SORTING ALGORITHMS

Julio Rodriguez-Zambrano

CSCI 3412 - Algorithms

Part 1 - Selecting Sorting Algorithms

The three algorithms I chose were bubble sort for the naive sort, merge sort for the expected running time of O(n log n) and counting sort for an expected running time of O(n). I choose these algorithms because professor Williams recomended these algorithms.

Part 2 - Pseudo-Code

Bubble sort The bubble sort algorithm is a simple comparison-based sorting algorithm that repeatedly steps through the list, compares adjacent elements and swaps them if they are in the wrong order. The pass through the list is repeated until the list is sorted. The outer loop (for i) iterates from the beginning to the end of the array. The inner loop (for j) iterates from the beginning to the end of the unsorted portion of the array. At each iteration, if the element at index j is greater than the element at index j+1, they are swapped to put them in the correct order. After completing each pass through the array, the largest unsorted element "bubbles up" to its correct position at the end of the array.

```
bubble_sort(arr):
    n = length of arr
    for i = 0 to n - 1:
        for j = 0 to n - i -1:
            if arr[j] > arr[j + 1]:
                swap arr[j] and arr[j + 1]
```

Merge sort Merge Sort is a divide-and-conquer algorithm that divides the input array into two halves, recursively sorts each half, and then merges the sorted halves to produce the final sorted array. merge(left_half, right_half, arr): The helper function that merges two sorted subarrays left_half and right_half into a single sorted array arr. The merge_sort function recursively divides the array into two halves until each subarray has only one element. The merge function merges the two sorted subarrays by comparing elements from each subarray and placing them in the correct order in the original array arr.

```
mege_sort(arr):
   if length of arr > 1:
       mid = length of arr / 2
       left_half = arr[:mid]
       right_half = arr[mid:]

       merge_sort(left_half)
       merge_sort(right_half)

       merge(left_half, right_half, arr)

 merge(left_half, right_half, arr):
     i = j = k = 0

     while i < length of left_half and j < length of right_half:
         if left_half[i] <= right_half[j]:
             arr[k] = left_half[i]
             i = i + 1
         else:
             arr[k] = right_half[j]
             j = j + 1
         k = k + 1

     while i < length of left_half:
         arr[k] = left_half[i]
         i = i + 1
         k = k + 1

     while j < length of right_half:
         arr[k] = right_half[j]
         j = j + 1
         k = k + 1
```

Counting sort Counting Sort is a non-comparison-based sorting algorithm that works by counting the number of occurrences of each distinct element in the input array and using this information to place the elements in sorted order. counts: An array used to count the occurrences of each distinct element in the input array. The first loop counts the occurrences of each element in the input array and stores the counts in the counts array. The second loop iterates over the counts array and reconstructs the sorted array by placing each element in its correct position based on the counts stored in the counts array.

```
counting_sort(arr):
    counts = array of size k initialized to 0

    for each element x in arr:
        counts[x] = counts[i] + 1

    index = 0
    for i = 1 to l:
        for j = 0 to counts[i] - 1:
            arr[index] = i
            index = index = 1
```

Part 3 - Static Analysis

Bubble sort loop invariant Loop Invariant: At the start of each iteration of the outer loop of Bubble Sort, the subarray arr[0...i-1] contains the smallest i elements of the input array arr, in sorted order.

Initialization: Before the first iteration of the outer loop (i = 0), the subarray arr[0...i-1] is empty, and thus trivially sorted.

Maintenance: At the start of each iteration of the outer loop (i), the inner loop compares adjacent elements and swaps them if they are in the wrong order. This ensures that after the iteration, the largest element in the unsorted portion of the array is moved to its correct position at index i.

Termination: When the outer loop terminates (i = n), the entire array is sorted since the loop invariant holds for all i.

Merge sort loop invariant Loop Invariant: At the start of each iteration of the merge operation in Merge Sort, the subarray being merged (left_half and right_half) is sorted.

