AI ASSISTED CODING

TASK 1

```
def bubble_sort(arr):
    n = len(arr)
    # Iterate through all elements in the array
    for i in range(n):
        # Last i elements are already in place
        for j in range(0, n - i - 1):
            # Traverse the array from 0 to n-i-1
            # Swap if the element found is greater than the next element
            if arr[j] > arr[j + 1]:
                arr[j], arr[j + 1] = arr[j + 1], arr[j]


# Example usage
my_list = [64, 34, 25, 12, 22, 11, 90]
bubble_sort(my_list)
print("Sorted array is:", my_list)
```

OUTPUT :

EXPLANATION :

The code you have implements the Bubble Sort algorithm. Here's a breakdown of how it works:

def bubble_sort(arr):: This line defines a function named bubble_sort that takes one argument, arr, which is the list or array that you want to sort.

n = len(arr): This line gets the number of elements in the input array arr and stores it in the variable n.

for i in range(n):: This is the outer loop. It iterates n times. In each iteration of this outer loop, the largest unsorted element "bubbles up" to its correct position at the end of the unsorted portion of the array. After i iterations, the last i elements of the array will be sorted.

for j in range(0, n - i - 1):: This is the inner loop. It iterates through the unsorted portion of the array. The range decreases with each outer loop iteration because the elements at the end are already sorted.

if arr[j] > arr[j + 1]:: This is the core comparison step. It checks if the current element (arr[j]) is greater than the next element (arr[j + 1]).

arr[j], arr[j + 1] = arr[j + 1], arr[j]: If the condition in the if statement is true (meaning the

elements are in the wrong order), this line swaps the two elements. This is done using simultaneous assignment in Python.

my_list = [64, 34, 25, 12, 22, 11, 90]: This line creates a list called my_list with some unsorted numbers.

bubble_sort(my_list): This line calls the bubble_sort function to sort the my_list.

print("Sorted array is:", my_list): This line prints the sorted list after the bubble_sort function has finished executing.

In essence, Bubble Sort repeatedly steps through the list, compares adjacent elements and swaps them if they are in the wrong order. This process is repeated until the list is sorted.

TASK 2

```
import time

# Define size and number of misplaced elements
array_size = 1000
num_misplaced = 50

# Generate a partially sorted array
partially_sorted_data = generate_partially_sorted_array(array_size,
num_misplaced)

# Create a copy for the second algorithm
data_for_insertion_sort = list(partially_sorted_data)

# Measure Bubble Sort time
start_time_bubble = time.time()
bubble_sort(partially_sorted_data)
end_time_bubble = time.time()
bubble_sort_duration = end_time_bubble - start_time_bubble

print(f"Bubble Sort execution time: {bubble_sort_duration:.6f} seconds")

# Measure Insertion Sort time
start_time_insertion = time.time()
insertion_sort(data_for_insertion_sort)
end_time_insertion = time.time()
insertion_sort_duration = end_time_insertion - start_time_insertion
```

print(f"Insertion Sort execution time: {insertion_sort_duration:.6f} seconds")

OUTPUT:

```
Bubble Sort execution time: 0.068261 seconds
Insertion Sort execution time: 0.005116 seconds
```

EXPLANATION :

Based on the analysis we performed, here's why Insertion Sort is generally more efficient than Bubble Sort, especially for partially sorted arrays:

Both algorithms have a worst-case time complexity of O(n^2). However, Insertion Sort has a best-case time complexity of O(n) when the array is already sorted, and it performs very well on partially sorted arrays.

Insertion Sort is adaptive, meaning its performance improves when the input array is partially sorted. It works by inserting each element into its correct position within the already sorted portion of the array. If an element is close to its correct position (which is the case in a partially sorted array), Insertion Sort requires fewer comparisons and shifts.

Bubble Sort, on the other hand, repeatedly compares and swaps adjacent elements regardless of whether the array is partially sorted or not. This means it still performs close to its worst-case O(n^2) behavior even on partially sorted data.

