

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:
 - Students implement Bubble Sort normally.
 - Ask AI to generate inline comments explaining key logic (like swapping, passes, and termination).
 - 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:
 - Students implement Bubble Sort normally.
 - Ask AI to generate inline comments explaining key

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

CODE & OUTPUT:

The screenshot shows the VS Code interface with the Python extension installed. The code editor displays a Python script named `bubble_sort.py` containing the following code:

```
def bubble_sort(arr):
    """ Sorts the list 'arr' in-place using Bubble Sort algorithm.
    """
    n = len(arr)
    # Outer loop for each pass over the array
    for i in range(n):
        # We assume no swaps this pass (to detect if array already sorted)
        swapped = False

        # For each pair of adjacent elements in positions [0 .. n-1]
        # (since last i elements already bubbled to correct positions)
        for j in range(0, n - i - 1):
            # If the element at j is greater than the one at j+1, swap them
            if arr[j] > arr[j + 1]:
                # Swap the two elements
                arr[j], arr[j + 1] = arr[j + 1], arr[j]
                swapped = True # mark that a swap happened in this pass

        # If no two elements were swapped in the inner loop, the list is already sorted
        if not swapped:
            break # Early termination - no further passes needed

    return arr
```

The terminal tab shows the execution of the script:

```
PS C:\Users\Kadal> python bubble_sort.py
Before: [5, 1, 4, 2, 8]
After: [1, 2, 4, 5, 8]
```

The screenshot shows the VS Code interface with the Python extension installed. The code editor displays the same `bubble_sort.py` script, but with additional code added at the bottom:

```
if __name__ == "__main__":
    data = [5, 1, 4, 2, 8]
    print("Before:", data)
    sorted_data = bubble_sort(data)
    print("After: ", sorted_data)
```

The terminal tab shows the execution of the script:

```
PS C:\Users\Kadal> python bubble_sort.py
Before: [5, 1, 4, 2, 8]
After: [1, 2, 4, 5, 8]
```

EXPLANATION:

- **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 $O(n^2)$.

- **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 $\sim n$ 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 $\sim n$ comparisons, so it is $O(n)$ in the best case (with the early termination optimization).

- **Average-case time complexity**:

Generally also $O(n^2)$.

- **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:

- Students implement Bubble Sort first.
- Ask AI to suggest an alternative (Insertion Sort).
- 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:

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

CODE & OUTPUT:

File Edit Selection View Go Run Terminal Help ↻ → ⌂ Search ⌂ v

RUN AND DEBUG ... LAB 11.1.py AI LAB 11.1 T8.py WEB LAB 4 HTMLhtml WEB LAB 5 HTMLhtml web 2.py 9+ 12.4 AI LAB T1.py AI T 2 12.4 LAB.py x web lab.py ⌂ v

To customize Run and Debug, open a folder and create a launch.json file.

Debug using a terminal command or in an interactive chat.

Run and Debug

```

23 def insertion_sort(arr):
24     key = arr[1]
25     j = i = 1
26     # Shift elements that are greater than key to one position ahead
27     while j >= 0 and arr[j] > key:
28         arr[j + 1] = arr[j]
29         j -= 1
30     # Place key in its correct spot
31     arr[j + 1] = key
32     return arr
33
34
35
36
37
38
39
40
41 def make_almost_sorted(n, num_swaps=5):
42     """
43     Return a list of size n that is nearly sorted,
44     by starting with [0,1,2,...,n-1] and performing a few random adjacent swaps.
45     """
46     lst = list(range(n))
47     for _ in range(num_swaps):
48         i = random.randint(0, n - 2)
49         # make adjacent pair
50         lst[i], lst[i + 1] = lst[i + 1], lst[i]
51     return lst

```

PROBLEMS 39 **OUTPUT** **DEBUG CONSOLE** **TERMINAL** **PORTS** **QUERY RESULTS**

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: 0.000632s, 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, 12, 13, 14, 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-py313hae95532_0\Lib\site-packages\alembic\ddl>

BREAKPOINTS

⌚ 30 ⚡ 5 ⚡ No connection

25°C Heavy rain

Spaces: 4 UTF-8 ⓘ Python 3.11.9 (Microsoft Store)

11:18:02 25-09-2025

File Edit Selection View Go Run Terminal Help ↻ → ⌂ Search ⌂ v

RUN AND DEBUG ... LAB 11.1.py AI LAB 11.1 T8.py WEB LAB 4 HTMLhtml WEB LAB 5 HTMLhtml web 2.py 9+ 12.4 AI LAB T1.py AI T 2 12.4 LAB.py x web lab.py ⌂ v

To customize Run and Debug, open a folder and create a launch.json file.

Debug using a terminal command or in an interactive chat.