Initialization: When the merge operation is performed for subarrays of size 1, each subarray is trivially sorted.

Maintenance: During each merge operation, elements from left_half and right_half are compared and merged into a sorted subarray. The merge operation ensures that the resulting subarray is sorted.

Termination: When all merge operations are completed, the entire array is sorted since each merge operation maintains the sorted property of subarrays.

Counting sort loop invariant Loop Invariant: During the counting phase of Counting Sort, the counts array contains the frequency count of each element in the input array arr.

Initialization: Before counting, the counts array is initialized to all zeros.

Maintenance: During counting, for each element x in the input array arr, the corresponding count in the counts array is incremented.

Termination: After counting all elements in the input array, the counts array contains the frequency count of each element in the input array.

Part 4 - Implementation

In [2]:

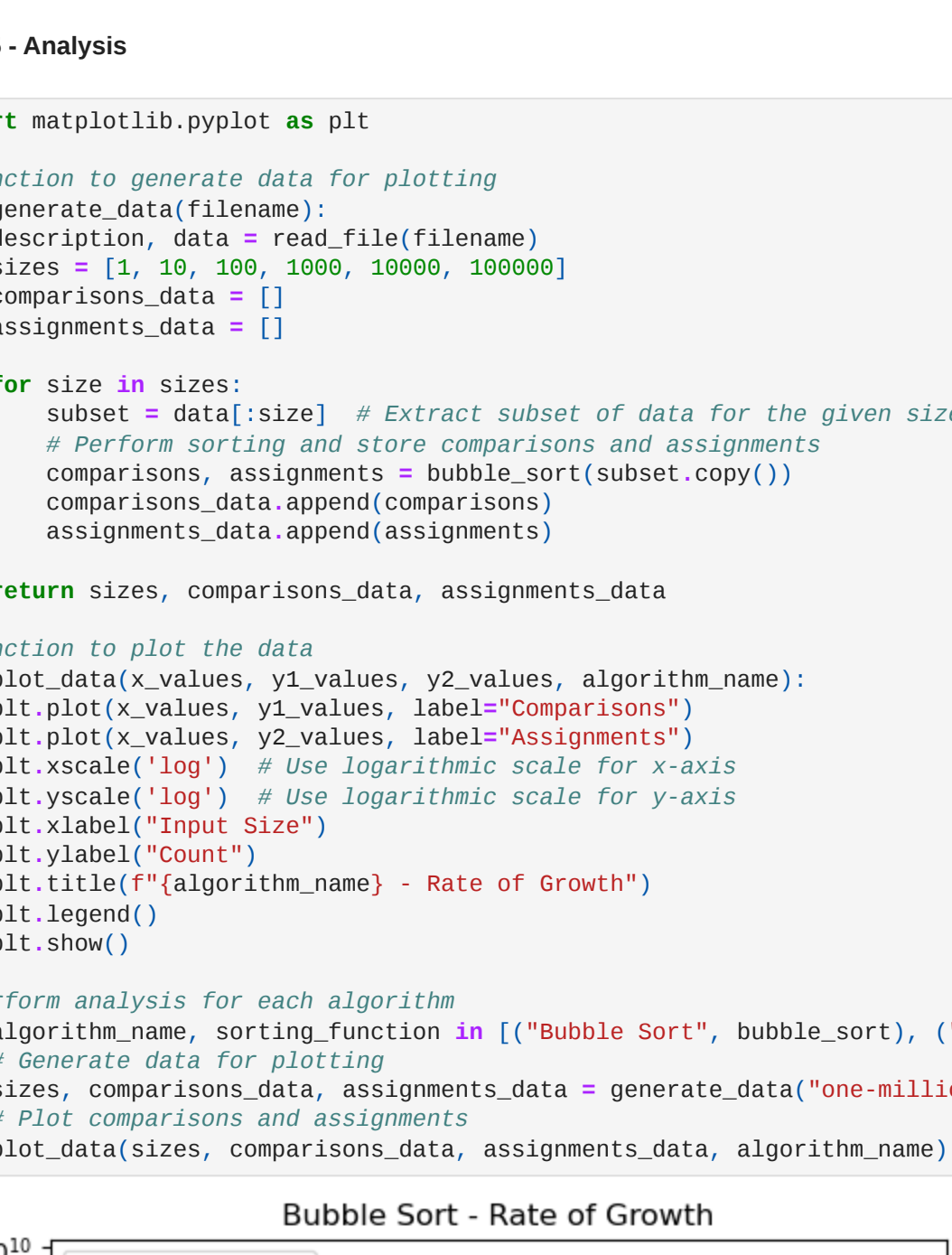
```python
# Function to read data from a file
def read_file(filename):
    # Open the file in read mode
    with open(filename, 'r') as file:
        # Read all lines from the file
        lines = file.readlines()
        # Extract description from the first line and remove leading/trailing whitespaces
        description = lines[0].strip()
        # Extract number of elements from the second line and convert to integer
        n = int(lines[1])
        # Extract data from subsequent lines, convert each element to integer, and store in a list
        data = [int(line.strip()) for line in lines[2:]]
        # Return the description and the data list
        return description, data

# Function to perform Bubble Sort on an array
def bubble_sort(arr):
    # Initialize comparison and assignment counters
    comparisons = 0
    assignments = 0
    # Get the length of the array
    n = len(arr)
    # Outer loop for each pass through the array
    for i in range(n):
        # Inner loop for pairwise comparison and swapping
        for j in range(0, n-i-1):
            # Increment comparison counter
            comparisons += 1
            # Compare adjacent elements and swap if necessary
            if arr[j] > arr[j+1]:
                arr[j], arr[j+1] = arr[j+1], arr[j]
                # Increment assignment counter
                assignments += 1
    # Return the total comparisons and assignments
    return comparisons, assignments

# Function to perform Merge Sort on an array
def merge_sort(arr):
    # Initialize comparison and assignment counters
    comparisons = [0]
    assignments = [0]
    # Call the helper function for Merge Sort
    _merge_sort(arr, comparisons, assignments)
    # Return the total comparisons and assignments
    return comparisons[0], assignments[0]

# Helper function for Merge Sort
def _merge_sort(arr, comparisons, assignments):
    # Check if the array has more than one element
    if len(arr) > 1:
        # Calculate the midpoint
        mid = len(arr) // 2