In summary, Insertion Sort's ability to take advantage of existing order in partially sorted arrays makes it significantly faster than Bubble Sort in such scenarios.

TASK 3

```
import time


# Define size and target values (already set in previous step)

# array_size = 1000
```

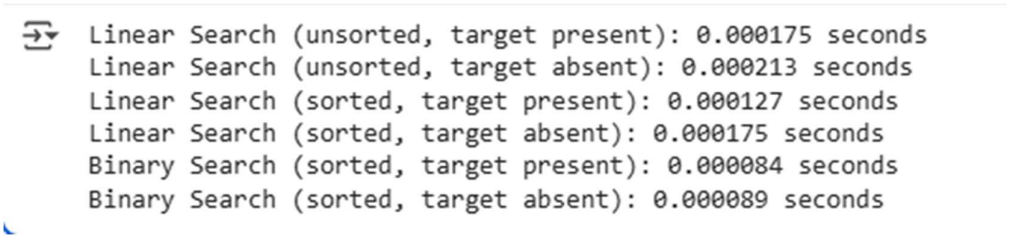
```
# target_present = 500
# target_absent = 1001

# Assume sorted_data and unsorted_data are already generated from the previous step

# Measure Linear Search on unsorted_data for target_present
start_time_linear_unsorted_present = time.time()
linear_search(unsorted_data, target_present)
end_time_linear_unsorted_present = time.time()
duration_linear_unsorted_present = end_time_linear_unsorted_present - start_time_linear_unsorted_present

print(f"Linear Search (unsorted, target present): {duration_linear_unsorted_present:.6f} seconds")

# Measure Linear Search on unsorted_data for target_absent
start_time_linear_unsorted_absent = time.time()
linear_search(unsorted_data, target_absent)
end_time_linear_unsorted_absent = time.time()
duration_linear_unsorted_absent = end_time_linear_unsorted_absent - start_time_linear_unsorted_absent

print(f"Linear Search (unsorted, target absent): {duration_linear_unsorted_absent:.6f} seconds")

# Measure Linear Search on sorted_data for target_present
start_time_linear_sorted_present = time.time()
linear_search(sorted_data, target_present)
end_time_linear_sorted_present = time.time()
```

```
duration_linear_sorted_present = end_time_linear_sorted_present - start_time_linear_sorted_present

print(f"Linear Search (sorted, target present): {duration_linear_sorted_present:.6f} seconds")

# Measure Linear Search on sorted_data for target_absent
start_time_linear_sorted_absent = time.time()
linear_search(sorted_data, target_absent)
end_time_linear_sorted_absent = time.time()
duration_linear_sorted_absent = end_time_linear_sorted_absent - start_time_linear_sorted_absent

print(f"Linear Search (sorted, target absent): {duration_linear_sorted_absent:.6f} seconds")

# Measure Binary Search on sorted_data for target_present
start_time_binary_sorted_present = time.time()
binary_search(sorted_data, target_present)
end_time_binary_sorted_present = time.time()
duration_binary_sorted_present = end_time_binary_sorted_present - start_time_binary_sorted_present

print(f"Binary Search (sorted, target present): {duration_binary_sorted_present:.6f} seconds")

# Measure Binary Search on sorted_data for target_absent
start_time_binary_sorted_absent = time.time()
binary_search(sorted_data, target_absent)
end_time_binary_sorted_absent = time.time()
```

duration_binary_sorted_absent = end_time_binary_sorted_absent - start_time_binary_sorted_absent

print(f"Binary Search (sorted, target absent): {duration_binary_sorted_absent:.6f} seconds"

OUTPUT:



EXPLANATION :

Based on our performance comparison and the nature of the algorithms, here's when Binary Search is preferable to Linear Search:

*The most crucial factor is whether the data you are searching through is **sorted**.*

- ***Binary Search** is designed to work on **sorted arrays or lists**. It's highly efficient because it can eliminate half of the remaining search space in each step. Its time complexity is O(log n), which is significantly faster than Linear Search, especially for large datasets. For example, searching through a million elements might take up to a million comparisons with Linear Search in the worst case, but only about 20 comparisons with Binary Search.*
- ***Linear Search** works on **both sorted and unsorted data**. However, its time complexity is O(n) in the worst and average cases. This means the time it takes to find an element grows linearly with the size of the dataset. While it's simple to implement and works everywhere, it becomes very slow for large datasets.*