Run and Debug

```

52
53
54 def benchmark(sort_fn, arr):
55     """
56     Time how long sort_fn takes to sort a *copy* of arr.
57     Returns the duration in seconds.
58     """
59     arr_copy = arr.copy()
60     start = time.perf_counter()
61     sort_fn(arr_copy)
62     end = time.perf_counter()
63     return end - start
64
65
66 def main():
67     # testing parameters
68     test_sizes = [1000, 5000, 10000]
69     swap_counts = [1, 5, 20] # how many adjacent swaps to create disorder
70
71     print("Comparing Bubble Sort vs Insertion Sort on nearly sorted arrays\n")
72     for n in test_sizes:
73         for swaps in swap_counts:

```

PROBLEMS 39 **OUTPUT** **DEBUG CONSOLE** **TERMINAL** **PORTS** **QUERY RESULTS**

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: 0.000632s, 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, 12, 13, 14, 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-py313hae95532_0\Lib\site-packages\alembic\ddl>

⌚ 30 ⚡ 5 ⚡ No connection

25°C Heavy rain

Spaces: 4 UTF-8 ⓘ Python 3.11.9 (Microsoft Store)

11:18:23 25-09-2025

```

File Edit Selection View Go Run Terminal Help ← → ⌂ Search ⌂ v
RUN AND DEBUG ... LAB 11.1.py AI LAB 11.1 T8.py WEB LAB 4 HTMLhtml WEB LAB 5 HTMLhtml web 2.py 9+ 12.4 AI LAB T1.py AI T 2 12.4 LAB.py ...
C:\> Users > kadal > anaconda3 > pkgs > alembic-1.16.4-py313ha095532_0 > Lib > site-packages > alembic > dd > AI T 2 12.4 LAB.py > ...
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.
66 def main():
    arr = make_almost_sorted(n, num_swaps=swaps)
    t_bubble = benchmark(bubble_sort, arr)
    t_insert = benchmark(insertion_sort, arr)
    print(f"\n{n=:d}, swaps = {swaps=:d} + bubble: {t_bubble:.6f}s, insertion: {t_insert:.6f}s")
    # demonstration: show sorting result correctness
    sample = make_almost_sorted(20, num_swaps=3)
    print("\nSample (nearly sorted) before:", sample)
    sorted_by_bubble = bubble_sort(sample.copy())
    sorted_by_insert = insertion_sort(sample.copy())
    print("Sorted by bubble : ", sorted_by_bubble)
    print("Sorted by insertion: ", sorted_by_insert)
    print("Sorted by insertion:", sorted_by_insert)

if __name__ == "__main__":
    main()

```

PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL PORTS QUERY RESULTS

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-py313ha095532_0\Lib\site-packages\alembic\ddl>

```

BREAKPOINTS

Spaces: 4 UTF-8 {} Python 3.11.9 (Microsoft Store)

25°C Heavy rain ENG IN 11:18:39 25-09-2025

EXPLANATION:

- **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:

The screenshot shows the Microsoft Visual Studio Code interface with the Python extension installed. The code editor displays a Python script for a linear search algorithm. The Python Debug Console tab is active, showing the execution of the script and its performance metrics. The status bar at the bottom indicates the date as 25-09-2025.

```
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 T 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 = 49895
Linear search found at index: 50000
```

The screenshot shows the Microsoft Visual Studio Code interface with the Python extension installed. The code editor displays a Python script for a binary search algorithm. The Python Debug Console tab is active, showing the execution of the script and its performance metrics. The status bar at the bottom indicates the date as 25-09-2025.

```
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 T 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 = 49895
Linear search found at index: 50000
```

The screenshot shows the Visual Studio Code interface with the Python extension installed. The left sidebar has 'RUN AND DEBUG' expanded, showing 'Run and Debug' selected. The main editor area contains a Python script for a binary search algorithm and a function to generate random lists. The bottom right shows the Python Debug Console output:

```
C:\> Users > kadal > anaconda3 > pkgs > alembic-1.16.4-py313haa95532_0 > Lib > site-packages > alembic > dd1 > AI T 12.4 LAB.py > ...
28 def binary_search(arr: List[Any], target: Any) -> Optional[int]:
49     low = 0
50     high = len(arr) - 1
51
52     while low <= high:
53         mid = (low + high) // 2
54         mid_val = arr[mid]
55
56         if mid_val == target:
57             return mid
58         elif mid_val < target:
59             low = mid + 1
60         else:
61             high = mid - 1
62
63     return None
64
65
66 def make_random_list(n: int, value_range: int = 100000) -> List[int]:
67     """Generate a list of n random integers (unsorted)."""
68     return [random.randint(0, value_range) for _ in range(n)]
69
70
PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL PORTS QUERY RESULTS
PS C:\Users\Kadal\Anaconda3\pkgs\alembic-1.16.4-py313haa95532_0\lib\site-packages\alembic\dd1 & 'c:\Users\kadal\AppData\Local\Microsoft\WindowsApps\python3.11.exe' 'c:\Users\kadal\anaconda3\pkgs\alembic-1.16.4-py313haa95532_0\lib\site-packages\alembic\dd1\AI T 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 = 49895
Linear search found at index: 50000
Spaces: 4 UTF-8 {} Python 3.11.9 (Microsoft Store)
Python Debug Console + v □ ⌂ ...
```

The screenshot shows the Visual Studio Code interface with the Python extension installed. The left sidebar has 'RUN AND DEBUG' expanded, showing 'Run and Debug' selected. The main editor area contains a Python script for benchmarking search functions. The bottom right shows the Python Debug Console output:

```
C:\> Users > kadal > anaconda3 > pkgs > alembic-1.16.4-py313haa95532_0 > Lib > site-packages > alembic > dd1 > AI T 12.4 LAB.py > benchmark
70 def make_sorted_list(n: int) -> List[int]:
71     """Generate a sorted list of n integers (0,1,2,...,n-1)."""
72     return list(range(n))
73
74
75 def benchmark(search_fn, arr: List[Any], target: Any, repeat: int = 1) -> float:
76     """
77     Time how long search_fn takes to find target in arr, repeated `repeat` times.
78     Returns average time in seconds.
79     """
80
81     total = 0.0
82     for _ in range(repeat):
83         start = time.perf_counter()
84         search_fn(arr, target)
85         end = time.perf_counter()
86         total += (end - start)
87     return total / repeat
88
89
90 def main():
91     # Test parameters
92     n = 100000 # size of list
PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL PORTS QUERY RESULTS
PS C:\Users\Kadal\Anaconda3\pkgs\alembic-1.16.4-py313haa95532_0\lib\site-packages\alembic\dd1 & 'c:\Users\kadal\AppData\Local\Microsoft\WindowsApps\python3.11.exe' 'c:\Users\kadal\anaconda3\pkgs\alembic-1.16.4-py313haa95532_0\lib\site-packages\alembic\dd1\benchmark\AI T 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 = 49895
Linear search found at index: 50000
Ln 78, Col 82 Spaces: 4 UTF-8 {} Python 3.11.9 (Microsoft Store)
Python Debug Console + v □ ⌂ ...
```

```

File Edit Selection View Go Run Terminal Help ⏪ → Search
RUN AND DEBUG ... 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 T 3 12.4 LAB.py we Dv ...
C:\> Users > kadal > anaconda3 > pkgs > alembic-1.16.4-py313haa95532_0 > Lib > site-packages > alembic > dd1 > AI T 3 12.4 LAB.py > ...
90 def main():
91     # Create unsorted list and sorted list
92     unsorted = make_random_list()
93     sorted_list = sorted(unsorted)
94     # Or simply: sorted_list = make_sorted_list(n)
95
96     # Pick a target (choose something that is in the list)
97     # For safety, pick from sorted list
98     target_value = sorted_list[n // 2]
99
100    print("== Testing on Unsorted Data ==")
101    t_lin = benchmark(linear_search, unsorted, target_value, repeat=5)
102    t_bin_unsorted = None
103
104    try:
105        # We can "call" binary search on unsorted data, but it will give incorrect result
106        t_bin_unsorted = benchmark(binary_search, unsorted, target_value, repeat=5)
107    except Exception as e:
108        print("Binary search on unsorted data failed: ", e)
109    print(f"Linear search time (unsorted): {t_lin:.6f} s")
110    if t_bin_unsorted is not None:
111        print(f"Binary search time (unsorted, invalid) : {t_bin_unsorted:.6f} s (result may be wrong)")
112
113    print("== Testing on Sorted Data ==")
114    t_lin_sorted = benchmark(linear_search, sorted_list, target_value, repeat=5)
115    t_bin_sorted = benchmark(binary_search, sorted_list, target_value, repeat=5)
116    print(f"Linear search time (sorted list) : {t_lin:.6f} s")
117    print(f"Binary search time (sorted list) : {t_bin:.6f} s")
118
119    correctness_check(target = 49805)
120
121    # Validate correctness
122    idx_lin = linear_search(sorted_list, target_value)
123    idx_bin = binary_search(sorted_list, target_value)
124    print(f"Correctness check: target = {target}, target_value = {target_value}")
125    print(f"Linear search found at index: {idx_lin}")
126    print(f"Binary search found at index: {idx_bin}")
127
128
129    print("== Summary & When to Prefer Binary Search ==")
130    print("1. Linear search works whether or not the data is sorted; binary search requires sorted data.")
131    print("2. Linear search is O(n), while binary search is O(log n) on sorted data.")
132    print("3. For small arrays or one-time searches, the overhead of binary search may not pay off.")
133    print("4. If you perform many searches on the same sorted list, binary search is much more efficient overall.")
134    print("5. If the data is unsorted and you only search once, linear search may be preferable (to avoid sorting).")
135    print("6. In practice, for large n and many searches, binary search is the better choice. (Binary search is preferred for large sorted datasets.)")
136
137 if __name__ == "__main__":

```

PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL PORTS QUERY RESULTS

Python Debug Console + v D v ...

Ln 88, Col 1 Spaces: 4 UTF-8 {} Python 3.11.9 (Microsoft Store)

25-09-2025 11:23:36 ENG IN WiFi