        # Divide the array into two halves
        left_half = arr[:mid]
        right_half = arr[mid:]
        # Recursively call merge_sort on each half
        _merge_sort(left_half, comparisons, assignments)
        _merge_sort(right_half, comparisons, assignments)
        # Merge the two sorted halves
        i = j = k = 0
        while i < len(left_half) and j < len(right_half):
            comparisons[0] += 1
            if left_half[i] <= right_half[j]:
                arr[k] = left_half[i]
                i += 1
            else:
                arr[k] = right_half[j]
                j += 1
            assignments[0] += 1
            k += 1
        # Copy remaining elements from left_half, if any
        while i < len(left_half):
            arr[k] = left_half[i]
            i += 1
            assignments[0] += 1
            k += 1
        # Copy remaining elements from right_half, if any
        while j < len(right_half):
            arr[k] = right_half[j]
            j += 1
            assignments[0] += 1
            k += 1

# Function to perform Counting Sort on an array
def counting_sort(arr):
    # Initialize comparison and assignment counters
    comparisons = 0
    assignments = 0
    # Create a counts array to store frequency of each element
    counts = [0] * 101  # Considering elements are integers between 1 and 100
    # Count occurrences of each element in the input array
    for x in arr:
        counts[x] += 1
        assignments += 1
    # Reconstruct the sorted array using counts
    i = 0
    for j in range(1, 101):
        for _ in range(counts[j]):
            arr[i] = j
            assignments += 1
            i += 1
            comparisons += 1
    # Return the total comparisons and assignments
    return comparisons, assignments

# List of dataset filenames
datasets = ["shuffled.txt", "sorted.txt", "nearly-sorted.txt", "unsorted.txt", "nearly-unsorted.txt"]

# Iterate over each dataset
for dataset in datasets:
    # Read data from the file
    description, data = read_file(dataset)
    # Print dataset description and initial data
    print("Dataset:", description)
    print("Initial Data:", data)

    # Bubble Sort
    bubble_data = data.copy()  # Make a copy of the original data
    comparisons, assignments = bubble_sort(bubble_data)  # Perform Bubble Sort
    print("Bubble Sort - Comparisons:", comparisons, "Assignments:", assignments, "Sorted Data:", bubble_data)

    # Merge Sort
    merge_data = data.copy()  # Make a copy of the original data
    comparisons, assignments = merge_sort(merge_data)  # Perform Merge Sort
    print("Merge Sort - Comparisons:", comparisons, "Assignments:", assignments, "Sorted Data:", merge_data)

    # Counting Sort
    counting_data = data.copy()  # Make a copy of the original data
    comparisons, assignments = counting_sort(counting_data)  # Perform Counting Sort
    print("Counting Sort - Comparisons:", comparisons, "Assignments:", assignments, "Sorted Data:", counting_data)

    print("\n")  # Print a newline for better readability between datasets
```