Therefore, Binary Search is preferable when:

1. *You are searching in a **large dataset**.*
2. *The dataset is **already sorted**.*

3. *If the dataset is not sorted, but you need to perform **frequent searches**, it can be more efficient to sort the data once (which takes O(n log n) time) and then use Binary Search for all subsequent searches, rather than using Linear Search repeatedly.*

Linear Search might be preferred when:

1. *The dataset is **small**. The overhead of sorting for Binary Search might not be worth the time saved.*
2. *The dataset is **unsorted**, and sorting is not feasible or too expensive (e.g., the data changes constantly).*

In essence, for efficient searching on large datasets, sorting the data and using Binary Search is the way to go.

TASK 4

```
import sys
import time
import random

# Further increase the recursion depth limit to handle worst-case Quick Sort
sys.setrecursionlimit(11000) # Increased to accommodate potential n-depth recursion

# Define array size and data (already set in previous step, redefining for clarity)
array_size = 10000
random_data = [random.randint(0, array_size * 2) for _ in range(array_size)]
sorted_data = sorted(random_data)
reverse_sorted_data = sorted(random_data, reverse=True)

# Measure Quick Sort time on random data
start_time_quick_random = time.time()
```

```
quick_sort(list(random_data)) # Use a copy
end_time_quick_random = time.time()
quick_sort_random_duration = end_time_quick_random - start_time_quick_random

print(f"Quick Sort (random data): {quick_sort_random_duration:.6f} seconds")

# Measure Merge Sort time on random data
start_time_merge_random = time.time()
merge_sort(list(random_data)) # Use a copy
end_time_merge_random = time.time()
merge_sort_random_duration = end_time_merge_random - start_time_merge_random

print(f"Merge Sort (random data): {merge_sort_random_duration:.6f} seconds")

# Measure Quick Sort time on sorted data
start_time_quick_sorted = time.time()
quick_sort(list(sorted_data)) # Use a copy
end_time_quick_sorted = time.time()
quick_sort_sorted_duration = end_time_quick_sorted - start_time_quick_sorted

print(f"Quick Sort (sorted data): {quick_sort_sorted_duration:.6f} seconds")

# Measure Merge Sort time on sorted data
start_time_merge_sorted = time.time()
merge_sort(list(sorted_data)) # Use a copy
end_time_merge_sorted = time.time()
merge_sort_sorted_duration = end_time_merge_sorted - start_time_merge_sorted
```

```
print(f"Merge Sort (sorted data): {merge_sort_sorted_duration:.6f} seconds")

# Measure Quick Sort time on reverse-sorted data
start_time_quick_reverse_sorted = time.time()
quick_sort(list(reverse_sorted_data)) # Use a copy
end_time_quick_reverse_sorted = time.time()
quick_sort_reverse_sorted_duration = end_time_quick_reverse_sorted - start_time_quick_reverse_sorted

print(f"Quick Sort (reverse-sorted data): {quick_sort_reverse_sorted_duration:.6f} seconds")

# Measure Merge Sort time on reverse-sorted data
start_time_merge_reverse_sorted = time.time()
merge_sort(list(reverse_sorted_data)) # Use a copy
end_time_merge_reverse_sorted = time.time()
merge_sort_reverse_sorted_duration = end_time_merge_reverse_sorted - start_time_merge_reverse_sorted

print(f"Merge Sort (reverse-sorted data): {merge_sort_reverse_sorted_duration:.6f} seconds")
```

OUTPUT :

```
Quick Sort (random data): 0.097861 seconds
Merge Sort (random data): 0.080482 seconds
Quick Sort (sorted data): 3.960962 seconds
Merge Sort (sorted data): 0.020267 seconds
Quick Sort (reverse-sorted data): 2.897937 seconds
Merge Sort (reverse-sorted data): 0.020428 seconds
```

