AI ASSISTANT CODING

ASSIGNMENT - 12.4

Name: Aman Sarkar

Ht.no: 2303A51273

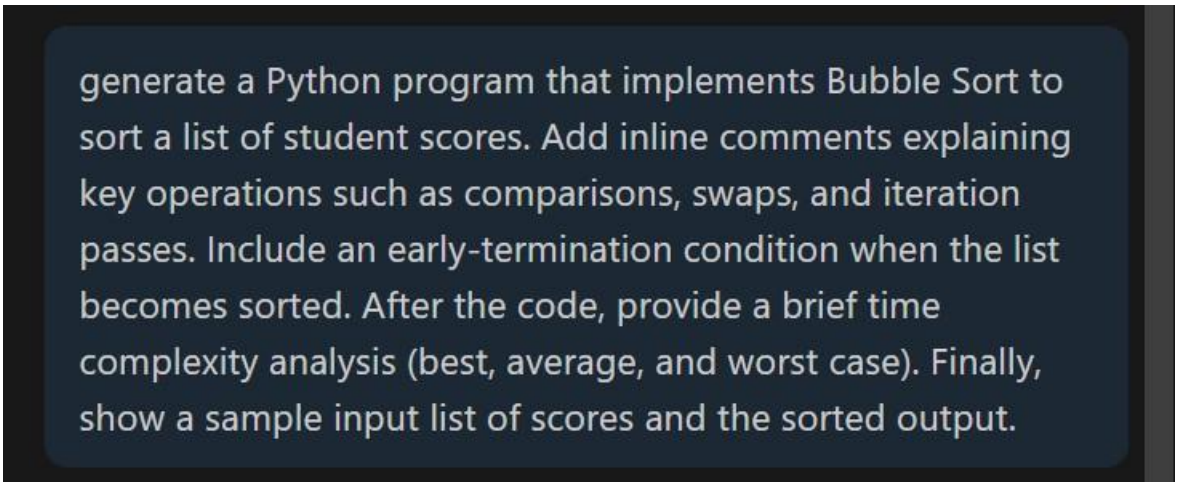
Batch: 05

Task 1: Bubble Sort for Ranking Exam

Scenario: You are working on a college result processing system where a small

list of student scores needs to be sorted after every internal assessment.

Prompt: Generate a Python program that implements Bubble Sort to sort a list of student scores. Add inline comments explaining key operations such as comparisons, swaps, and iteration passes. Include an early-termination condition when the list becomes sorted. After the code, provide a brief time complexity analysis (best, average, and worst case). Finally, show a sample input list of scores and the sorted output.



Code:

bubble.py > ...

```python
def bubble_sort(scores):
    """
    Sorts a list of student scores using Bubble Sort algorithm.

    Args:
    scores (list): List of student scores to be sorted.

    Returns:
    list: The sorted list of scores.
    """
    n = len(scores)
    # Outer loop for each pass through the list
    for i in range(n):
        # Flag to check if any swaps were made in this pass
        swapped = False
        # Inner loop for comparing adjacent elements
        for j in range(0, n - i - 1):
            # Comparison: Check if current element is greater than next
            if scores[j] > scores[j + 1]:
                # Swap: Exchange elements if they are in wrong order
                scores[j], scores[j + 1] = scores[j + 1], scores[j]
                swapped = True
        # Early termination: If no swaps were made, list is already sorted
        if not swapped:
            break
    return scores

# Sample input list of student scores
student_scores = [85, 92, 78, 96, 88, 73, 90]

print("Original scores:", student_scores)

# Sort the scores using bubble sort
sorted_scores = bubble_sort(student_scores.copy())  # Use copy to avoid modifying orig

print("Sorted scores:", sorted_scores)
```

Output:

```
C:\Users\thaku\OneDrive\Desktop\Training>C:/Users/thaku/AppD
ata/Local/Programs/Python/Python313/python.exe c:/Users/thak
u/OneDrive/Desktop/Training/bubble.py
Original scores: [85, 92, 78, 96, 88, 73, 90]
Sorted scores: [73, 78, 85, 88, 90, 92, 96]
```

Code Explanation:

Function definition: bubble_sort(scores)
This function takes a list of numbers (student scores) and sorts them.