```

File Edit Selection View Go Run Terminal Help ⏪ → Search
RUN AND DEBUG ... 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 T 3 12.4 LAB.py we Dv ...
C:\> Users > kadal > anaconda3 > pkgs > alembic-1.16.4-py313haa95532_0 > Lib > site-packages > alembic > dd1 > AI T 3 12.4 LAB.py > ...
90 def main():
91     # Create unsorted list and sorted list
92     unsorted = make_random_list()
93     sorted_list = sorted(unsorted)
94     # Or simply: sorted_list = make_sorted_list(n)
95
96     # Pick a target (choose something that is in the list)
97     # For safety, pick from sorted list
98     target_value = sorted_list[n // 2]
99
100    print("== Testing on Unsorted Data ==")
101    t_lin = benchmark(linear_search, unsorted, target_value, repeat=5)
102    t_bin_unsorted = None
103
104    try:
105        # We can "call" binary search on unsorted data, but it will give incorrect result
106        t_bin_unsorted = benchmark(binary_search, unsorted, target_value, repeat=5)
107    except Exception as e:
108        print("Binary search on unsorted data failed: ", e)
109    print(f"Linear search time (unsorted): {t_lin:.6f} s")
110    if t_bin_unsorted is not None:
111        print(f"Binary search time (unsorted, invalid) : {t_bin_unsorted:.6f} s (result may be wrong)")
112
113    print("== Testing on Sorted Data ==")
114    t_lin_sorted = benchmark(linear_search, sorted_list, target_value, repeat=5)
115    t_bin_sorted = benchmark(binary_search, sorted_list, target_value, repeat=5)
116    print(f"Linear search time (sorted list) : {t_lin:.6f} s")
117    print(f"Binary search time (sorted list) : {t_bin:.6f} s")
118
119    correctness_check(target = 49805)
120
121    # Validate correctness
122    idx_lin = linear_search(sorted_list, target_value)
123    idx_bin = binary_search(sorted_list, target_value)
124    print(f"Correctness check: target = {target}, target_value = {target_value}")
125    print(f"Linear search found at index: {idx_lin}")
126    print(f"Binary search found at index: {idx_bin}")
127
128
129    print("== Summary & When to Prefer Binary Search ==")
130    print("1. Linear search works whether or not the data is sorted; binary search requires sorted data.")
131    print("2. Linear search is O(n), while binary search is O(log n) on sorted data.")
132    print("3. For small arrays or one-time searches, the overhead of binary search may not pay off.")
133    print("4. If you perform many searches on the same sorted list, binary search is much more efficient overall.")
134    print("5. If the data is unsorted and you only search once, linear search may be preferable (to avoid sorting).")
135    print("6. In practice, for large n and many searches, binary search is the better choice. (Binary search is preferred for large sorted datasets.)")
136
137 if __name__ == "__main__":

```

PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL PORTS QUERY RESULTS

Python Debug Console + v D v ...

Ln 88, Col 1 Spaces: 4 UTF-8 {} Python 3.11.9 (Microsoft Store)

25-09-2025 11:23:52 ENG IN WiFi

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.)

EXPLANATION:

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.) [dirtyhandsCoding... +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:

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

- File Explorer:** Shows files like LAB 4 HTML.html, web 2.py, AI T 2 12.4 LAB.py, AI T 3 12.4 LAB.py, AI T 4 12.4 LAB.py, and web lab.py.
- Code Editor:** Displays Python code for implementing Quick Sort and Merge Sort. The Quick Sort implementation uses recursion and partitioning. The Merge Sort implementation uses a divide-and-conquer approach with a pivot element.
- Output Panel:** Shows the results of running the code, comparing execution times for random, sorted, and reverse-sorted lists.
- Terminal:** Shows the command used to run the code: "c:\Windows\app\python3.11.exe" "c:\Users\kadal\vscodeextensions\ms-python.debugger-2025.10.0-win32-x64\bundled\libs\debug\launcher" "55214" "...".
- Performance Results:** The output indicates the execution times for Quick Sort and Merge Sort on different list types.

The screenshot shows the Visual Studio Code interface with the Python extension installed. The code editor displays a Python script named `web 2.py` containing implementations of Quick Sort and Merge Sort. The Python Debug Console tab is active, showing the execution of the script. The output indicates a random list of size 50000 was sorted using both algorithms, with Merge Sort being faster at approximately 1.4798 seconds compared to Quick Sort's 1.0666 seconds.