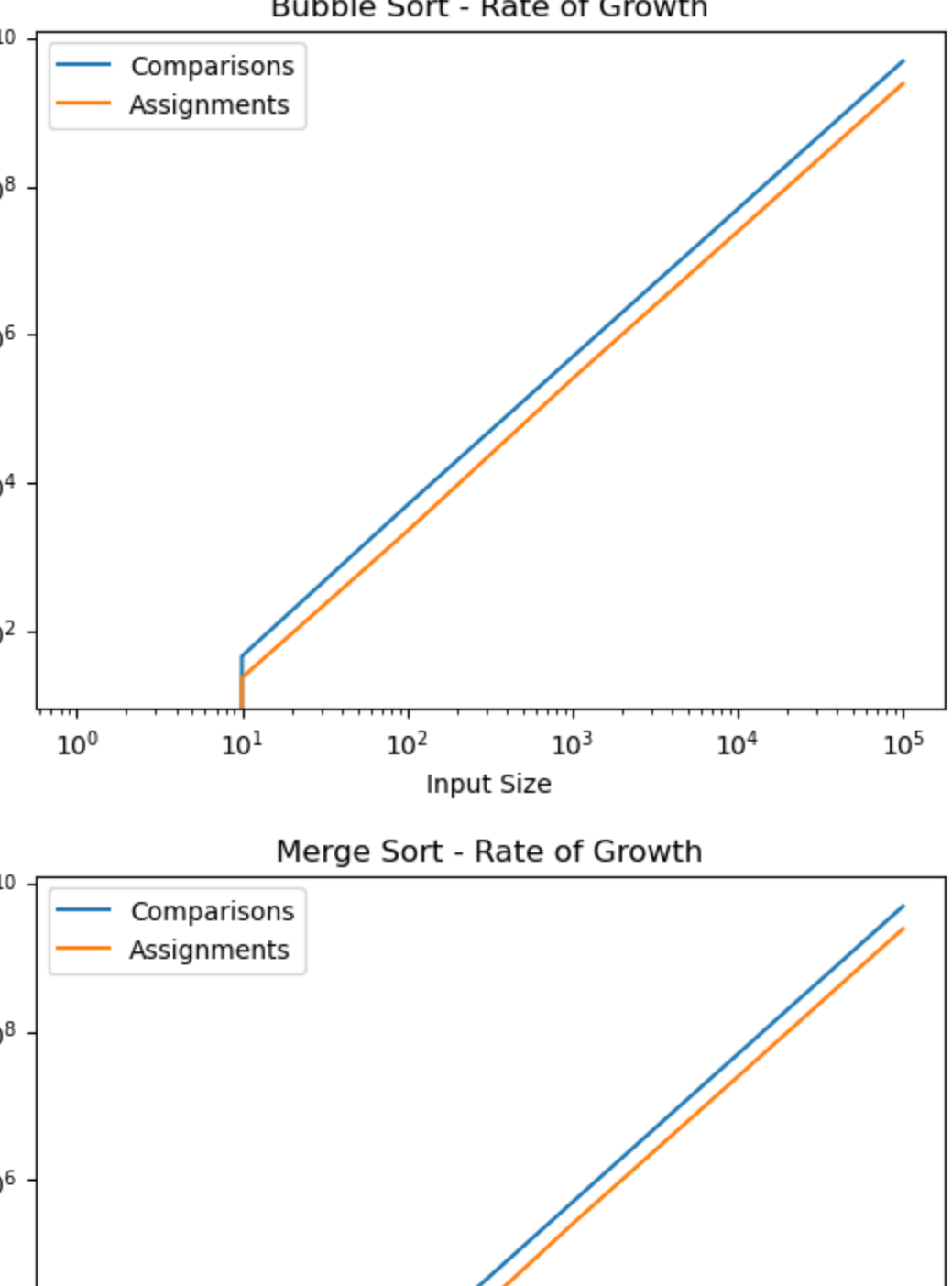
```
Dataset: Shuffled Integers 1..100
Initial Data: [11, 5, 42, 80, 6, 17, 36, 68, 70, 40, 30, 65, 15, 13, 19, 75, 79, 71, 95, 22, 25, 21, 47, 76, 90, 57, 94, 7, 32, 2, 81, 31, 82, 38,
97, 18, 16, 37, 24, 28, 64, 43, 33, 34, 83, 93, 46, 50, 27, 55, 12, 86, 77, 41, 62, 56, 96, 99, 60, 84, 49, 69, 3, 89, 61, 10, 1, 78, 48, 44, 67, 8
8, 100, 98, 54, 20, 8, 45, 26, 74, 51, 59, 87, 63, 52, 53, 66, 23, 72, 29, 4, 85, 9, 91, 35, 73, 92, 58, 14, 39]
Bubble Sort - Comparisons: 4950 Assignments: 2201 Sorted Data: [1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20, 21, 22, 23,
24, 25, 26, 27, 28, 29, 30, 31, 32, 33, 34, 35, 36, 37, 38, 39, 40, 41, 42, 43, 44, 45, 46, 47, 48, 49, 50, 51, 52, 53, 54, 55, 56, 57, 58, 59, 60,
61, 62, 63, 64, 65, 66, 67, 68, 69, 70, 71, 72, 73, 74, 75, 76, 77, 78, 79, 80, 81, 82, 83, 84, 85, 86, 87, 88, 89, 90, 91, 92, 93, 94, 95, 96, 97,
98, 99, 100]
Merge Sort - Comparisons: 537 Assignments: 672 Sorted Data: [1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20, 21, 22, 23, 24,
25, 26, 27, 28, 29, 30, 31, 32, 33, 34, 35, 36, 37, 38, 39, 40, 41, 42, 43, 44, 45, 46, 47, 48, 49, 50, 51, 52, 53, 54, 55, 56, 57, 58, 59, 60, 61,
62, 63, 64, 65, 66, 67, 68, 69, 70, 71, 72, 73, 74, 75, 76, 77, 78, 79, 80, 81, 82, 83, 84, 85, 86, 87, 88, 89, 90, 91, 92, 93, 94, 95, 96, 97, 98,
99, 100]
Counting Sort - Comparisons: 100 Assignments: 200 Sorted Data: [1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20, 21, 22, 23,
24, 25, 26, 27, 28, 29, 30, 31, 32, 33, 34, 35, 36, 37, 38, 39, 40, 41, 42, 43, 44, 45, 46, 47, 48, 49, 50, 51, 52, 53, 54, 55, 56, 57, 58, 59, 60,
61, 62, 63, 64, 65, 66, 67, 68, 69, 70, 71, 72, 73, 74, 75, 76, 77, 78, 79, 80, 81, 82, 83, 84, 85, 86, 87, 88, 89, 90, 91, 92, 93, 94, 95, 96, 97,