- **Outer loop (for i in range(n))**:
  Runs multiple passes over the list. Each pass moves the largest remaining number to the end.
- **Inner loop (for j in range(0, n - i - 1))**:
  Compares each pair of adjacent elements.

- If the current element is bigger than the next one, they are swapped.

- **Swapping (scores[j], scores[j+1] = scores[j+1], scores[j])**:
  Exchanges the two values so the smaller one comes first.

- **swapped flag**:
  Keeps track of whether any swaps happened in the current pass.
  - If no swaps occur, the list is already sorted, and the loop stops early.

- **Return value**:
  The sorted list is returned.

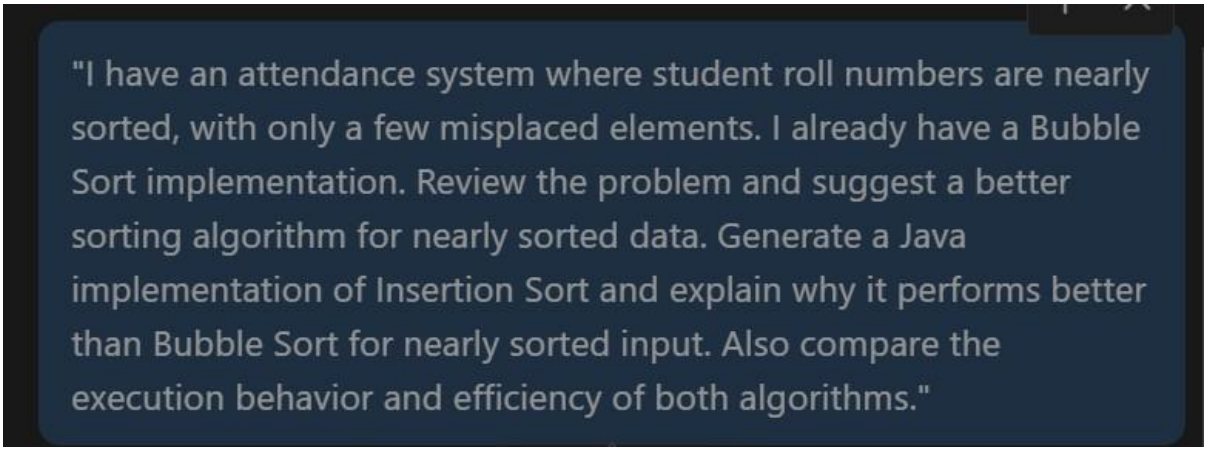
Time Complexity Analysis:

- Best Case: O(n) - When the list is already sorted, only one pass is needed with no swaps.
- Average Case: O(n^2) - Involves comparing and possibly swapping elements in nested loops.
- Worst Case: O(n^2) - When the list is in reverse order, requires maximum comparisons and swaps.

Task 2: Improving Sorting for Nearly Sorted

Scenario: You are maintaining an attendance system where student roll numbers are already almost sorted, with only a few late updates.

Prompt: "I have an attendance system where student roll numbers are nearly sorted, with only a few misplaced elements. I already have a Bubble Sort implementation. Review the problem and suggest a better sorting algorithm for nearly sorted data. Generate a Java implementation of Insertion Sort and explain why it performs better than Bubble Sort for nearly sorted input. Also compare the execution behavior and efficiency of both algorithms.