```
C:\> Users > kadal > anaconda3 > pkgs > alembic-1.16.4-py313sha95532.0 > Lib > site-packages > alembic > dd1 > AI T 2 12.4 LAB.py > ...
25
26
27 def quick_sort(arr):
28     """
29         Wrapper function for Quick Sort.
30     """
31     quick_sort_recursive(arr, 0, len(arr) - 1)
32
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:]
42
43         merge_sort(left_half)
44         merge_sort(right_half)
45
46         i = j = k = 0
47
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
57
58     # Copy any remaining elements
59     while i < len(left_half):
60         arr[k] = left_half[i]
61         i += 1
62         k += 1
63
64     while j < len(right_half):
65         arr[k] = right_half[j]
66         j += 1
67         k += 1
68
69 # --- Comparison ---

... Random List ...
Quick Sort time: 1.0666 seconds
Merge Sort time: 1.4798 seconds

... Sorted List ...

PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL PORTS QUERY RESULTS Python Debug Console + v □ ⌂ ... x
crossoft\WindowsApps\python3.11.exe" "C:\Users\kadal\vscode\extensions\ms-python.python\2025.10.0-win32-x64\bundled\libs\debugpy\launcher" "55214" "--" "C:\Users\kadal\anaconda3\pkgs\alembic-1.16.4-py313sha95532.0\lib\site-packages\alembic\dd1\AI T 2 12.4 LAB.py"
--- Performance Comparison (List Size: 50000) ---

In 135, Col 21, Spaces 4, UTF-8, CRLF, Python 3.11.9 (Microsoft Store) 11:29:16
ENG IN 25-09-2025
```

This screenshot shows the same Visual Studio Code environment, but the code editor now displays the merge sort implementation from the previous screenshot. The Python Debug Console output remains identical, showing the execution of the merge sort function and its performance comparison with quick sort.

```
C:\> Users > kadal > anaconda3 > pkgs > alembic-1.16.4-py313sha95532.0 > Lib > site-packages > alembic > dd1 > AI T 2 12.4 LAB.py > ...
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:]
42
43         merge_sort(left_half)
44         merge_sort(right_half)
45
46         i = j = k = 0
47
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
57
58     # Copy any remaining elements
59     while i < len(left_half):
60         arr[k] = left_half[i]
61         i += 1
62         k += 1
63
64     while j < len(right_half):
65         arr[k] = right_half[j]
66         j += 1
67         k += 1
68
69 # --- Comparison ---

... Random List ...
Quick Sort time: 1.0666 seconds
Merge Sort time: 1.4798 seconds

... Sorted List ...

PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL PORTS QUERY RESULTS Python Debug Console + v □ ⌂ ... x
crossoft\WindowsApps\python3.11.exe" "C:\Users\kadal\vscode\extensions\ms-python.python\2025.10.0-win32-x64\bundled\libs\debugpy\launcher" "55214" "--" "C:\Users\kadal\anaconda3\pkgs\alembic-1.16.4-py313sha95532.0\lib\site-packages\alembic\dd1\AI T 2 12.4 LAB.py"
--- Performance Comparison (List Size: 50000) ---

In 135, Col 21, Spaces 4, UTF-8, CRLF, Python 3.11.9 (Microsoft Store) 11:29:31
ENG IN 25-09-2025
```

The screenshot shows the VS Code interface during a debugging session. The top navigation bar includes File, Edit, Selection, View, Go, Run, Terminal, Help, and a search bar. The left sidebar features RUN AND DEBUG, VARIABLES, WATCH, CALL STACK, and BREAKPOINTS sections. The main editor area contains a Python script named AI T4 12.4 LAB.py. The script performs a performance comparison between Quick Sort and Merge Sort on lists of size 50000. It includes imports for random and time, defines functions for quick sort and merge sort, and prints execution times for each sort type on both random and sorted lists. The Python Debug Console at the bottom shows the output of the script's execution.

