

AI ASSISTED CODING  
LAB ASSIGNMENT: 12.4

NAME: SK.NAHEEMA

H.NO : 2403A52059

BATCH NO : 03

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.

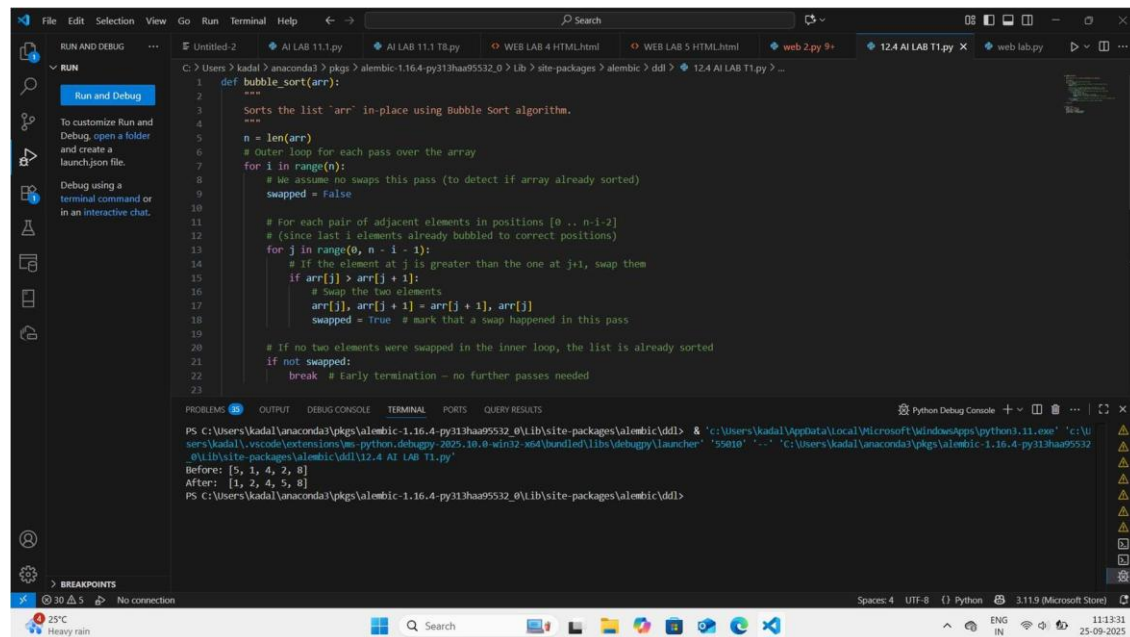
PROMPT:

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:

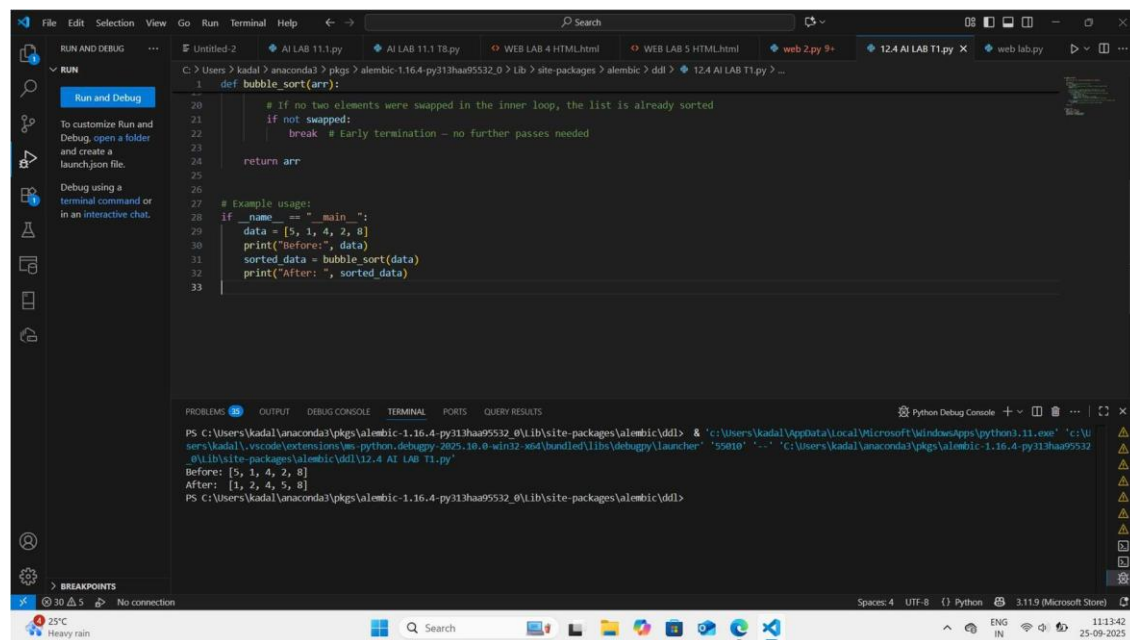


The screenshot shows the VS Code editor with a Python file named `124 AI LAB T1.py`. The code implements a bubble sort algorithm. The `def bubble_sort(arr):` function sorts the list `arr` in-place using the Bubble Sort algorithm. It includes comments explaining the steps: finding the length of the array, looping through each element, comparing adjacent elements, and swapping them if necessary. The algorithm terminates early if no swaps occur in a pass. The output window shows the execution of the function with the input array `[5, 1, 4, 2, 8]` and the output array `[1, 2, 4, 5, 8]`.

```
1 def bubble_sort(arr):
2     """
3     Sorts the list 'arr' in-place using Bubble Sort algorithm.
4     """
5     n = len(arr)
6     # Outer loop for each pass over the array
7     for i in range(n):
8         # We assume no swaps this pass (to detect if array already sorted)
9         swapped = False
10
11        # For each pair of adjacent elements in positions [0 .. n-i-2]
12        # (since last i elements already bubbled to correct positions)
13        for j in range(0, n - i - 1):
14            # If the element at j is greater than the one at j+1, swap them
15            if arr[j] > arr[j + 1]:
16                # Swap the two elements
17                arr[j], arr[j + 1] = arr[j + 1], arr[j]
18                swapped = True # mark that a swap happened in this pass
19
20        # If no two elements were swapped in the inner loop, the list is already sorted
21        if not swapped:
22            break # Early termination - no further passes needed
23
```

PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL PORTS QUERY RESULTS

```
PS C:\Users\kadal\anaconda3\pkgs\alembic-1.16.4-py313ha95532_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\bin\debugpy_launcher' '55010' '-' 'C:\Users\kadal\anaconda3\pkgs\alembic-1.16.4-py313ha95532_0\Lib\site-packages\alembic\ddl\12.4 AI LAB T1.py'
Before: [5, 1, 4, 2, 8]
After: [1, 2, 4, 5, 8]
PS C:\Users\kadal\anaconda3\pkgs\alembic-1.16.4-py313ha95532_0\Lib\site-packages\alembic\ddl>
```



The screenshot shows the VS Code editor with the same Python file `124 AI LAB T1.py`. The code is identical to the previous screenshot, but it includes an example usage section at the bottom. The output window shows the execution of the function with the input array `[5, 1, 4, 2, 8]` and the output array `[1, 2, 4, 5, 8]`.

```
20        # If no two elements were swapped in the inner loop, the list is already sorted
21        if not swapped:
22            break # Early termination - no further passes needed
23
24        return arr
25
26 # Example usage:
27 if __name__ == "__main__":
28     data = [5, 1, 4, 2, 8]
29     print("Before:", data)
30     sorted_data = bubble_sort(data)
31     print("After: ", sorted_data)
32
33
```

PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL PORTS QUERY RESULTS

```
PS C:\Users\kadal\anaconda3\pkgs\alembic-1.16.4-py313ha95532_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\bin\debugpy_launcher' '55010' '-' 'C:\Users\kadal\anaconda3\pkgs\alembic-1.16.4-py313ha95532_0\Lib\site-packages\alembic\ddl\12.4 AI LAB T1.py'
Before: [5, 1, 4, 2, 8]
After: [1, 2, 4, 5, 8]
PS C:\Users\kadal\anaconda3\pkgs\alembic-1.16.4-py313ha95532_0\Lib\site-packages\alembic\ddl>
```

## 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:
  - 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:

The screenshot shows a VS Code editor with a Python file named `LAB 11.1.py`. The code implements a bubble sort algorithm and a function to generate nearly sorted arrays. The terminal output shows the results of comparing bubble sort and insertion sort on arrays of size 1000, 5000, 10000, and 20000. The results show that bubble sort is faster than insertion sort for all tested array sizes.

```
C:\Users\kadal\anaconda3> pip install alembic-1.16.4-py313hae95532.0 > Lib > site-packages > alembic > db > AI T 2 12.4 LAB.py ...
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 def make_almost_sorted(n, num_swaps=5):
35     """
36     Return a list of size n that is nearly sorted,
37     by starting with [0,1,2,...,n-1] and performing a few random adjacent swaps.
38     """
39     lst = list(range(n))
40     for i in range(num_swaps):
41         i = random.randint(0, n - 2)
42         # swap adjacent pair
43         lst[i], lst[i + 1] = lst[i + 1], lst[i]
44     return lst
45
46 PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL PORTS QUERY RESULTS
47 Comparing Bubble Sort vs Insertion Sort on nearly sorted arrays
48
49 n = 1000, swaps = 1 + bubble: 0.003613s, insertion: 0.001743s
50 n = 5000, swaps = 20 + bubble: 0.004632s, insertion: 0.005434s
51 n = 10000, swaps = 1 + bubble: 0.011942s, insertion: 0.012826s
52 n = 10000, swaps = 5 + bubble: 0.008574s, insertion: 0.011675s
53 n = 10000, swaps = 20 + bubble: 0.010897s, insertion: 0.011559s
54
55 Sample (nearly sorted) before: [0, 2, 1, 3, 4, 5, 6, 7, 8, 9, 10, 11, 14, 12, 13, 15, 16, 17, 18, 19]
56 Sorted by bubble : [0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19]
57 Sorted by insertion: [0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19]
58 PS C:\Users\kadal\anaconda3\pkgs\alembic-1.16.4-py313hae95532.0\lib\site-packages\alembic\ddl>
```

The screenshot shows a VS Code editor with a Python file named `LAB 11.1.py`. The code implements a benchmark function to measure the time taken by bubble sort and insertion sort to sort a copy of an array. The terminal output shows the results of comparing bubble sort and insertion sort on arrays of size 1000, 5000, 10000, and 20000. The results show that bubble sort is faster than insertion sort for all tested array sizes.

```
C:\Users\kadal\anaconda3> pip install alembic-1.16.4-py313hae95532.0 > Lib > site-packages > alembic > ddl > AI T 2 12.4 LAB.py ...
51 return lst
52
53 def benchmark(sort_fn, arr):
54     """
55     Time how long sort_fn takes to sort a "copy" of arr.
56     Returns the duration in seconds.
57     """
58     arr_copy = arr.copy()
59     start = time.perf_counter()
60     sort_fn(arr_copy)
61     end = time.perf_counter()
62     return end - start
63
64 def main():
65     # testing parameters
66     test_sizes = [1000, 5000, 10000]
67     swap_counts = [1, 5, 20] # how many adjacent swaps to create disorder
68
69     print("Comparing Bubble Sort vs Insertion Sort on nearly sorted arrays\n")
70     for n in test_sizes:
71         for swaps in swap_counts:
72             ...
73
74 PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL PORTS QUERY RESULTS
75 Comparing Bubble Sort vs Insertion Sort on nearly sorted arrays
76
77 n = 1000, swaps = 1 + bubble: 0.003613s, insertion: 0.001743s
78 n = 5000, swaps = 20 + bubble: 0.004632s, insertion: 0.005434s
79 n = 10000, swaps = 1 + bubble: 0.011942s, insertion: 0.012826s
80 n = 10000, swaps = 5 + bubble: 0.008574s, insertion: 0.011675s
81 n = 10000, swaps = 20 + bubble: 0.010897s, insertion: 0.011559s
82
83 Sample (nearly sorted) before: [0, 2, 1, 3, 4, 5, 6, 7, 8, 9, 10, 11, 14, 12, 13, 15, 16, 17, 18, 19]
84 Sorted by bubble : [0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19]
85 Sorted by insertion: [0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19]
86 PS C:\Users\kadal\anaconda3\pkgs\alembic-1.16.4-py313hae95532.0\lib\site-packages\alembic\ddl>
```

```
def main():
    74     arr = make_almost_sorted(n, num_swaps=swaps)
    75     t_bubble = benchmark(bubble_sort, arr)
    76     t_insert = benchmark(insertion_sort, arr)
    77     print(f"n = {n:6d}, swaps = {swaps:3d} + bubble: {t_bubble:.6f}s, insertion: {t_insert:.6f}s")
    78
    79     # demonstration: show sorting result correctness
    80     sample = make_almost_sorted(20, num_swaps=3)
    81     print("\nsample (nearly sorted) before:", sample)
    82     sorted_by_bubble = bubble_sort(sample.copy())
    83     sorted_by_insert = insertion_sort(sample.copy())
    84     print("sorted by bubble   :", sorted_by_bubble)
    85     print("sorted by insertion:", sorted_by_insert)
    86
    87
    88 if __name__ == "__main__":
    89     main()
    90
```