Code:

bubble.py > ...

```python
def bubble_sort(scores):
    """
    Sorts a list of student scores using Bubble Sort algorithm.

    Args:
    scores (list): List of student scores to be sorted.

    Returns:
    list: The sorted list of scores.
    """
    n = len(scores)
    # Outer loop for each pass through the list
    for i in range(n):
        # Flag to check if any swaps were made in this pass
        swapped = False
        # Inner loop for comparing adjacent elements
        for j in range(0, n - i - 1):
            # Comparison: Check if current element is greater than next
            if scores[j] > scores[j + 1]:
                # Swap: Exchange elements if they are in wrong order
                scores[j], scores[j + 1] = scores[j + 1], scores[j]
                swapped = True
        # Early termination: If no swaps were made, list is already sorted
        if not swapped:
            break
    return scores

# Sample input list of student scores
student_scores = [85, 92, 78, 96, 88, 73, 90]

print("Original scores:", student_scores)

# Sort the scores using bubble sort
sorted_scores = bubble_sort(student_scores.copy())  # Use copy to avoid modifying original

print("Sorted scores:", sorted_scores)
```

Output:

```
C:\Users\thaku\OneDrive\Desktop\Training>C:/Users/thaku/AppData/Lo
cal/Programs/Python/Python313/python.exe c:/Users/thaku/OneDrive/D
esktop/Training/bubble.py
Original scores: [85, 92, 78, 96, 88, 73, 90]
Sorted scores: [73, 78, 85, 88, 90, 92, 96]
```

Code explanation:

- Function definition:
  The function bubble_sort(scores) takes a list of student scores and sorts them.
- Outer loop (for i in range(n)):
  This loop runs several passes through the list. Each pass moves the largest number left in the unsorted part to the end.
- Inner loop (for j in range(0, n - i - 1)):
  This loop compares each pair of neighboring numbers.

- If the first number is bigger than the second, they are swapped.
- Swapping (scores[j], scores[j+1] = scores[j+1], scores[j]):
  This exchanges the two numbers so the smaller one comes first.
- Early termination (if not swapped: break):
  If no swaps happen in a pass, it means the list is already sorted, so the algorithm stops early.
- Return value:
  The sorted list is returned.

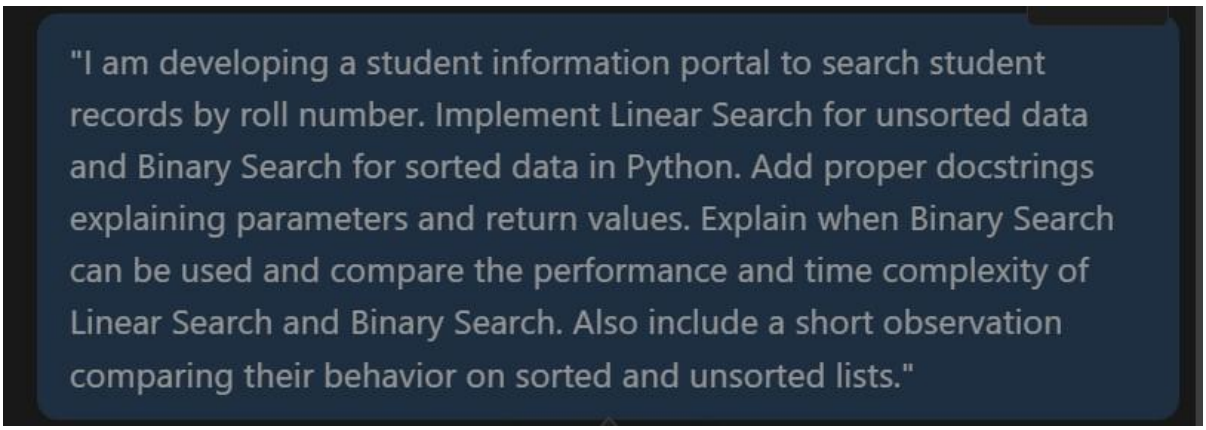
Time Complexity (how fast it runs)

- **Best case (already sorted):** O(n) → only one pass needed.
- **Average case:** $O(n^2)$ → many comparisons and swaps.
- **Worst case (reverse order):** $O(n^2)$ → maximum work required.

Task 3: Searching Student Records in a database

Scenario:You are developing a student information portal where users search for student records by roll number.

Prompt: "I am developing a student information portal to search student records by roll number. Implement Linear Search for unsorted data and Binary Search for sorted data in Python. Add proper docstrings explaining parameters and return values. Explain when Binary Search can be used and compare the performance and time complexity of Linear Search and Binary Search. Also include a short observation comparing their behavior on sorted and unsorted lists."