```
C:\> Users > kadal > anaconda3 > pkgs > alembic-1.16.4-py313haa95532_0 > Lib > site-packages > alembic > ddl > AI T4 12.4 LAB.py > ...
70 def run_comparison():
71     """
72     Compares Quick Sort and Merge Sort on different list types.
73     """
74     list_size = 50000
75
76     # Test cases
77     random_list = [random.randint(0, list_size) for _ in range(list_size)]
78     sorted_list = sorted(random_list)
79     reverse_sorted_list = sorted(random_list, reverse=True)
80
81     print("---- Performance Comparison (List Size: {}) ----".format(list_size))
82
83     # Test on Random List
84     print("\n---- Random List ---")
85
86     # Quick Sort
87     quick_list = list(random_list)
88     start_time = time.time()
89     quick_sort(quick_list)
90     end_time = time.time()
91     print("Quick Sort time: {:.4f} seconds".format(end_time - start_time))
92
93     # Merge Sort
94     merge_list = list(random_list)
95     start_time = time.time()
96     merge_sort(merge_list)
97     end_time = time.time()
98     print("Merge Sort time: {:.4f} seconds".format(end_time - start_time))
99
100    # Test on Sorted List
101   print("\n---- Sorted List ---")
102
103   # Quick Sort
104   quick_list = list(sorted_list)
105   start_time = time.time()
106   quick_sort(quick_list)
107   end_time = time.time()
108   print("Quick Sort time: {:.4f} seconds".format(end_time - start_time))
109
110   # Merge Sort
111   merge_list = list(sorted_list)
112   start_time = time.time()
113   merge_sort(merge_list)
114
115
116
117
118
119
120
121
122
123
124
125
126
127
128
129
130
131
132
133
134
135
136
137
138
139
140
141
142
143
144
145
146
147
148
149
150
151
152
153
154
155
156
157
158
159
160
161
162
163
164
165
166
167
168
169
170
171
172
173
174
175
176
177
178
179
180
181
182
183
184
185
186
187
188
189
190
191
192
193
194
195
196
197
198
199
200
201
202
203
204
205
206
207
208
209
210
211
212
213
214
215
216
217
218
219
220
221
222
223
224
225
226
227
228
229
230
231
232
233
234
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
300
301
302
303
304
305
306
307
308
309
310
311
312
313
314
315
316
317
318
319
320
321
322
323
324
325
326
327
328
329
330
331
332
333
334
335
336
337
338
339
340
341
342
343
344
345
346
347
348
349
350
351
352
353
354
355
356
357
358
359
360
361
362
363
364
365
366
367
368
369
370
371
372
373
374
375
376
377
378
379
380
381
382
383
384
385
386
387
388
389
390
391
392
393
394
395
396
397
398
399
400
401
402
403
404
405
406
407
408
409
410
411
412
413
414
415
416
417
418
419
420
421
422
423
424
425
426
427
428
429
430
431
432
433
434
435
436
437
438
439
440
441
442
443
444
445
446
447
448
449
450
451
452
453
454
455
456
457
458
459
460
461
462
463
464
465
466
467
468
469
470
471
472
473
474
475
476
477
478
479
480
481
482
483
484
485
486
487
488
489
490
491
492
493
494
495
496
497
498
499
500
501
502
503
504
505
506
507
508
509
510
511
512
513
514
515
516
517
518
519
520
521
522
523
524
525
526
527
528
529
530
531
532
533
534
535
536
537
538
539
540
541
542
543
544
545
546
547
548
549
550
551
552
553
554
555
556
557
558
559
559
```

The screenshot shows the VS Code interface during a debugging session. The top navigation bar includes File, Edit, Selection, View, Go, Run, Terminal, Help, and a search bar. The left sidebar features RUN AND DEBUG, VARIABLES, WATCH, CALL STACK, and BREAKPOINTS sections. The main editor area contains a Python script named AI T4 12.4 LAB.py. The script performs a performance comparison between Quick Sort and Merge Sort on lists of size 50000. It includes imports for random and time, defines functions for quick sort and merge sort, and prints execution times for each sort type on both random and sorted lists. The Python Debug Console at the bottom shows the output of the script's execution.

```
C:\> Users > kadal > anaconda3 > pkgs > alembic-1.16.4-py313haa95532_0 > Lib > site-packages > alembic > ddl > AI T4 12.4 LAB.py > ...
70 def run_comparison():
71     """
72     Compares Quick Sort and Merge Sort on different list types.
73     """
74     list_size = 50000
75
76     # Test cases
77     random_list = [random.randint(0, list_size) for _ in range(list_size)]
78     sorted_list = sorted(random_list)
79     reverse_sorted_list = sorted(random_list, reverse=True)
80
81     print("---- Performance Comparison (List Size: {}) ----".format(list_size))
82
83     # Test on Random List
84     print("\n---- Random List ---")
85
86     # Quick Sort
87     quick_list = list(random_list)
88     start_time = time.time()
89     quick_sort(quick_list)
90     end_time = time.time()
91     print("Quick Sort time: {:.4f} seconds".format(end_time - start_time))
92
93     # Merge Sort
94     merge_list = list(random_list)
95     start_time = time.time()
96     merge_sort(merge_list)
97     end_time = time.time()
98     print("Merge Sort time: {:.4f} seconds".format(end_time - start_time))
99
100    # Test on Sorted List
101   print("\n---- Sorted List ---")
102
103   # Quick Sort
104   quick_list = list(sorted_list)
105   start_time = time.time()
106   quick_sort(quick_list)
107   end_time = time.time()
108   print("Quick Sort time: {:.4f} seconds".format(end_time - start_time))
109
110   # Merge Sort
111   merge_list = list(sorted_list)
112   start_time = time.time()
113   merge_sort(merge_list)
114
115
116
117
118
119
119
```