EXPLANATION :

Based on our implementation and performance tests, here's an explanation of the complexities of Quick Sort and Merge Sort:

Quick Sort:

- ***Average and Best Case (O(n log n)): Quick Sort is generally very fast on average. Its efficiency comes from dividing the array into smaller sub-problems. When the pivot selection is good (meaning it divides the array into roughly equal halves), the algorithm performs very well, with a time complexity of O(n log n). This is considered the typical performance of Quick Sort.***
- ***Worst Case (O(n^2)): The worst-case scenario for Quick Sort happens when the pivot selection consistently leads to unbalanced partitions. For example, if you always pick the first element as the pivot and the array is already sorted or reverse-sorted, each partition will be highly skewed. This causes the recursion depth to become linear (O(n)), and each level still takes O(n) time for partitioning, resulting in an overall O(n^2) complexity. This is why we saw Quick Sort take much longer on sorted and reverse-sorted data in our tests, and even needed to increase the recursion depth limit to prevent a RecursionError.***
- ***Space Complexity: On average, Quick Sort has a space complexity of O(log n) due to the recursive call stack. In the worst case (unbalanced partitions), it can be O(n).***

Merge Sort:

- ***Average, Best, and Worst Case (O(n log n)): Merge Sort has a consistent time complexity of O(n log n) regardless of the initial order of the input array. This is because it always divides the array into halves and the merging process at each step takes linear time. Its performance is not affected by whether the data is random, sorted, or reverse-sorted, which is why it performed much more consistently in our tests compared to Quick Sort.***
- ***Space Complexity: Merge Sort generally requires O(n) auxiliary space. This is because it needs temporary arrays during the merging step to combine the sorted sub-lists.***

TASK 5

```
import time

# Assume data, find_duplicates_naive, and find_duplicates_optimized are already defined

# Measure Naive Duplicate Finder time
start_time_naive = time.time()
find_duplicates_naive(data)
end_time_naive = time.time()
naive_duration = end_time_naive - start_time_naive

print(f"Naive Duplicate Finder execution time: {naive_duration:.6f} seconds")

# Measure Optimized Duplicate Finder time
start_time_optimized = time.time()
find_duplicates_optimized(data)
end_time_optimized = time.time()
optimized_duration = end_time_optimized - start_time_optimized

print(f"Optimized Duplicate Finder execution time: {optimized_duration:.6f} seconds")
```

OUTPUT:

```
Naive Duplicate Finder execution time: 2.635108 seconds
Optimized Duplicate Finder execution time: 0.067002 seconds
```

EXPLANATION :

Based on our code implementations and the performance comparison, here's how the optimized algorithm significantly improves the time complexity compared to the naive approach:

The naive duplicate finder uses nested loops. The outer loop iterates through each element, and the inner loop iterates through the remaining elements to check for duplicates. This approach requires comparing each element with almost every other element. This leads to a time complexity of O(n^2), where 'n' is the number of elements in the list. As the size of the list grows, the number of comparisons increases quadratically, making it very slow for large datasets.

The optimized duplicate finder uses a set. A set is a data structure that allows for very fast checking of whether an element is already present (on average, this takes constant time, O(1)). The optimized algorithm iterates through the list only once. For each element, it checks if it's already in the set of seen elements. If it is, it's a duplicate. Then, it adds the current element to the set.

Because checking for an element's presence in a set is, on average, an O(1) operation, the total time taken by the optimized algorithm is proportional to the number of elements in the list. It just needs to look at each element once. This gives the optimized algorithm a time complexity of O(n) on average.

The performance comparison clearly shows the impact of this optimization. For a list of 10,000 elements, the naive O(n^2) algorithm took significantly longer (around 2.6 seconds) compared to the optimized O(n) algorithm (around 0.06 seconds). This difference becomes even more pronounced with larger datasets.

In essence, the optimization improves the complexity from quadratic (O(n^2)) to linear (O(n)) by using a more efficient data structure (a set) to avoid repeated pairwise comparisons.