98, 99, 100]


Dataset: Sorted Integers 1..100
Initial Data: [1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20, 21, 22, 23, 24, 25, 26, 27, 28, 29, 30, 31, 32, 33, 34, 35, 3
6, 37, 38, 39, 40, 41, 42, 43, 44, 45, 46, 47, 48, 49, 50, 51, 52, 53, 54, 55, 56, 57, 58, 59, 60, 61, 62, 63, 64, 65, 66, 67, 68, 69, 70, 71, 72,
73, 74, 75, 76, 77, 78, 79, 80, 81, 82, 83, 84, 85, 86, 87, 88, 89, 90, 91, 92, 93, 94, 95, 96, 97, 98, 99, 100]
Bubble Sort - Comparisons: 4950 Assignments: 0 Sorted Data: [1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20, 21, 22, 23, 24,
25, 26, 27, 28, 29, 30, 31, 32, 33, 34, 35, 36, 37, 38, 39, 40, 41, 42, 43, 44, 45, 46, 47, 48, 49, 50, 51, 52, 53, 54, 55, 56, 57, 58, 59, 60, 61,
62, 63, 64, 65, 66, 67, 68, 69, 70, 71, 72, 73, 74, 75, 76, 77, 78, 79, 80, 81, 82, 83, 84, 85, 86, 87, 88, 89, 90, 91, 92, 93, 94, 95, 96, 97, 98,
99, 100]
Merge Sort - Comparisons: 316 Assignments: 672 Sorted Data: [1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20, 21, 22, 23, 24,
25, 26, 27, 28, 29, 30, 31, 32, 33, 34, 35, 36, 37, 38, 39, 40, 41, 42, 43, 44, 45, 46, 47, 48, 49, 50, 51, 52, 53, 54, 55, 56, 57, 58, 59, 60, 61,
62, 63, 64, 65, 66, 67, 68, 69, 70, 71, 72, 73, 74, 75, 76, 77, 78, 79, 80, 81, 82, 83, 84, 85, 86, 87, 88, 89, 90, 91, 92, 93, 94, 95, 96, 97, 98,
99, 100]
Counting Sort - Comparisons: 100 Assignments: 200 Sorted Data: [1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20, 21, 22, 23,
24, 25, 26, 27, 28, 29, 30, 31, 32, 33, 34, 35, 36, 37, 38, 39, 40, 41, 42, 43, 44, 45, 46, 47, 48, 49, 50, 51, 52, 53, 54, 55, 56, 57, 58, 59, 60,
61, 62, 63, 64, 65, 66, 67, 68, 69, 70, 71, 72, 73, 74, 75, 76, 77, 78, 79, 80, 81, 82, 83, 84, 85, 86, 87, 88, 89, 90, 91, 92, 93, 94, 95, 96, 97,