```

70 def run_comparison():
115     print("Merge Sort time: {:.4f} seconds".format(end_time - start_time))
116
117     # Test on Reverse Sorted List
118     print("\n--- Reverse Sorted List ---")
119
120     # Quick Sort
121     quick_list = list(reverse_sorted_list)
122     start_time = time.time()
123     quick_sort(quick_list)
124     end_time = time.time()
125     print("Quick Sort time: {:.4f} seconds".format(end_time - start_time))
126
127     # Merge Sort
128     merge_list = list(reverse_sorted_list)
129     start_time = time.time()
130     merge_sort(merge_list)
131     end_time = time.time()
132     print("Merge Sort time: {:.4f} seconds".format(end_time - start_time))
133
134 if __name__ == "__main__":
135     run_comparison()

```

PROBLEMS 35 OUTPUT DEBUG CONSOLE TERMINAL PORTS QUERY RESULTS

```

crossoft\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.

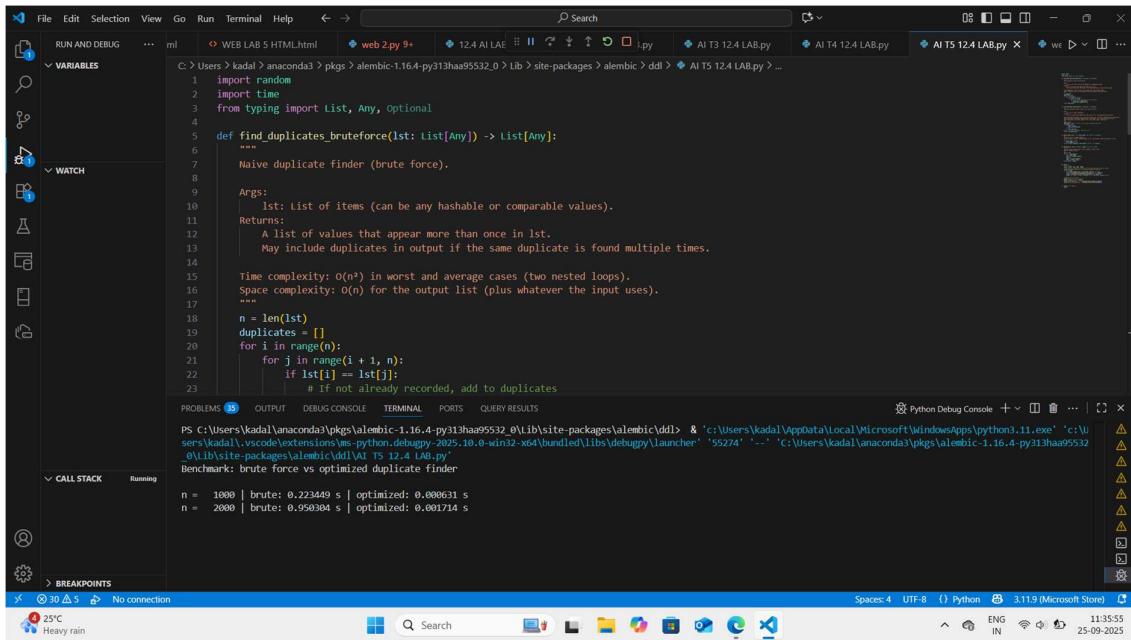
PROMPT:

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:



The screenshot shows the Microsoft Visual Studio Code interface. The code editor displays a Python script named `find_duplicates_bruteforce.py`. The script defines a function `find_duplicates_bruteforce` that takes a list of items and returns a list of values that appear more than once. It includes docstrings and a benchmark section comparing brute force vs optimized duplicate finder. The terminal below shows execution times for two benchmarks: one with 1000 items and another with 2000 items, comparing the performance of the two methods.

```

import random
import time
from typing import List, Any, Optional

def find_duplicates_bruteforce(lst: List[Any]) -> List[Any]:
    """
    Naive duplicate finder (brute force).

    Args:
        lst: List of items (can be any hashable or comparable values).
    Returns:
        A list of values that appear more than once in lst.
        May include duplicates in output if the same duplicate is found multiple times.
    Time complexity: O(n^2) in worst and average cases (two nested loops).
    Space complexity: O(n) for the output list (plus whatever the input uses).
    """
    n = len(lst)
    duplicates = []
    for i in range(n):
        for j in range(i + 1, n):
            if lst[i] == lst[j]:
                # If not already recorded, add to duplicates
    return duplicates

n = 1000 | brute: 0.222449 s | optimized: 0.000631 s
n = 2000 | brute: 0.950304 s | optimized: 0.001714 s

