ASSIGNMENT-12.4

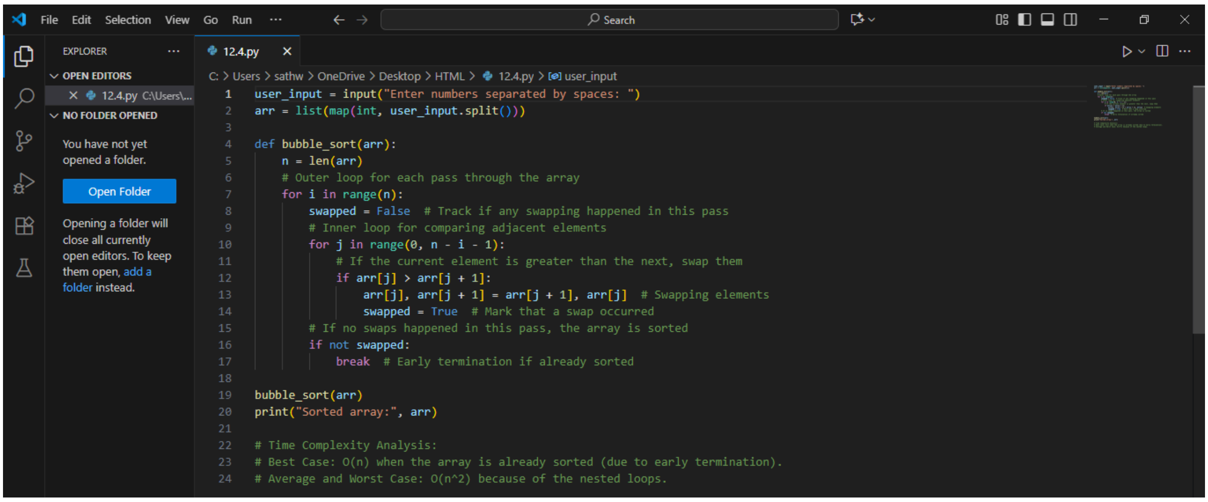
2403A52060

BATCH-03

TASK-01

PROMPT: **Implementing Bubble Sort with AI Comments** Write a python code which performs the bubble short and the code must contain the following 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



```python
user_input = input("Enter numbers separated by spaces: ")
arr = list(map(int, user_input.split()))

def bubble_sort(arr):
    n = len(arr)
    # Outer loop for each pass through the array
    for i in range(n):
        swapped = False  # Track if any swapping happened in this pass
        # Inner loop for comparing adjacent elements
        for j in range(0, n - i - 1):
            # If the current element is greater than the next, swap them
            if arr[j] > arr[j + 1]:
                arr[j], arr[j + 1] = arr[j + 1], arr[j]  # Swapping elements
                swapped = True  # Mark that a swap occurred
        # If no swaps happened in this pass, the array is sorted
        if not swapped:
            break  # Early termination if already sorted

bubble_sort(arr)
print("Sorted array:", arr)

# Time Complexity Analysis:
# Best Case: O(n) when the array is already sorted (due to early termination).
# Average and Worst Case: O(n^2) because of the nested loops.
```

OUTPUT:

Enter numbers separated by spaces: 12 34 21 56 60

Sorted array: [12, 21, 34, 56, 60]

EXPLANATION:

• def bubble_sort(arr):: This line defines a function named bubble_sort that takes one argument, arr, which is the list to be sorted. • n = len(arr): This gets the number of elements in the input list and stores it in the variable n. • for i in range(n):: This is the outer loop. It iterates n times. In each iteration, the largest unsorted element "bubbles up" to its correct position at the end of the unsorted portion of the list. • for j in range(0, n - i - 1):: This is the inner loop. It traverses the unsorted portion of the array (from the beginning up to the i-th element from the end, which is already sorted).