Code:

bubble.py > ...

```python
def bubble_sort(scores):
    """
    Sorts a list of student scores using Bubble Sort algorithm.

    Args:
    scores (list): List of student scores to be sorted.

    Returns:
    list: The sorted list of scores.
    """
    n = len(scores)
    # Outer loop for each pass through the list
    for i in range(n):
        # Flag to check if any swaps were made in this pass
        swapped = False
        # Inner loop for comparing adjacent elements
        for j in range(0, n - i - 1):
            # Comparison: Check if current element is greater than next
            if scores[j] > scores[j + 1]:
                # Swap: Exchange elements if they are in wrong order
                scores[j], scores[j + 1] = scores[j + 1], scores[j]
                swapped = True
        # Early termination: If no swaps were made, list is already sorted
        if not swapped:
            break
    return scores

# Sample input list of student scores
student_scores = [85, 92, 78, 96, 88, 73, 90]

print("Original scores:", student_scores)

# Sort the scores using bubble sort
sorted_scores = bubble_sort(student_scores.copy())  # Use copy to avoid modifying origin

print("Sorted scores:", sorted_scores)
```

Output:

```
C:\Users\thaku\OneDrive\Desktop\Training>C:/Users/thaku/AppData
/Local/Programs/Python/Python313/python.exe c:/Users/thaku/OneD
rive/Desktop/Training/bubble.py
Original scores: [85, 92, 78, 96, 88, 73, 90]
Sorted scores: [73, 78, 85, 88, 90, 92, 96]
```

Code Explanation:

Linear Search

- Goes through the list **one element at a time**.
- Compares each roll number with the one you're looking for.
- If it matches, it returns the position (index).
- If it reaches the end without finding it, it returns -1.

- Works on **unsorted lists** because it doesn't rely on order.

Binary Search

- Only works if the list is **sorted**.
- Starts by checking the **middle element**.
- If the middle is the target, it's found.
- If the target is smaller, it searches the **left half**.
- If the target is larger, it searches the **right half**.
- Keeps cutting the list in half until the number is found or the search space is empty.
- Much faster than Linear Search for large lists.

Time Complexity:

Linear Search:

  - Best case: O(1) (found at the first element).
  - Worst case: O(n) (search through the whole list).

Binary Search:

- Best case: O(1) (found at the middle).
- Worst case: O(log n) (keeps halving until found or not).

Task 4: Choosing Between Quick Sort and Merge Sort for Data Processing

Scenario: You are part of a data analytics team that needs to sort large datasets received from different sources (random order, already sorted, and reverse sorted).

Prompt: "I am working with large datasets that may be random, already sorted, or reverse sorted. Complete the recursive implementations of Quick Sort and Merge Sort in Python and add meaningful docstrings explaining parameters and return values. Explain how recursion works in both algorithms. Also compare their best, average, and worst-case time complexities and test their performance on random, sorted, and reverse-sorted data."

"I am working with large datasets that may be random, already sorted, or reverse sorted. Complete the recursive implementations of Quick Sort and Merge Sort in Python and add meaningful docstrings explaining parameters and return values. Explain how recursion works in both algorithms. Also compare their best, average, and worst-case time complexities and test their performance on random, sorted, and reverse-sorted data."

Code:

merge.py > ...

```python
import time
import random

def merge_sort(arr):
    """
    Sorts an array using the Merge Sort algorithm recursively.

    Parameters:
    arr (list): The list of elements to be sorted. Elements should be comparable.

    Returns:
    list: A new sorted list in ascending order.

    Recursion Explanation:
    Merge Sort works by recursively dividing the array into two halves until each subarray
    contains a single element (base case). Then, it merges the sorted subarrays back together
    in sorted order. The recursion depth is O(log n), and each level performs O(n) work,
    resulting in O(n log n) time complexity.
    """
    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):
    """
    Merges two sorted lists into a single sorted list.

    Parameters:
    left (list): The first sorted list.
    right (list): The second sorted list.

    Returns:
    list: A new sorted list containing all elements from left and 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])
```