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: 0.011042s, insertion: 0.012265s
n = 10000, swaps = 5 + bubble: 0.005745s, 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-py311ha95532\_0\lib\site-packages\alembic\ddl>

## 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:



File Edit Selection View Go Run Terminal Help

AI LAB 11.1 TB.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

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.

```
1 import random
2 import time
3 from typing import List, Optional, Any
4
5 def linear_search(arr: List[Any], target: Any) -> Optional[int]:
6     """
7     Linear search (sequential search).
8
9     Args:
10         arr: A list of items (unsorted or sorted).
11         target: The value to search for.
12
13     Returns:
14         The index of target in arr if found, else None.
15
16     Performance:
17         - Worst-case time complexity: O(n)
18         - Best-case time complexity: O(1) (if target is at first position)
19         - Average-case: O(n)
20         - Space complexity: O(1)
21     """
22     for idx, val in enumerate(arr):
23         if val == target:
```

PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL PORTS QUERY RESULTS

Python Debug Console

```
PS C:\Users\kadal\anaconda3\pkgs\alembic-1.16.4-py313ha95532_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-py313ha95532_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
```

30 25°C Heavy rain

Search

ENG IN

11:21:26 25-09-2025

File Edit Selection View Go Run Terminal Help

AI LAB 11.1 TB.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

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.

```
27
28 def binary_search(arr: List[Any], target: Any) -> Optional[int]:
29     """
30     Binary search (assumes arr is sorted in ascending order).
31
32     Args:
33         arr: A sorted list of comparable items.
34         target: The value to search for.
35
36     Returns:
37         The index of target in arr if found, else None.
38
39     Performance:
40         - Worst-case time complexity: O(log n)
41         - Average-case: O(log n)
42         - Best-case: O(1) (if target is at mid initially)
43         - Space complexity: O(1) (iterative)
44
45     Notes:
46         - Works only on sorted lists.
47         - Each iteration halves the search interval.
48     """
49     low = 0
```

PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL PORTS QUERY RESULTS

Python Debug Console

```
PS C:\Users\kadal\anaconda3\pkgs\alembic-1.16.4-py313ha95532_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-py313ha95532_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
```

30 25°C Heavy rain

Search

ENG IN

11:21:50 25-09-2025



The screenshot shows a VS Code editor with a Python file named `AI T3 12.4 LAB.py`. The code implements a binary search algorithm. The `def binary_search(arr: List[Any], target: Any) -> Optional[int]:` function uses a while loop to find the target in a sorted array. The `make_random_list` function generates a list of random integers. The terminal output shows the execution of the code, including the results of linear and binary search on both unsorted and sorted data.

```
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)]
```

Terminal Output:

```
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\bundle\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
```

The screenshot shows a VS Code editor with a Python file named `AI T3 12.4 LAB.py`. The code implements a benchmarking script. The `def make_sorted_list(n: int) -> List[int]:` function generates a sorted list of integers. The `def benchmark(search_fn, arr: List[Any], target: Any, repeat: int = 1) -> float:` function measures the time taken by a search function to find a target in an array. The `def main():` function tests the benchmarking script on both unsorted and sorted data.

```
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
```

Terminal Output:

```
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\bundle\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
```

```
def main():  
    # Create unsorted list and sorted list  
    unsorted = make_random_list(n)  
    sorted_list = sorted(unsorted)  
    # or simply: sorted_list = make_sorted_list(n)  
  
    # Pick a target (choose something that is in the list)  
    # For safety, pick from sorted_list  
    target_value = sorted_list[n // 2]  
  
    print("=== Testing on Unsorted Data ===")  
    t_lin = benchmark(linear_search, unsorted, target_value, repeat=5)  
    t_bin_unsorted = None  
    try:  
        # We can "call" binary_search on unsorted data, but it will give incorrect result  
        t_bin_unsorted = benchmark(binary_search, unsorted, target_value, repeat=5)  
    except Exception as e:  
        print("Binary search on unsorted data failed:", e)  
        print(f"Linear search time (unsorted): {t_lin:.6f} s")  
        if t_bin_unsorted is not None:  
            print(f"Binary search time (unsorted, invalid): {t_bin_unsorted:.6f} s (result may be wrong)")  
  
    print("=== Testing on Sorted Data ===")  
    t_lin_sorted = benchmark(linear_search, sorted_list, target_value, repeat=5)  
    t_bin_sorted = benchmark(binary_search, sorted_list, target_value, repeat=5)  
    print(f"Linear search time (sorted list): {t_lin_sorted:.6f} s")  
    print(f"Binary search time (sorted list): {t_bin_sorted:.6f} s")  
  
    # Validate correctness  
    idx_lin = linear_search(sorted_list, target_value)  
    idx_bin = binary_search(sorted_list, target_value)  
    print(f"Correctness check: target = {target_value}")  
    print(f"Linear search found at index: {idx_lin}")  
    print(f"Binary search found at index: {idx_bin}")  
  
    print("=== Summary & when to Prefer Binary Search ===")  
    print("1. Linear search works whether or not the data is sorted; binary search requires sorted data.")  
    print("2. Linear search is O(n), while binary search is O(log n) on sorted data.")  
    print("3. For small arrays or one-time searches, the overhead of binary search may not pay off.")  
    print("4. If you perform many searches on the same sorted list, binary search is much more efficient overall.")  
    print("5. If the data is unsorted and you only search once, linear search may be preferable (to avoid sorting).")  
    print("6. In practice, for large n and many searches, binary search is the better choice. (Binary search is preferred for large sorted datasets.)")  
  
if __name__ == "__main__":  
    main()
```

PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL PORTS QUERY RESULTS

Python Debug Console

PS C:\Users\kadal\anaconda3\pkgs\alembic-1.16.4-py311ha95532\_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\bundle\libs\debugpy\launcher' '55118' '-.' 'c:\Users\kadal\anaconda3\pkgs\alembic-1.16.4-py311ha95532\_0\Lib\site-packages\alembic\ddl\AT 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

```
print("=== Testing on Sorted Data ===")  
t_lin_sorted = benchmark(linear_search, sorted_list, target_value, repeat=5)  
t_bin_sorted = benchmark(binary_search, sorted_list, target_value, repeat=5)  
print(f"Linear search time (sorted list): {t_lin_sorted:.6f} s")  
print(f"Binary search time (sorted list): {t_bin_sorted:.6f} s")  
  
# Validate correctness  
idx_lin = linear_search(sorted_list, target_value)  
idx_bin = binary_search(sorted_list, target_value)  
print(f"Correctness check: target = {target_value}")  
print(f"Linear search found at index: {idx_lin}")  
print(f"Binary search found at index: {idx_bin}")  
  
print("=== Summary & when to Prefer Binary Search ===")  
print("1. Linear search works whether or not the data is sorted; binary search requires sorted data.")  
print("2. Linear search is O(n), while binary search is O(log n) on sorted data.")  
print("3. For small arrays or one-time searches, the overhead of binary search may not pay off.")  
print("4. If you perform many searches on the same sorted list, binary search is much more efficient overall.")  
print("5. If the data is unsorted and you only search once, linear search may be preferable (to avoid sorting).")  
print("6. In practice, for large n and many searches, binary search is the better choice. (Binary search is preferred for large sorted datasets.)")  
  
if __name__ == "__main__":  
    main()
```

PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL PORTS QUERY RESULTS

Python Debug Console

PS C:\Users\kadal\anaconda3\pkgs\alembic-1.16.4-py311ha95532\_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\bundle\libs\debugpy\launcher' '55118' '-.' 'c:\Users\kadal\anaconda3\pkgs\alembic-1.16.4-py311ha95532\_0\Lib\site-packages\alembic\ddl\AT 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

```
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.) 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:

The screenshot displays the Visual Studio Code (VS Code) editor interface on a Windows 11 desktop. The main editor window shows a Python file named `web lab.py` with the following code:

```

1 import random
2
3
4 # --- Quick Sort Implementation ---
5 def quick_sort_recursive(arr, low, high):
6     """
7     Recursively sorts an array using the Quick Sort algorithm.
8     """
9     if low < high:
10         pi = partition(arr, low, high)
11         quick_sort_recursive(arr, low, pi - 1)
12         quick_sort_recursive(arr, pi + 1, high)
13
14 def partition(arr, low, high):
15     """
16     Partitions the array and returns the pivot index.
17     """
18     pivot = arr[high]
19     i = low - 1
20     for j in range(low, high):
21         if arr[j] <= pivot:
22             i += 1
23             arr[i], arr[j] = arr[j], arr[i]

```

The left sidebar contains three panels: **VARIABLES**, **WATCH**, and **CALL STACK**. The **CALL STACK** panel shows the function `quick_sort_recursive` is currently running. The bottom status bar indicates the file is at **Ln 135, Col 21** and the Python version is **3.11.9 (Microsoft Store)**. The system tray at the bottom shows the date and time as **25-09-2025 11:28:59**.

```
File Edit Selection View Go Run Terminal Help
C:\Users\kadal> anaconda3> pip install alembic-1.16.4-py311ha95532.0 --lib=site-packages> alembic> db> AI T4 12.4 LAB.py> ...

25     return i + 1
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
PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL PORTS QUERY RESULTS
Python Debug Console
c:\cross\tools\python3.11.exe "c:\Users\kadal\.vscode\extensions\ms-python.debugpy-2025.18.0-win32-x64\bundle\libs\debugpy\launcher" "55214" "-" "c:\Users\kadal\an
conda\pkgs\alembic-1.16.4-py311ha95532.0\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
BROKERSPOINTS
```

```
File Edit Selection View Go Run Terminal Help
C:\Users\kadal> anaconda3> pip install alembic-1.16.4-py311ha95532.0 --lib=site-packages> alembic> ddl> AI T4 12.4 LAB.py> ...

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 ---

PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL PORTS QUERY RESULTS
Python Debug Console
c:\cross\tools\python3.11.exe "c:\Users\kadal\.vscode\extensions\ms-python.debugpy-2025.18.0-win32-x64\bundle\libs\debugpy\launcher" "55214" "-" "c:\Users\kadal\an
conda\pkgs\alembic-1.16.4-py311ha95532.0\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
BROKERSPOINTS
```



```
File Edit Selection View Go Run Terminal Help
LAB 4 HTML.html WEB LAB 5 HTML.html web 2.py 9+ AI T 2 12.4 LAB.py AI T 3 12.4 LAB.py AI T 4 12.4 LAB.py web lab.py

C:\Users\kadal> kadal > anaconda3 > pkgs > alembic-1.16.4-py313haa95532_0 > Lib > site-packages > alembic > ddl > AI T 4 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
PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL PORTS QUERY RESULTS
Python Debug Console
c:\Users\kadal\vscode\extensions\ms-python.debugpy-2025.10.0-win32-x64\bundled\libs\debugpy\launcher '55214' ... 'C:\Users\kadal\an
aconda3\pkgs\alembic-1.16.4-py313haa95532_0\Lib\site-packages\alembic\ddl\AI T 4 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
```

```
C:\Users\kadal> anaconda3 > pkgs > alembic-1.16.4-py313haa95532_0 > Lib > site-packages > alembic > ddl > AI T 4 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     end_time = time.time()
115     print("Merge Sort time: {:.4f} seconds".format(end_time - start_time))
116
PROBLEMS 35 OUTPUT DEBUG CONSOLE TERMINAL PORTS QUERY RESULTS
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 T 4 12.4 LAB.py'
--- Performance Comparison (List Size: 50000) ---
--- Random List ---
Quick Sort time: 1.0666 seconds
Merge Sort time: 1.4798 seconds
```