TASK-02

PROMPT: **Optimizing Bubble Sort → Insertion Sort** write a code which

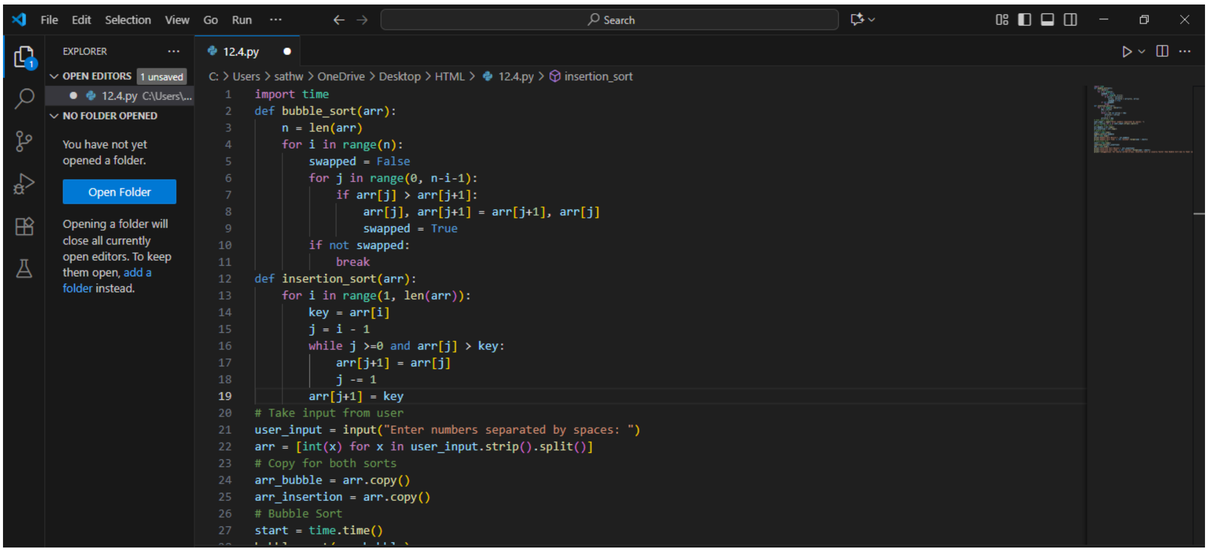
Provide Bubble Sort code and the code suggests a

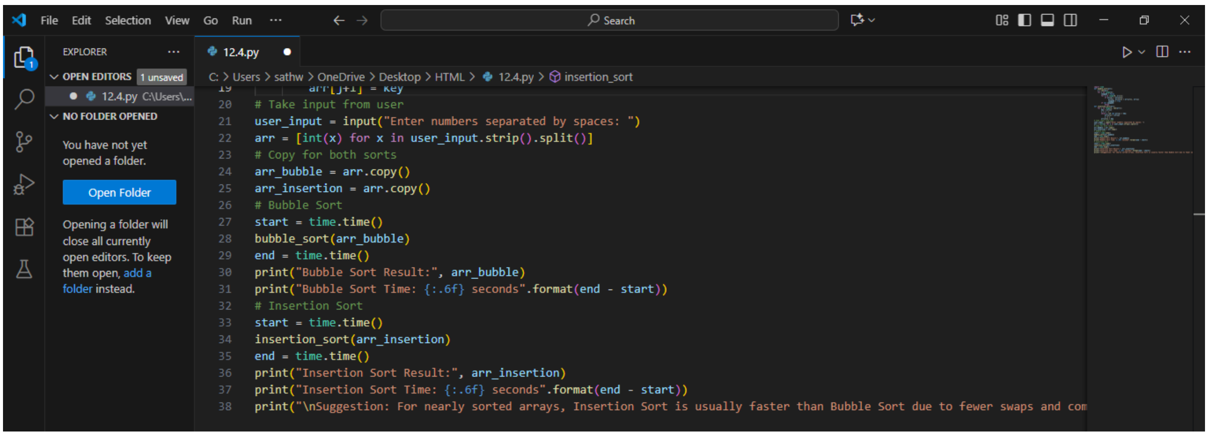
more efficient algorithm for partially sorted arrays which includes the Instructions:

o Students implement Bubble Sort first.

o Ask AI to suggest an alternative (Insertion Sort).

o Compare performance on nearly sorted input and input must be taken from user.





OUTPUT:

Enter numbers separated by spaces: 23 78 12 3

Bubble Sort Result: [3, 12, 23, 78]

Bubble Sort Time: 0.000057 seconds

Insertion Sort Result: [3, 12, 23, 78]

Insertion Sort Time: 0.000021 seconds

Suggestion: For nearly sorted arrays, Insertion Sort is usually faster than Bubble Sort due to fewer swaps and comparisons.