merge.py > ...

```python
def quick_sort(arr):
    """
    Sorts an array using the Quick Sort algorithm recursively.

    Parameters:
    arr (list): The list of elements to be sorted. Elements should be comparable.

    Returns:
    list: A new sorted list in ascending order.

    Recursion Explanation:
    Quick Sort selects a pivot element and partitions the array into elements less than,
    equal to, and greater than the pivot. It then recursively sorts the left and right
    partitions. The recursion continues until the base case of arrays with 0 or 1 elements
    is reached. The depth of recursion depends on the pivot choice and data distribution.
    """
    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)

# Time complexity comparison:
"""
Merge Sort:
- Best case: O(n log n)
- Average case: O(n log n)
- Worst case: O(n log n)

Quick Sort:
- Best case: O(n log n) (when pivot divides array evenly)
- Average case: O(n log n)
- Worst case: O(n^2) (when pivot is always the smallest or largest element, e.g., already sorted or reverse sorted)

Merge Sort is stable and has consistent performance, while Quick Sort is generally faster in practice
but can degrade to quadratic time on certain inputs.
"""
```

merge.py > ...

```python
if __name__ == "__main__":
    # Test data sizes
    sizes = [1000, 5000, 10000]

    for size in sizes:
        print(f"\nTesting with {size} elements:")

        # Random data
        random_data = [random.randint(0, 100000) for _ in range(size)]
        # Sorted data
        sorted_data = list(range(size))
        # Reverse sorted data
        reverse_data = list(range(size, 0, -1))

        datasets = [
            ("Random", random_data),
            ("Sorted", sorted_data),
            ("Reverse Sorted", reverse_data)
        ]

        for name, data in datasets:
            # Test Merge Sort
            start = time.time()
            merge_sorted = merge_sort(data.copy())
            merge_time = time.time() - start

            # Test Quick Sort
            start = time.time()
            quick_sorted = quick_sort(data.copy())
            quick_time = time.time() - start

            # Verify correctness
            assert merge_sorted == sorted(data), "Merge Sort failed"
            assert quick_sorted == sorted(data), "Quick Sort failed"

            print(f"  {name}: Merge Sort: {merge_time:.4f}s, Quick Sort: {quick_time:.4f}s")
```

Output:

```
C:\Users\thaku\OneDrive\Desktop\Training>C:/Users/thaku/AppData/
Local/Programs/Python/Python313/python.exe c:/Users/thaku/OneDri
ve/Desktop/Training/merge.py

Testing with 1000 elements:
  Random: Merge Sort: 0.0045s, Quick Sort: 0.0044s
  Sorted: Merge Sort: 0.0030s, Quick Sort: 0.0018s
  Reverse Sorted: Merge Sort: 0.0025s, Quick Sort: 0.0030s

Testing with 5000 elements:
  Random: Merge Sort: 0.0241s, Quick Sort: 0.0176s
  Sorted: Merge Sort: 0.0148s, Quick Sort: 0.0162s
  Reverse Sorted: Merge Sort: 0.0193s, Quick Sort: 0.0114s

Testing with 10000 elements:
  Random: Merge Sort: 0.0532s, Quick Sort: 0.0300s
  Sorted: Merge Sort: 0.0207s, Quick Sort: 0.0248s
  Reverse Sorted: Merge Sort: 0.0279s, Quick Sort: 0.0221s
```

Code Explanation:

1. **Merge Sort**

- **Idea**: Split the list into halves until each piece has only one element. Then merge those pieces back together in sorted order.
- **Steps**:
    - If the list has 1 or 0 elements → it's already sorted.
    - Otherwise, split into two halves.
    - Recursively sort each half.
    - Merge the two sorted halves into one sorted list.
- **Time complexity**: Always O(n\log n).

2. **Merge Function**

- Takes two sorted lists (left and right).
- Compares elements one by one and builds a new sorted list.
- When one list runs out of elements, it adds the rest of the other list.