```

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\bundle\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

```

- 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.

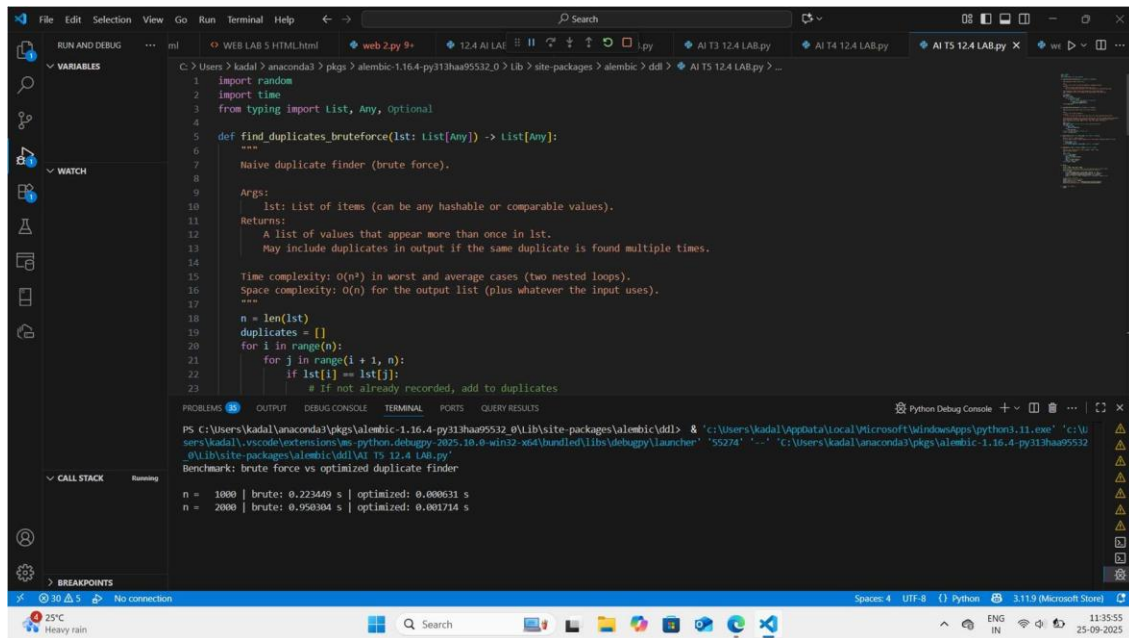
## PROMPT:

### 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 a Python IDE with a dark theme. The main editor displays a function `find_duplicates_bruteforce` that takes a list and returns a list of duplicates using a nested loop approach. The function includes docstrings for arguments, returns, and time/space complexity. Below the code, the `TERMINAL` tab shows the execution of a benchmark script comparing the brute-force method with an optimized version for input sizes  $n=1000$  and  $n=2000$ .

```
1 import random
2 import time
3 from typing import List, Any, Optional
4
5 def find_duplicates_bruteforce(lst: List[Any]) -> List[Any]:
6     """
7     Naive duplicate finder (brute force).
8
9     Args:
10         lst: List of items (can be any hashable or comparable values).
11     Returns:
12         A list of values that appear more than once in lst.
13         May include duplicates in output if the same duplicate is found multiple times.
14
15     Time complexity:  $O(n^2)$  in worst and average cases (two nested loops).
16     Space complexity:  $O(n)$  for the output list (plus whatever the input uses).
17     """
18     n = len(lst)
19     duplicates = []
20     for i in range(n):
21         for j in range(i + 1, n):
22             if lst[i] == lst[j]:
23                 # If not already recorded, add to duplicates
```

PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL PORTS QUERY RESULTS

```
PS C:\Users\kadal\anaconda3\pkgs\alembic-1.16.4-py313ha95532_0\lib\site-packages\alembic\ddl> & "c:\Users\kadal\AppData\Local\Microsoft\WindowsApps\python3.11.exe" "c:\Users\kadal\vscode\extensions\ms-python.debugpy-2025.18.0-win32-x64\bin\debugpy_launcher" "55274" "--" "c:\Users\kadal\anaconda3\pkgs\alembic-1.16.4-py313ha95532_0\lib\site-packages\alembic\ddl\AI TS 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
```

Space: 4 UTF-8 Python 3.11.9 (Microsoft Store)

25°C Heavy rain 11:35:55 25-09-2025

```
File Edit Selection View Go Run Terminal Help
C:\Users\kadal\anaconda3> pip install alembic==1.16.4 py313ha95532.0 --lib=site-packages> alembic> db> AI T5 12.4 LAB.py> ...
def find_duplicates_bruteforce(lst: List[Any]) -> List[Any]:
    24     if lst[i] not in duplicates:
    25         duplicates.append(lst[i])
    26     return duplicates
    27
    28
def find_duplicates_optimized(lst: List[Any]) -> List[Any]:
    29     """
    30     Optimized duplicate finder using a set & seen structure.
    31
    32     Args:
    33         lst: List of items (hashable).
    34     Returns:
    35         A list of values that appear more than once in lst, each duplicate exactly once.
    36
    37     Time complexity (average): O(n) - one pass through list, with O(1) average set/dict operations.
    38     Worst-case (hash collisions, degenerate): still higher, but in practice close to O(n).
    39     Space complexity: O(n) extra space for the 'seen' and 'dups' sets/lists.
    40
    41     """
    42     seen = set()
    43     duplicates = set() # we use a set to avoid repeated duplicates
    44     for item in lst:
    45         if item in seen:
    46             duplicates.add(item)
    47         else:
    48             seen.add(item)
    49     # Return as list (if needed, sorted or not)
    50     return list(duplicates)
    51
    52
def make_random_list(n: int, value_range: int = None) -> List[int]:
    53     """
    54     Return a list of n random integers.
    55     If value_range is given, random integers in [0, value_range); otherwise default 0 to n*2.
    56
    57     """
    58     if value_range is None:
    59         value_range = n * 2
    60     return [random.randint(0, value_range - 1) for _ in range(n)]
    61
    62
def benchmark(fn, data: List[Any], repeat: int = 1) -> float:
    63     """
    64     Time how long "fn(data_copy)" takes, repeated "repeat" times.
    65     Returns average seconds.
    66
    67     """
    68     total_time = 0
    69     for _ in range(repeat):
    70         start_time = time.perf_counter()
    71         fn(data.copy())
    72         end_time = time.perf_counter()
    73         total_time += end_time - start_time
    74     return total_time / repeat