EXPLANATION:

• def insertion_sort(arr):: This line defines a function named insertion_sort that takes one argument, arr, which is the list to be sorted. • for i in range(1, len(arr)):: This is the main loop. It iterates through the list starting from the second element (i = 1) up to the last element. It considers each element arr[i] as the "key" to be inserted into the already sorted portion of the array to its left. • key = arr[i]: This line stores the current element being considered for insertion in the key variable. • j = i - 1: This initializes a variable j to the index of the last element in the sorted portion of the array (to the left of the key).

TASK-03

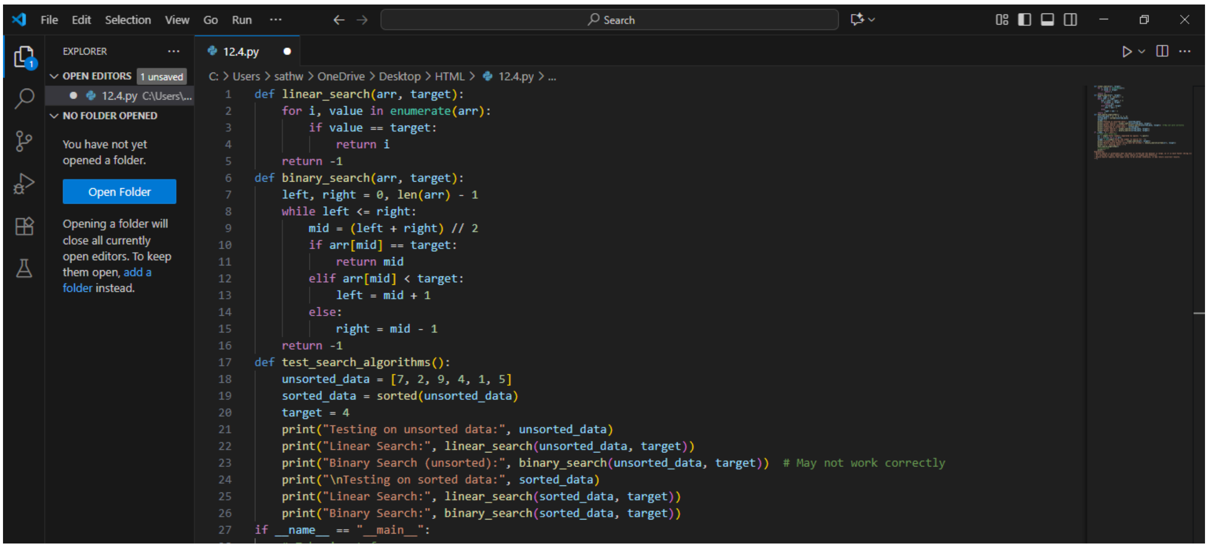
PROMPT: **Binary Search vs Linear Search** write a code that

Implement both Linear Search and Binary Search. Which takes the following Instructions:

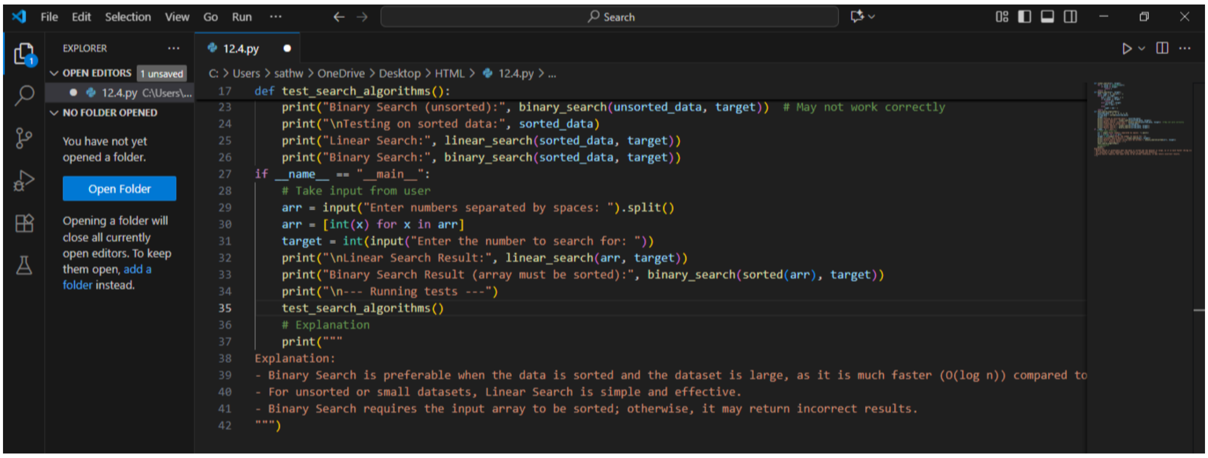
o Use AI to generate docstrings and performance notes.