98, 99, 100]


Dataset: Nearly Sorted Integers 1..100
Initial Data: [2, 1, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 17, 18, 16, 19, 20, 21, 22, 23, 24, 25, 27, 26, 28, 29, 30, 31, 32, 34, 33, 35, 3
6, 37, 38, 39, 40, 41, 42, 43, 44, 45, 46, 47, 48, 49, 50, 51, 52, 53, 55, 54, 56, 57, 58, 59, 60, 61, 62, 63, 64, 65, 66, 68, 67, 70, 69, 71, 72,
74, 73, 75, 76, 77, 78, 79, 80, 81, 83, 82, 85, 84, 86, 87, 88, 90, 89, 91, 92, 93, 94, 95, 96, 97, 98, 99, 100]
Bubble Sort - Comparisons: 4950 Assignments: 12 Sorted Data: [1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20, 21, 22, 23, 2
4, 25, 26, 27, 28, 29, 30, 31, 32, 33, 34, 35, 36, 37, 38, 39, 40, 41, 42, 43, 44, 45, 46, 47, 48, 49, 50, 51, 52, 53, 54, 55, 56, 57, 58, 59, 60,
61, 62, 63, 64, 65, 66, 67, 68, 69, 70, 71, 72, 73, 74, 75, 76, 77, 78, 79, 80, 81, 82, 83, 84, 85, 86, 87, 88, 89, 90, 91, 92, 93, 94, 95, 96, 97,
98, 99, 100]
Merge Sort - Comparisons: 324 Assignments: 672 Sorted Data: [1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20, 21, 22, 23, 24,
25, 26, 27, 28, 29, 30, 31, 32, 33, 34, 35, 36, 37, 38, 39, 40, 41, 42, 43, 44, 45, 46, 47, 48, 49, 50, 51, 52, 53, 54, 55, 56, 57, 58, 59, 60, 61,
62, 63, 64, 65, 66, 67, 68, 69, 70, 71, 72, 73, 74, 75, 76, 77, 78, 79, 80, 81, 82, 83, 84, 85, 86, 87, 88, 89, 90, 91, 92, 93, 94, 95, 96, 97, 98,
99, 100]
Counting Sort - Comparisons: 100 Assignments: 200 Sorted Data: [1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20, 21, 22, 23,
24, 25, 26, 27, 28, 29, 30, 31, 32, 33, 34, 35, 36, 37, 38, 39, 40, 41, 42, 43, 44, 45, 46, 47, 48, 49, 50, 51, 52, 53, 54, 55, 56, 57, 58, 59, 60,
61, 62, 63, 64, 65, 66, 67, 68, 69, 70, 71, 72, 73, 74, 75, 76, 77, 78, 79, 80, 81, 82, 83, 84, 85, 86, 87, 88, 89, 90, 91, 92, 93, 94, 95, 96, 97,