PS C:\Users\kadal\anaconda3\pkgs\alembic-1.16.4-py313ha95532.0\lib\site-packages\alembic> & 'c:\Users\kadal\AppData\Local\Microsoft\WindowsApps\python3.11.exe' 'c:\Users\kadal\vscode\extensions\ms-python.debugpy-2025.10.0-win32-x64\bin\debugpy\launcher' '55274' '-' 'c:\Users\kadal\anaconda3\pkgs\alembic-1.16.4-py313ha95532.0\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
```

```
File Edit Selection View Go Run Terminal Help
C:\Users\kadal\anaconda3> pip install alembic==1.16.4 py313ha95532.0 --lib=site-packages> alembic> ddl> AI T5 12.4 LAB.py> ...
def find_duplicates_optimized(lst: List[Any]) -> List[Any]:
    47     else:
    48         seen.add(item)
    49     # Return as list (if needed, sorted or not)
    50     return list(duplicates)
    51
    52
def make_random_list(n: int, value_range: int = None) -> List[int]:
    53     """
    54     Return a list of n random integers.
    55     If value_range is given, random integers in [0, value_range); otherwise default 0 to n*2.
    56
    57     """
    58     if value_range is None:
    59         value_range = n * 2
    60     return [random.randint(0, value_range - 1) for _ in range(n)]
    61
    62
def benchmark(fn, data: List[Any], repeat: int = 1) -> float:
    63     """
    64     Time how long "fn(data_copy)" takes, repeated "repeat" times.
    65     Returns average seconds.
    66
    67     """
    68     total_time = 0
    69     for _ in range(repeat):
    70         start_time = time.perf_counter()
    71         fn(data.copy())
    72         end_time = time.perf_counter()
    73         total_time += end_time - start_time
    74     return total_time / repeat

PS C:\Users\kadal\anaconda3\pkgs\alembic-1.16.4-py313ha95532.0\lib\site-packages\alembic> & 'c:\Users\kadal\AppData\Local\Microsoft\WindowsApps\python3.11.exe' 'c:\Users\kadal\vscode\extensions\ms-python.debugpy-2025.10.0-win32-x64\bin\debugpy\launcher' '55274' '-' 'c:\Users\kadal\anaconda3\pkgs\alembic-1.16.4-py313ha95532.0\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
```