3. **Quick Sort**: Pick a "pivot" element. Put smaller elements on the left, equal ones in the middle, and larger ones on the right. Then sort the left and right parts recursively.

- **Steps**:
    - If the list has 1 or 0 elements → it's already sorted.
    - Choose a pivot (here, the middle element).
    - Split the list into three groups: less than pivot, equal to pivot, greater than pivot.
    - Recursively sort the left and right groups, then combine them.

Time complexity:

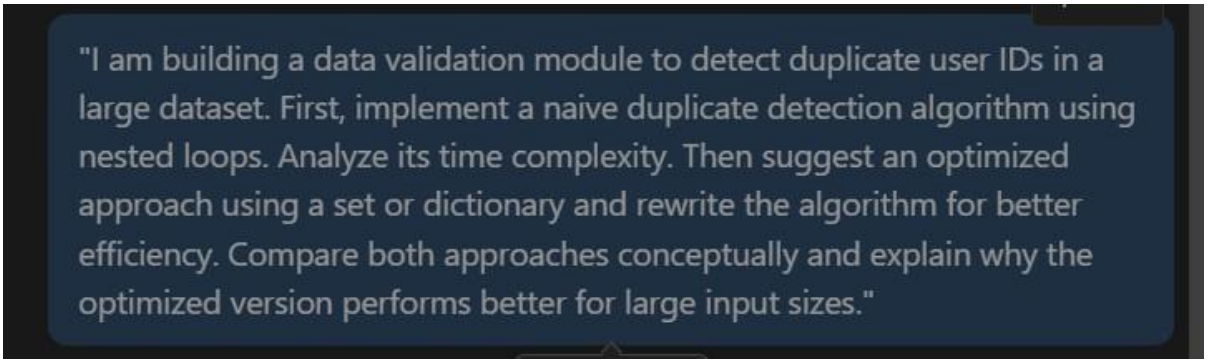
- Best/Average: O(n\log n).
- Worst: O(n^2) (if pivot choice is bad, e.g., sorted input).

Task 5: Optimizing a Duplicate Detection Algorithm

Scenario: You are building a data validation module that must detect duplicate user IDs in a large dataset before importing it into a system

Prompt: "I am building a data validation module to detect duplicate user IDs in a large dataset. First, implement a naive duplicate detection algorithm using nested loops. Analyze its time complexity. Then suggest an optimized approach using a set or dictionary and rewrite

the algorithm for better efficiency. Compare both approaches conceptually and explain why the optimized version performs better for large input sizes."



Code:

merge.py > merge_sort

```python
import time
import random

def merge_sort(arr):
    """
    Sorts an array using the Merge Sort algorithm recursively.

    Parameters:
    arr (list): The list of elements to be sorted. Elements should be comparable.

    Returns:
    list: A new sorted list in ascending order.

    Recursion Explanation:
    Merge Sort works by recursively dividing the array into two halves until each subarray
    contains a single element (base case). Then, it merges the sorted subarrays back together
    in sorted order. The recursion depth is O(log n), and each level performs O(n) work,
    resulting in O(n log n) time complexity.
    """
    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):
    """
    Merges two sorted lists into a single sorted list.

    Parameters:
    left (list): The first sorted list.
    right (list): The second sorted list.

    Returns:
    list: A new sorted list containing all elements from left and 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