o Test both algorithms on sorted and unsorted data.

o Ask AI to explain when Binary Search is preferable and generate a output which takes inputs from user.



```python
def linear_search(arr, target):
    for i, value in enumerate(arr):
        if value == target:
            return i
    return -1
def binary_search(arr, target):
    left, right = 0, len(arr) - 1
    while left <= right:
        mid = (left + right) // 2
        if arr[mid] == target:
            return mid
        elif arr[mid] < target:
            left = mid + 1
        else:
            right = mid - 1
    return -1
def test_search_algorithms():
    unsorted_data = [7, 2, 9, 4, 1, 5]
    sorted_data = sorted(unsorted_data)
    target = 4
    print("Testing on unsorted data:", unsorted_data)
    print("Linear Search:", linear_search(unsorted_data, target))
    print("Binary Search (unsorted):", binary_search(unsorted_data, target))  # May not work correctly
    print("\nTesting on sorted data:", sorted_data)
    print("Linear Search:", linear_search(sorted_data, target))
    print("Binary Search:", binary_search(sorted_data, target))
if __name__ == "__main__":
```



```python
def test_search_algorithms():
    print("Binary Search (unsorted):", binary_search(unsorted_data, target))  # May not work correctly
    print("\nTesting on sorted data:", sorted_data)
    print("Linear Search:", linear_search(sorted_data, target))
    print("Binary Search:", binary_search(sorted_data, target))
if __name__ == "__main__":
    # Take input from user
    arr = input("Enter numbers separated by spaces: ").split()
    arr = [int(x) for x in arr]
    target = int(input("Enter the number to search for: "))
    print("\nLinear Search Result:", linear_search(arr, target))
    print("Binary Search Result (array must be sorted):", binary_search(sorted(arr), target))
    print("\n--- Running tests ---")
    test_search_algorithms()
    # Explanation
    print("""
Explanation:
- Binary Search is preferable when the data is sorted and the dataset is large, as it is much faster (O(log n)) compared to
- For unsorted or small datasets, Linear Search is simple and effective.
- Binary Search requires the input array to be sorted; otherwise, it may return incorrect results.
""")
```

OUTPUT:

Enter numbers separated by spaces: 90 32 67 60 1

Enter the number to search for: 60

Linear Search Result: 3

Binary Search Result (array must be sorted): 2

--- Running tests ---

Testing on unsorted data: [7, 2, 9, 4, 1, 5]

Linear Search: 3

Binary Search (unsorted): -1

Testing on sorted data: [1, 2, 4, 5, 7, 9]

Linear Search: 2

Binary Search: 2

Explanation:- Binary Search is preferable when the data is sorted and the dataset is large, as it is much faster (O(log n)) compared to Linear Search (O(n)).

- For unsorted or small datasets, Linear Search is simple and effective.

- Binary Search requires the input array to be sorted; otherwise, it may return incorrect results.

EXPLANATION:

• def linear_search(arr, target):: This line defines a function named linear_search that takes two arguments: arr (the list to search within) and target (the value to search for). • """ ... """: This is a docstring, which explains what the function does, its arguments (Args), and what it returns (Returns). • for index, element in enumerate(arr):: This loop iterates through each element in the input list arr. The enumerate() function provides both the index and the value of each element.