```
C:\Users\kadal> anaconda3 > pkgs > alembic-1.16.4-py313ha95532_0\lib\site-packages\alembic> ddl > AI T5 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 def main():  
74     # Test sizes  
75     sizes = [1000, 2000, 5000, 10000]  
76     # For each size, generate a random list with many duplicates (smaller range)  
77     print("Benchmark: brute force vs optimized duplicate finder\n")  
78     for n in sizes:  
79         lst = make_random_list(n, value_range = max(10, n // 10))  
80         t_brute = benchmark(find_duplicates_bruteforce, lst, repeat=3)  
81         t_opt = benchmark(find_duplicates_optimized, lst, repeat=3)  
82         print(f"n = {n:6d} | brute: {t_brute:.6f} s | optimized: {t_opt:.6f} s")  
83  
84     # Demonstrate correctness  
85  
86  
87  
88  
89  
90  
91  
92  
93  
94  
95  
96  
97  
98  
99  
100  
101  
102  
103  
104  
105  
106  
107  
108  
109  
110  
111  
112  
113  
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  
560  
561  
562  
563  
564  
565  
566  
567  
568  
569  
570  
571  
572  
573  
574  
575  
576  
577  
578  
579  
580  
581  
582  
583  
584  
585  
586  
587  
588  
589  
590  
591  
592  
593  
594  
595  
596  
597  
598  
599  
600  
601  
602  
603  
604  
605  
606  
607  
608  
609  
610  
611  
612  
613  
614  
615  
616  
617  
618  
619  
620  
621  
622  
623  
624  
625  
626  
627  
628  
629  
630  
631  
632  
633  
634  
635  
636  
637  
638  
639  
640  
641  
642  
643  
644  
645  
646  
647  
648  
649  
650  
651  
652  
653  
654  
655  
656  
657  
658  
659  
660  
661  
662  
663  
664  
665  
666  
667  
668  
669  
670  
671  
672  
673  
674  
675  
676  
677  
678  
679  
680  
681  
682  
683  
684  
685  
686  
687  
688  
689  
690  
691  
692  
693  
694  
695  
696  
697  
698  
699  
700  
701  
702  
703  
704  
705  
706  
707  
708  
709  
710  
711  
712  
713  
714  
715  
716  
717  
718  
719  
720  
721  
722  
723  
724  
725  
726  
727  
728  
729  
730  
731  
732  
733  
734  
735  
736  
737  
738  
739  
740  
741  
742  
743  
744  
745  
746  
747  
748  
749  
750  
751  
752  
753  
754  
755  
756  
757  
758  
759  
760  
761  
762  
763  
764  
765  
766  
767  
768  
769  
770  
771  
772  
773  
774  
775  
776  
777  
778  
779  
780  
781  
782  
783  
784  
785  
786  
787  
788  
789  
790  
791  
792  
793  
794  
795  
796  
797  
798  
799  
800  
801  
802  
803  
804  
805  
806  
807  
808  
809  
810  
811  
812  
813  
814  
815  
816  
817  
818  
819  
820  
821  
822  
823  
824  
825  
826  
827  
828  
829  
830  
831  
832  
833  
834  
835  
836  
837  
838  
839  
840  
841  
842  
843  
844  
845  
846  
847  
848  
849  
850  
851  
852  
853  
854  
855  
856  
857  
858  
859  
860  
861  
862  
863  
864  
865  
866  
867  
868  
869  
870  
871  
872  
873  
874  
875  
876  
877  
878  
879  
880  
881  
882  
883  
884  
885  
886  
887  
888  
889  
890  
891  
892  
893  
894  
895  
896  
897  
898  
899  
900  
901  
902  
903  
904  
905  
906  
907  
908  
909  
910  
911  
912  
913  
914  
915  
916  
917  
918  
919  
920  
921  
922  
923  
924  
925  
926  
927  
928  
929  
930  
931  
932  
933  
934  
935  
936  
937  
938  
939  
940  
941  
942  
943  
944  
945  
946  
947  
948  
949  
950  
951  
952  
953  
954  
955  
956  
957  
958  
959  
960  
961  
962  
963  
964  
965  
966  
967  
968  
969  
970  
971  
972  
973  
974  
975  
976  
977  
978  
979  
980  
981  
982  
983  
984  
985  
986  
987  
988  
989  
990  
991  
992  
993  
994  
995  
996  
997  
998  
999  
1000  
1001  
1002  
1003  
1004  
1005  
1006  
1007  
1008  
1009  
1010  
1011  
1012  
1013  
1014  
1015  
1016  
1017  
1018  
1019  
1020  
1021  
1022  
1023  
1024  
1025  
1026  
1027  
1028  
1029  
1030  
1031  
1032  
1033  
1034  
1035  
1036  
1037  
1038  
1039  
1040  
1041  
1042  
1043  
1044  
1045  
1046  
1047  
1048  
1049  
1050  
1051  
1052  
1053  
1054  
1055  
1056  
1057  
1058  
1059  
1060  
1061  
1062  
1063  
1064  
1065  
1066  
1067  
1068  
1069  
1070  
1071  
1072  
1073  
1074  
1075  
1076  
1077  
1078  
1079  
1080  
1081  
1082  
1083  
1084  
1085  
1086  
1087  
1088  
1089  
1090  
1091  
1092  
1093  
1094  
1095  
1096  
1097  
1098  
1099  
1100  
1101  
1102  
1103  
1104  
1105  
1106  
1107  
1108  
1109  
1110  
1111  
1112  
1113  
1114  
1115  
1116  
1117  
1118  
1119  
1120  
1121  
1122  
1123  
1124  
1125  
1126  
1127  
1128  
1129  
1130  
1131  
1132  
1133  
1134  
1135  
1136  
1137  
1138  
1139  
1140  
1141  
1142  
1143  
1144  
1145  
1146  
1147  
1148  
1149  
1150  
1151  
1152  
1153  
1154  
1155  
1156  
1157  
1158  
1159  
1160  
1161  
1162  
1163  
1164  
1165  
1166  
1167  
1168  
1169  
1170  
1171  
1172  
1173  
1174  
1175  
1176  
1177  
1178  
1179  
1180  
1181  
1182  
1183  
1184  
1185  
1186  
1187  
1188  
1189  
1190  
1191  
1192  
1193  
1194  
1195  
1196  
1197  
1198  
1199  
1200  
1201  
1202  
1203  
1204  
1205  
1206  
1207  
1208  
1209  
1210  
1211  
1212  
1213  
1214  
1215  
1216  
1217  
1218  
1219  
1220  
1221  
1222  
1223  
1224  
1225  
1226  
1227  
1228  
1229  
1230  
1231  
1232  
1233  
1234  
1235  
1236  
1237  
1238  
1239  
1240  
1241  
1242  
1243  
1244  
1245  
1246  
1247  
1248  
1249  
1250  
1251  
1252  
1253  
1254  
1255  
1256  
1257  
1258  
1259  
1260  
1261  
1262  
1263  
1264  
1265  
1266  
1267  
1268  
1269  
1270  
1271  
1272  
1273  
1274  
1275  
1276  
1277  
1278  
1279  
1280  
1281  
1282  
1283  
1284  
1285  
1286  
1287  
1288  
1289  
1290  
1291  
1292  
1293  
1294  
1295  
1296  
1297  
1298  
1299  
1300  
1301  
1302  
1303  
1304  
1305  
1306  
1307  
1308  
1309  
1310  
1311  
1312  
1313  
1314  
1315  
1316  
1317  
1318  
1319  
1320  
1321  
1322  
1323  
1324  
1325  
1326  
1327  
1328  
1329  
1330  
1331  
1332  
1333  
1334  
1335  
1336  
1337  
1338  
1339  
1340  
1341  
1342  
1343  
1344  
1345  
1346  
1347  
1348  
1349  
1350  
1351  
1352  
1353  
1354  