98, 99, 100]


Dataset: Unsorted Integers 1..100
Initial Data: [100, 99, 98, 97, 96, 95, 94, 93, 92, 91, 90, 89, 88, 87, 86, 85, 84, 83, 82, 81, 80, 79, 78, 77, 76, 75, 74, 73, 72, 71, 70, 69, 68,
67, 66, 65, 64, 63, 62, 61, 60, 59, 58, 57, 56, 55, 54, 53, 52, 51, 50, 49, 48, 47, 46, 45, 44, 43, 42, 41, 40, 39, 38, 37, 36, 35, 34, 33, 32, 31,
30, 29, 28, 27, 26, 25, 24, 23, 22, 21, 20, 19, 18, 17, 16, 15, 14, 13, 12, 11, 10, 9, 8, 7, 6, 5, 4, 3, 2, 1]
Bubble Sort - Comparisons: 4950 Assignments: 4950 Sorted Data: [1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20, 21, 22, 23,
24, 25, 26, 27, 28, 29, 30, 31, 32, 33, 34, 35, 36, 37, 38, 39, 40, 41, 42, 43, 44, 45, 46, 47, 48, 49, 50, 51, 52, 53, 54, 55, 56, 57, 58, 59, 60,
61, 62, 63, 64, 65, 66, 67, 68, 69, 70, 71, 72, 73, 74, 75, 76, 77, 78, 79, 80, 81, 82, 83, 84, 85, 86, 87, 88, 89, 90, 91, 92, 93, 94, 95, 96, 97,
98, 99, 100]
Merge Sort - Comparisons: 356 Assignments: 672 Sorted Data: [1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20, 21, 22, 23, 24,
25, 26, 27, 28, 29, 30, 31, 32, 33, 34, 35, 36, 37, 38, 39, 40, 41, 42, 43, 44, 45, 46, 47, 48, 49, 50, 51, 52, 53, 54, 55, 56, 57, 58, 59, 60, 61,
62, 63, 64, 65, 66, 67, 68, 69, 70, 71, 72, 73, 74, 75, 76, 77, 78, 79, 80, 81, 82, 83, 84, 85, 86, 87, 88, 89, 90, 91, 92, 93, 94, 95, 96, 97, 98,
99, 100]
Counting Sort - Comparisons: 100 Assignments: 200 Sorted Data: [1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20, 21, 22, 23,
24, 25, 26, 27, 28, 29, 30, 31, 32, 33, 34, 35, 36, 37, 38, 39, 40, 41, 42, 43, 44, 45, 46, 47, 48, 49, 50, 51, 52, 53, 54, 55, 56, 57, 58, 59, 60,
61, 62, 63, 64, 65, 66, 67, 68, 69, 70, 71, 72, 73, 74, 75, 76, 77, 78, 79, 80, 81, 82, 83, 84, 85, 86, 87, 88, 89, 90, 91, 92, 93, 94, 95, 96, 97,
98, 99, 100]