TASK-04

PROMPT: **Quick Sort and Merge Sort Comparison** Write a code that 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 and take inputs from user.

```python
import random
import time
def quick_sort(arr):
    if len(arr) <= 1:
        return arr
    pivot = arr[len(arr) // 2]
    left = [x for x in arr if x < pivot]
    middle = [x for x in arr if x == pivot]
    right = [x for x in arr if x > pivot]
    return quick_sort(left) + middle + quick_sort(right)
def merge_sort(arr):
    if len(arr) <= 1:
        return arr
    mid = len(arr) // 2
    left = merge_sort(arr[:mid])
    right = merge_sort(arr[mid:])
    return merge(left, right)
def merge(left, right):
    result = []
    i = j = 0
    while i < len(left) and j < len(right):
        if left[i] < right[j]:
            result.append(left[i])
            i += 1
        else:
            result.append(right[j])
            j += 1
    result.extend(left[i:])
    result.extend(right[j:])
    return result
def compare_sorts(arr):
    arr1 = arr.copy()
    arr2 = arr.copy()
    start = time.time()
    sorted_quick = quick_sort(arr1)
    quick_time = time.time() - start
    start = time.time()
    sorted_merge = merge_sort(arr2)
    merge_time = time.time() - start
    print(f"Quick Sort Time: {quick_time:.6f} seconds")
    print(f"Merge Sort Time: {merge_time:.6f} seconds")
    print(f"Both sorted correctly: {sorted_quick == sorted_merge}")
def main():
    n = int(input("Enter the number of elements: "))
    print("Choose input type:")
    print("1. Random list")
    print("2. Sorted list")
    print("3. Reverse-sorted list")
    choice = int(input("Enter choice (1/2/3): "))
    if choice == 1:
        arr = [random.randint(1, 10000) for _ in range(n)]
    elif choice == 2:
        arr = list(range(1, n+1))
    elif choice == 3:
        arr = list(range(n, 0, -1))
    else:
        print("Invalid choice.")
        return
    print("Comparing Quick Sort and Merge Sort...")
    compare_sorts(arr)
if __name__ == "__main__":
    main()
```

OUTPUT:

Enter the number of elements: 3

Choose input type:

1. Random list

2. Sorted list

3. Reverse-sorted list

Enter choice (1/2/3): 1

Comparing Quick Sort and Merge Sort...

Quick Sort Time: 0.000028 seconds

Merge Sort Time: 0.000020 seconds

Both sorted correctly: True

EXPLANATION:

def quick_sort(arr):: This defines the recursive function quick_sort that takes a list arr as input. """ ... """: This is a docstring explaining the function's purpose, arguments, and return value. if len(arr) <= 1:: This is the base case for the recursion. If the list has 0 or 1 element, it's already sorted, so the function simply returns the list. pivot = arr[0]: This selects the first element of the list as the pivot. Note: Different pivot selection strategies exist, and the choice of pivot significantly impacts performance.

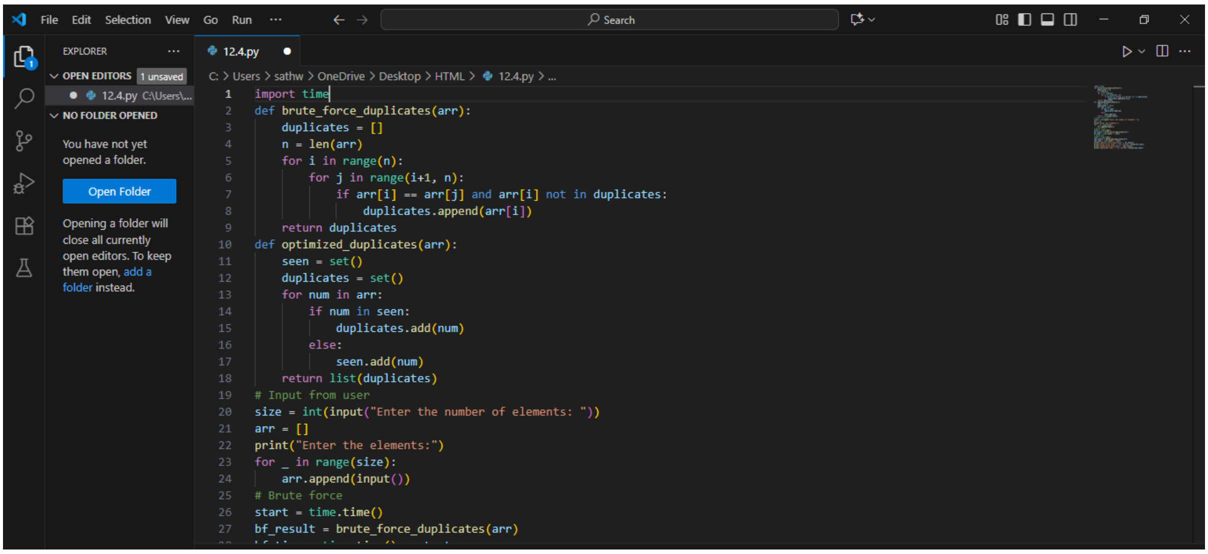
TASK-05

PROMPT: **AI-Suggested Algorithm Optimization** Write a code that Give AI a naive algorithm (e.g., $O(n^2)$ duplicate search) which allows the following Instructions:

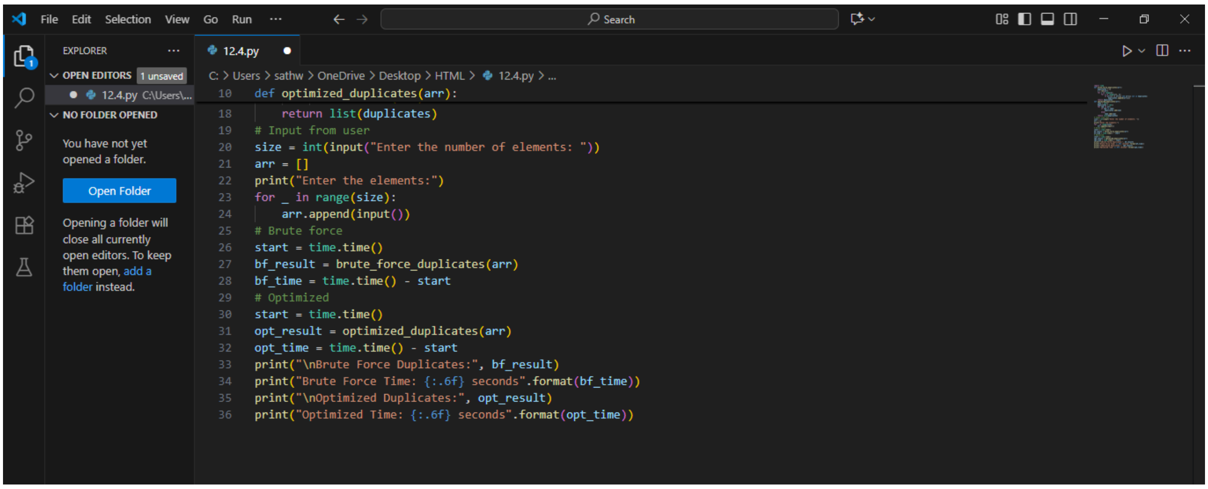
o Students write a brute force duplicate-finder.

o Ask AI to optimize it (e.g., by using sets/dictionaries with $O(n)$ time).

o Compare execution times with large input sizes which takes inputs from user.



```python
import time
def brute_force_duplicates(arr):
    duplicates = []
    n = len(arr)
    for i in range(n):
        for j in range(i+1, n):
            if arr[i] == arr[j] and arr[i] not in duplicates:
                duplicates.append(arr[i])
    return duplicates
def optimized_duplicates(arr):
    seen = set()
    duplicates = set()
    for num in arr:
        if num in seen:
            duplicates.add(num)
        else:
            seen.add(num)
    return list(duplicates)
# Input from user
size = int(input("Enter the number of elements: "))
arr = []
print("Enter the elements:")
for _ in range(size):
    arr.append(input())
# Brute force
start = time.time()
bf_result = brute_force_duplicates(arr)
```



```python
def optimized_duplicates(arr):
    return list(duplicates)
# Input from user
size = int(input("Enter the number of elements: "))
arr = []
print("Enter the elements:")
for _ in range(size):
    arr.append(input())
# Brute force
start = time.time()
bf_result = brute_force_duplicates(arr)
bf_time = time.time() - start
# Optimized
start = time.time()
opt_result = optimized_duplicates(arr)
opt_time = time.time() - start
print("\nBrute Force Duplicates:", bf_result)
print("Brute Force Time: {:.6f} seconds".format(bf_time))
print("\nOptimized Duplicates:", opt_result)
print("Optimized Time: {:.6f} seconds".format(opt_time))
```

OUTPUT:

Enter the number of elements: 3

Enter the elements:

12

54

33

Brute Force Duplicates: []

Brute Force Time: 0.000024 seconds

Optimized Duplicates: []

Optimized Time: 0.000020 seconds

EXPLANATION:

• performance_data = []: This initializes an empty list called performance_data. This list will be used to store dictionaries, where each dictionary will represent a row in the final performance table. • for size, data_types in execution_times.items():: This is the outer loop that iterates through the execution_times dictionary. execution_times likely contains the measured execution times for different list sizes. • for data_type, algorithms in data_types.items():: This inner loop iterates through the data types within each list size (e.g., 'random', 'sorted', 'reverse-sorted').