```

VS Code interface showing the Python Debug Console output:

```
PS C:\Users\kadal> cdanaconda3\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\vscodeextensions\ms-python.debugger-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 T 12.4 LAB.py"
Benchmark: brute force vs optimized duplicate finder
n = 1000 | brute: 0.22349 s | optimized: 0.00031 s
n = 2000 | brute: 0.950304 s | optimized: 0.001714 s
n = 5000 | brute: 6.414864 s | optimized: 0.003992 s
```

VS Code interface showing the Python Debug Console output:

```
PS C:\Users\kadal> cdanaconda3\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\vscodeextensions\ms-python.debugger-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 T 12.4 LAB.py"
Benchmark: brute force vs optimized duplicate finder
n = 1000 | brute: 0.22349 s | optimized: 0.00031 s
n = 2000 | brute: 0.950304 s | optimized: 0.001714 s
n = 5000 | brute: 6.414864 s | optimized: 0.003992 s
```

```

C:\> Users > kadal> anaconda3> pkgs > alembic-1.16.4-py313haa95532_0> lib > site-packages > alembic > dd1 > AI T3 12.4 LAB.py > ...
63 def benchmark(fn, data: List[Any], repeat: int = 1) > float:
64     total = 0.0
65     for _ in range(repeat):
66         arr = data.copy()
67         start = time.perf_counter()
68         fn(arr)
69         end = time.perf_counter()
70         total += (end - start)
71     return total / repeat
72
73
74
75
76
77
78 def main():
79     # Test sizes
80     sizes = [1000, 2000, 5000, 10000]
81     # For each size, generate a random list with many duplicates (smaller range)
82     print("Benchmark: brute force vs optimized duplicate finder")
83     for n in sizes:
84         lst = make_random_list(n, value_range = max(10, n // 10))
85         t_brute = benchmark(find_duplicates_bruteforce, lst, repeat=3)
86         t_opt = benchmark(find_duplicates_optimized, lst, repeat=3)
87         print(f'n = {n:6d} | brute: {t_brute:.6f} s | optimized: {t_opt:.6f} s')
88
89     # Demonstrate correctness
90
91
92
93
94
95
96
97
98

```

PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL PORTS QUERY RESULTS

PS C:\Users\Kadal\Anaconda3\pkgs\alembic-1.16.4-py313haa95532_0\lib\site-packages\alembic\dd1 & 'c:\Users\kadal\AppData\Local\Microsoft\WindowsApps\python3.11.exe' 'c:\Users\kadal\vscodeextensions\ms-python.debugger-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\dd1\AI T3 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

Spaces: 4 UTF-8 ⓘ Python 3.11.9 (Microsoft Store) 11:36:33 ENG IN 25-09-2025

```

C:\> Users > kadal> anaconda3> pkgs > alembic-1.16.4-py313haa95532_0> lib > site-packages > alembic > dd1 > AI T3 12.4 LAB.py > ...
78 def main():
79     # Test sizes
80     sizes = [1000, 2000, 5000, 10000]
81     # For each size, generate a random list with many duplicates (smaller range)
82     print("Benchmark: brute force vs optimized duplicate finder")
83     for n in sizes:
84         lst = make_random_list(n, value_range = max(10, n // 10))
85         t_brute = benchmark(find_duplicates_bruteforce, lst, repeat=3)
86         t_opt = benchmark(find_duplicates_optimized, lst, repeat=3)
87         print(f'n = {n:6d} | brute: {t_brute:.6f} s | optimized: {t_opt:.6f} s')
88
89     # Demonstrate correctness
90     sample = [5, 1, 3, 5, 2, 3, 8, 5]
91     print("\nSample list:", sample)
92     print("Brute duplicates : ", find_duplicates_bruteforce(sample))
93     print("Optimized duplicates: ", find_duplicates_optimized(sample))
94
95
96
97
98

```

PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL PORTS QUERY RESULTS

PS C:\Users\Kadal\Anaconda3\pkgs\alembic-1.16.4-py313haa95532_0\lib\site-packages\alembic\dd1 & 'c:\Users\kadal\AppData\Local\Microsoft\WindowsApps\python3.11.exe' 'c:\Users\kadal\vscodeextensions\ms-python.debugger-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\dd1\AI T3 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

Spaces: 4 UTF-8 ⓘ Python 3.11.9 (Microsoft Store) 11:36:44 ENG IN 25-09-2025

```

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]

```

EXPLANATION:

- 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.