```

```python
def quick_sort(arr):
    """
    Sorts an array using the Quick Sort algorithm recursively.

    Parameters:
    arr (list): The list of elements to be sorted. Elements should be comparable.

    Returns:
    list: A new sorted list in ascending order.

    Recursion Explanation:
    Quick Sort selects a pivot element and partitions the array into elements less than,
    equal to, and greater than the pivot. It then recursively sorts the left and right
    partitions. The recursion continues until the base case of arrays with 0 or 1 elements
    is reached. The depth of recursion depends on the pivot choice and data distribution.
    """
    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)

# Time complexity comparison:
"""
Merge Sort:
- Best case: O(n log n)
- Average case: O(n log n)
- Worst case: O(n log n)

Quick Sort:
- Best case: O(n log n) (when pivot divides array evenly)
- Average case: O(n log n)
- Worst case: O(n^2) (when pivot is always the smallest or largest element, e.g., already sorted or reverse sorted)

Merge Sort is stable and has consistent performance, while Quick Sort is generally faster in practice
but can degrade to quadratic time on certain inputs.
"""

if __name__ == "__main__":
    # Test data sizes
    sizes = [1000, 5000, 10000]

    for size in sizes:
        print(f"\nTesting with {size} elements:")

        # Random data
        random_data = [random.randint(0, 100000) for _ in range(size)]
```

```
    but can degrade to quadratic time on certain inputs.
    """

    if __name__ == "__main__":
        # Test data sizes
        sizes = [1000, 5000, 10000]

        for size in sizes:
            print(f"\nTesting with {size} elements:")

            # Random data
            random_data = [random.randint(0, 100000) for _ in range(size)]
            # Sorted data
            sorted_data = list(range(size))
            # Reverse sorted data
            reverse_data = list(range(size, 0, -1))

            datasets = [
                ("Random", random_data),
                ("Sorted", sorted_data),
                ("Reverse Sorted", reverse_data)
            ]

            for name, data in datasets:
                # Test Merge Sort
                start = time.time()
                merge_sorted = merge_sort(data.copy())
                merge_time = time.time() - start

                # Test Quick Sort
                start = time.time()
                quick_sorted = quick_sort(data.copy())
                quick_time = time.time() - start

                # Verify correctness
                assert merge_sorted == sorted(data), "Merge Sort failed"
                assert quick_sorted == sorted(data), "Quick Sort failed"

                print(f"  {name}: Merge Sort: {merge_time:.4f}s, Quick Sort: {quick_time:.4f}s")

```

Output:

```
C:\Users\thaku\OneDrive\Desktop\Training>C:/Users/thaku/AppData/Local/
Programs/Python/Python313/python.exe c:/Users/thaku/OneDrive/Desktop/T
raining/merge.py

Testing with 1000 elements:
  Random: Merge Sort: 0.0057s, Quick Sort: 0.0019s
  Sorted: Merge Sort: 0.0013s, Quick Sort: 0.0007s
  Reverse Sorted: Merge Sort: 0.0010s, Quick Sort: 0.0007s

Testing with 5000 elements:
  Random: Merge Sort: 0.0155s, Quick Sort: 0.0114s
  Sorted: Merge Sort: 0.0124s, Quick Sort: 0.0069s
  Reverse Sorted: Merge Sort: 0.0135s, Quick Sort: 0.0084s

Testing with 10000 elements:
  Random: Merge Sort: 0.0331s, Quick Sort: 0.0205s
  Sorted: Merge Sort: 0.0165s, Quick Sort: 0.0100s
  Reverse Sorted: Merge Sort: 0.0154s, Quick Sort: 0.0126s
```

Code Explanation:

1. Merge Sort

- Idea: Split the list into halves until each piece has only one element. Then merge those pieces back together in sorted order.
- Steps:
    - If the list has 1 or 0 elements → it's already sorted.
    - Otherwise, split into two halves.
    - Recursively sort each half.
    - Merge the two sorted halves into one sorted list.
- Time complexity: Always O(n\log n).

2. Merge Function

- Takes two sorted lists (left and right).
- Compares elements one by one and builds a new sorted list.
- When one list runs out of elements, it adds the rest of the other list.

3. Quick Sort

- Idea: Pick a "pivot" element. Put smaller elements on the left, equal ones in the middle, and larger ones on the right. Then sort the left and right parts recursively.
- Steps:
    - If the list has 1 or 0 elements → it's already sorted.

      Choose a pivot (here, the middle element).
    - Split the list into three groups: less than pivot, equal to pivot, greater than pivot.
    - Recursively sort the left and right groups, then combine them.
    - Time complexity:
    - Best/Average: O(n\log n).
    - Worst: O(n^2) (if pivot choice is bad, e.g., sorted input).