
if lst[i] not in duplicates:
duplicates.append(lst[i])
return duplicates
def find_duplicates_optimized(lst: List[Any]) -> List[Any]:
"""
Optimized duplicate finder using a set & seen structure.
Args:
lst: List of items (hashable).
Returns:
A list of values that appear more than once in lst, each duplicate exactly once.
Time complexity (average): O(n) — one pass through list, with O(1) average set/dict operations.
Worst-case (hash collisions, degenerate): still higher, but in practice close to O(n).
Space complexity: O(n) extra space for the `seen` and `dupes` sets/lists.
"""
seen = set()
duplicates = set() # we use a set to avoid repeated duplicates
for item in lst:
PS C:\Users\kadal\anaconda3\pkgs\alembic-1.16.4-py313haa95532_0\Lib\site-packages\alembic\ddl> & 'c:\Users\kadal\AppData\Local\Microsoft\WindowsApps\python3.11.exe' 'c:\Users\kadal\.vscode\extensions\ms-python.debugpy-2025.10.0-win32-x64\bundled\libs\debugpy\launcher' '55274' '--' 'C:\Users\kadal\anaconda3\pkgs\alembic-1.16.4-py313haa95532_0\Lib\site-packages\alembic\ddl\AI T5 12.4 LAB.py'
Benchmark: brute force vs optimized duplicate finder
n = 1000 | brute: 0.223449 s | optimized: 0.000631 s
n = 2000 | brute: 0.950304 s | optimized: 0.001714 s
n = 5000 | brute: 6.414864 s | optimized: 0.003992 s


def find_duplicates_optimized(lst: List[Any]) -> List[Any]:
else:
seen.add(item)
# Return as list (if needed, sorted or not)
return list(duplicates)
def make_random_list(n: int, value_range: int = None) -> List[int]:
"""
Return a list of n random integers.
If value_range is given, random integers in [0, value_range); otherwise default 0 to n*2.
"""
if value_range is None:
value_range = n * 2
return [random.randint(0, value_range - 1) for _ in range(n)]
def benchmark(fn, data: List[Any], repeat: int = 1) -> float:
"""
Time how long `fn(data_copy)` takes, repeated `repeat` times.
Returns average seconds.
"""
PS C:\Users\kadal\anaconda3\pkgs\alembic-1.16.4-py313haa95532_0\Lib\site-packages\alembic\ddl> & 'c:\Users\kadal\AppData\Local\Microsoft\WindowsApps\python3.11.exe' 'c:\Users\kadal\.vscode\extensions\ms-python.debugpy-2025.10.0-win32-x64\bundled\libs\debugpy\launcher' '55274' '--' 'C:\Users\kadal\anaconda3\pkgs\alembic-1.16.4-py313haa95532_0\Lib\site-packages\alembic\ddl\AI T5 12.4 LAB.py'
Benchmark: brute force vs optimized duplicate finder
n = 1000 | brute: 0.223449 s | optimized: 0.000631 s
n = 2000 | brute: 0.950304 s | optimized: 0.001714 s
n = 5000 | brute: 6.414864 s | optimized: 0.003992 s






```
n =   1000 | brute: 0.223449 s | optimized: 0.000631 s
n =   2000 | brute: 0.950304 s | optimized: 0.001714 s
n =   5000 | brute: 6.414864 s | optimized: 0.003992 s
n =  10000 | brute: 43.288533 s | optimized: 0.007612 s

Sample list: [5, 1, 3, 5, 2, 3, 8, 5]
Brute duplicates   : [5, 3]
Optimized duplicates: [3, 5]
```

EXPLAINATION:

- The **brute force version** uses nested loops (`i` and `j`) to compare every pair. That gives **O(n^2)** time complexity. It also checks `lst[i] not in duplicates` to avoid adding the same duplicate multiple times, but that check itself is O(n) in worst case, making it even slower in practice.
- The **optimized version** uses two sets:
  - `seen` to record which items have already been encountered.
  - `duplicates` to record items that appear more than once (without repetition).
  - For each item, you do `if item in seen: duplicates.add(item)` else `seen.add(item)` .
  - Both set membership / insertion operations are average O(1), so total is **O(n)** average time.
- We convert `duplicates` set to list at the end for output; you could also return a set directly.
- The benchmarking compares the two methods on random lists with many duplicates (using a relatively small `value_range`) so collisions / duplicates are common — that better highlights the speed difference.

You'll observe that for moderate `n` (e.g. 5,000+), the brute force method becomes very slow, while the optimized one scales linearly.