1355  
1356  
1357  
1358  
1359  
1360  
1361  
1362  
1363  
1364  
1365  
1366  
1367  
1368  
1369  
1370  
1371  
1372  
1373  
1374  
1375  
1376  
1377  
1378  
1379  
1380  
1381  
1382  
1383  
1384  
1385  
1386  
1387  
1388  
1389  
1390  
1391  
1392  
1393  
1394  
1395  
1396  
1397  
1398  
1399  
1400  
1401  
1402  
1403  
1404  
1405  
1406  
1407  
1408  
1409  
1410  
1411  
1412  
1413  
1414  
1415  
1416  
1417  
1418  
1419  
1420  
1421  
1422  
1423  
1424  
1425  
1426  
1427  
1428  
1429  
1430  
1431  
1432  
1433  
1434  
1435  
1436  
1437  
1438  
1439  
1440  
1441  
1442  
1443  
1444  
1445  
1446  
1447  
1448  
1449  
1450  
1451  
1452  
1453  
1454  
1455  
1456  
1457  
1458  
1459  
1460  
1461  
1462  
1463  
1464  
1465  
1466  
1467  
1468  
1469  
1470  
1471  
1472  
1473  
1474  
1475  
1476  
1477  
1478  
1479  
1480  
1481  
1482  
1483  
1484  
1485  
1486  
1487  
1488  
1489  
1490  
1491  
1492  
1493  
1494  
1495  
1496  
1497  
1498  
1499  
1500  
1501  
1502  
1503  
1504  
1505  
1506  
1507  
1508  
1509  
1510  
1511  
1512  
1513  
1514  
1515  
1516  
1517  
1518  
1519  
1520  
1521  
1522  
1523  
1524  
1525  
1526  
1527  
1528  
1529  
1530  
1531  
1532  
1533  
1534  
1535  
1536  
1537  
1538  
1539  
1540  
1541  
1542  
1543  
1544  
1545  
1546  
1547  
1548  
1549  
1550  
1551  
1552  
1553  
1554  
1555  
1556  
1557  
1558  
1559  
1560  
1561  
1562  
1563  
1564  
1565  
1566  
1567  
1568  
1569  
1570  
1571  
1572  
1573  
1574  
1575  
1576  
1577  
1578  
1579  
1580  
1581  
1582  
1583  
1584  
1585  
1586  
1587  
1588  
1589  
1590  
1591  
1592  
1593  
1594  
1595  
1596  
1597  
1598  
1599  
1600  
1601  
1602  
1603  
1604  
1605  
1606  
1607  
1608  
1609  
1610  
1611  
1612  
1613  
1614  
1615  
1616  
1617  
1618  
1619  
1620  
1621  
1622  
1623  
1624  
1625  
1626  
1627  
1628  
1629  
1630  
1631  
1632  
1633  
1634  
1635  
1636  
1637  
1638  
1639  
1640  
1641  
1642  
1643  
1644  
1645  
1646  
1647  
1648  
1649  
1650  
1651  
1652  
1653  
1654  
1655  
1656  
1657  
1658  
1659  
1660  
1661  
1662  
1663  
1664  
1665  
1666  
1667  
1668  
1669  
1670  
1671  
1672  
1673  
1674  
1675  
1676  
1677  
1678  
1679  
1680  
1681  
1682  
1683  
1684  
1685  
1686  
1687  
1688  
1689  
1690  
1691  
1692  
1693  
1694  
1695  
1696  
1697  
1698  
1699  
1700  
1701  
1702  
1703  
1704  
1705  
1706  
1707  
1708  
1709  
1710  
1711  
1712  
1713  
1714  
1715  
1716  
1717  
1718  
1719  
1720  
1721  
1722  
1723  
1724  
1725  
1726  
1727  
1728  
1729  
1730  
1731  
1732  
1733  
1734  
1735  
1736  
1737  
1738  
1739  
1740  
1741  
1742  
1743  
1744  
1745  
1746  
1747  
1748  
1749  
1750  
1751  
1752  
1753  
1754  
1755  
1756  
1757  
1758  
1759  
1760  
1761  
1762  
1763  
1764  
1765  
1766  
1767  
1768  
1769  
1770  
1771  
1772  
1773  
1774  
1775  
1776  
1777  
1778  
1779  
1780  
1781  
1782  
1783  
1784  
1785  
1786  
1787  
1788  
1789  
1790  
1791  
1792  
1793  
1794  
1795  
1796  
1797  
1798  
1799  
1800  
1801  
1802  
1803  
1804  
1805  
1806  
1807  
1808  
1809  
1810  
1811  
1812  
1813  
1814  
1815  
1816  
1817  
1818  
1819  
1820  
1821  
1822  
1823  
1824  
1825  
1826  
1827  
1828  
1829  
1830  
1831  
1832  
1833  
1834  
1835  
1836  
1837  
1838  
1839  
1840  
1841  
1842  
1843  
1844  
1845  
1846  
1847  
1848  
1849  
1850  
1851  
1852  
1853  
1854  
1855  
1856  
1857  
1858  
1859  
1860  
1861  
1862  
1863  
1864  
1865  
1866  
1867  
1868  
1869  
1870  
1871  
1872  
1873  
1874  
1875  
1876  
1877  
1878  
1879  
1880  
1881  
1882  
1883  
1884  
1885  
1886  
1887  
1888  
1889  
1890  
1891  
1892  
1893  
1894  
1895  
1896  
1897  
1898  
1899  
1900  
1901  
1902  
1903  
1904  
1905  
1906  
1907  
1908  
1909  
1910  
1911  
1912  
1913  
1914  
1915  
1916  
1917  
1918  
1919  
1920  
1921  
1922  
1923  
1924  
1925  
1926  
1927  
1928  
1929  
1930  
1931  
1932  
1933  
1934  
1935  
1936  
1937  
1938  
1939  
1940  
1941  
1942  
1943  
1944  
1945  
1946  
1947  
1948  
1949  
1950  
1951  
1952  
1953  
1954  
1955  
1956  
1957  
1958  
1959  
1960  
1961  
1962  
1963  
1964  
1965  
1966  
1967  
1968  
1969  
1970  
1971  
1972  
1973  
1974  
1975  
1976  
1977  
1978  
1979  
1980  
1981  
1982  
1983  
1984  
1985  
1986  
1987  
1988  
1989  
1990  
1991  
1992  
1993  
1994  
1995  
1996  
1997  
1998  
1999  
2000  
2001  
2002  
2003  
2004  
2005  
2006  
2007  
2008  
2009  
2010  
2011  
2012  
2013  
2014  
2015  
2016  
2017  
2018  
2019  
2020  
2021  
2022  
2023  
2024  
2025  
2026  
2027  
2028  
2029  
2030  
2031  
2032  
2033  
2034  
2035  
2036  
2037  
2038  
2039  
2040  
2041  
2042  
2043  
2044  
2045  
2046  
2047  
2048  
2049  
2050  
2051  
2052  
2053  
2054  
2055  
2056  
2057  
2058  
2059  
2060  
2061  
2062  
2063  
2064  
2065  
2066  
2067  
2068  
2069  
2070  
2071  
2072  
2073  
2074  
2075  
2076  
2077  
2078  
2079  
2080  
2081  
2082  
2083  
2084  
2085  
2086  
2087  
2088  
2089  
2090  
2091  
2092  
2093  
2094  
2095  
2096  
2097  
2098  
2099  
2100  
2101  
2102  
2103  
2104  
2105  
2106  
2107  
2108  
2109  
2110  
2111  
2112  
2113  
2114  
2115  
2116  
2117  
2118  
2119  
2120  
2121  
2122  
2123  
2124  
2125  
2126  
2127  
2128  
2129  
2130  
2131  
2132  
2133  
2134  
2135  
2136  
2137  
2138  
2139  
2140  
2141  
2142  
2143  
2144  
2145  
2146  
2147  
2148  
2149  
2150  
2151  
2152  
2153  
2154  
2155  
2156  
2157  
2158  
2159  
2160  
2161  
2162  
2163  
2164  
2165  
2166  
2167  
2168  
2169  
2170  
2171  
2172  
2173  
2174  
2175  
2176  
2177  
2178  
2179  
2180  
2181  
2182  
2183  
2184  
2185  
2186  
2187  
2188  
2189  
2190  
2191  
2192  
2193  
2194  
2195  
2196  
2197  
2198  
2199  
2200  
2201  
2202  
2203  
2204  
2205  
2206  
2207  
2208  
2209  
2210  
2211  
2212  
2213  
2214  
2215  
2216  
2217  
2218  
2219  

```

## 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.