Dataset: Nearly Unsorted Integers 1..100
Initial Data: [100, 99, 98, 97, 95, 96, 94, 93, 92, 91, 90, 89, 88, 87, 86, 84, 85, 83, 82, 81, 79, 80, 78, 77, 75, 74, 76, 73, 72, 70, 71, 69, 68,
67, 66, 65, 64, 63, 62, 61, 60, 59, 58, 56, 57, 55, 54, 53, 52, 51, 50, 49, 48, 47, 46, 45, 44, 43, 42, 41, 40, 39, 38, 37, 36, 35, 34, 33, 32, 31,
30, 29, 28, 27, 26, 25, 24, 23, 22, 21, 20, 19, 18, 16, 17, 15, 14, 13, 12, 11, 10, 9, 8, 7, 6, 5, 4, 3, 2, 1]
Bubble Sort - Comparisons: 4950 Assignments: 4942 Sorted Data: [1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20, 21, 22, 23,
24, 25, 26, 27, 28, 29, 30, 31, 32, 33, 34, 35, 36, 37, 38, 39, 40, 41, 42, 43, 44, 45, 46, 47, 48, 49, 50, 51, 52, 53, 54, 55, 56, 57, 58, 59, 60,
61, 62, 63, 64, 65, 66, 67, 68, 69, 70, 71, 72, 73, 74, 75, 76, 77, 78, 79, 80, 81, 82, 83, 84, 85, 86, 87, 88, 89, 90, 91, 92, 93, 94, 95, 96, 97,
98, 99, 100]
Merge Sort - Comparisons: 359 Assignments: 672 Sorted Data: [1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20, 21, 22, 23, 24,
25, 26, 27, 28, 29, 30, 31, 32, 33, 34, 35, 36, 37, 38, 39, 40, 41, 42, 43, 44, 45, 46, 47, 48, 49, 50, 51, 52, 53, 54, 55, 56, 57, 58, 59, 60, 61,
62, 63, 64, 65, 66, 67, 68, 69, 70, 71, 72, 73, 74, 75, 76, 77, 78, 79, 80, 81, 82, 83, 84, 85, 86, 87, 88, 89, 90, 91, 92, 93, 94, 95, 96, 97, 98,
99, 100]
Counting Sort - Comparisons: 100 Assignments: 200 Sorted Data: [1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20, 21, 22, 23,
24, 25, 26, 27, 28, 29, 30, 31, 32, 33, 34, 35, 36, 37, 38, 39, 40, 41, 42, 43, 44, 45, 46, 47, 48, 49, 50, 51, 52, 53, 54, 55, 56, 57, 58, 59, 60,
61, 62, 63, 64, 65, 66, 67, 68, 69, 70, 71, 72, 73, 74, 75, 76, 77, 78, 79, 80, 81, 82, 83, 84, 85, 86, 87, 88, 89, 90, 91, 92, 93, 94, 95, 96, 97,
98, 99, 100]
```

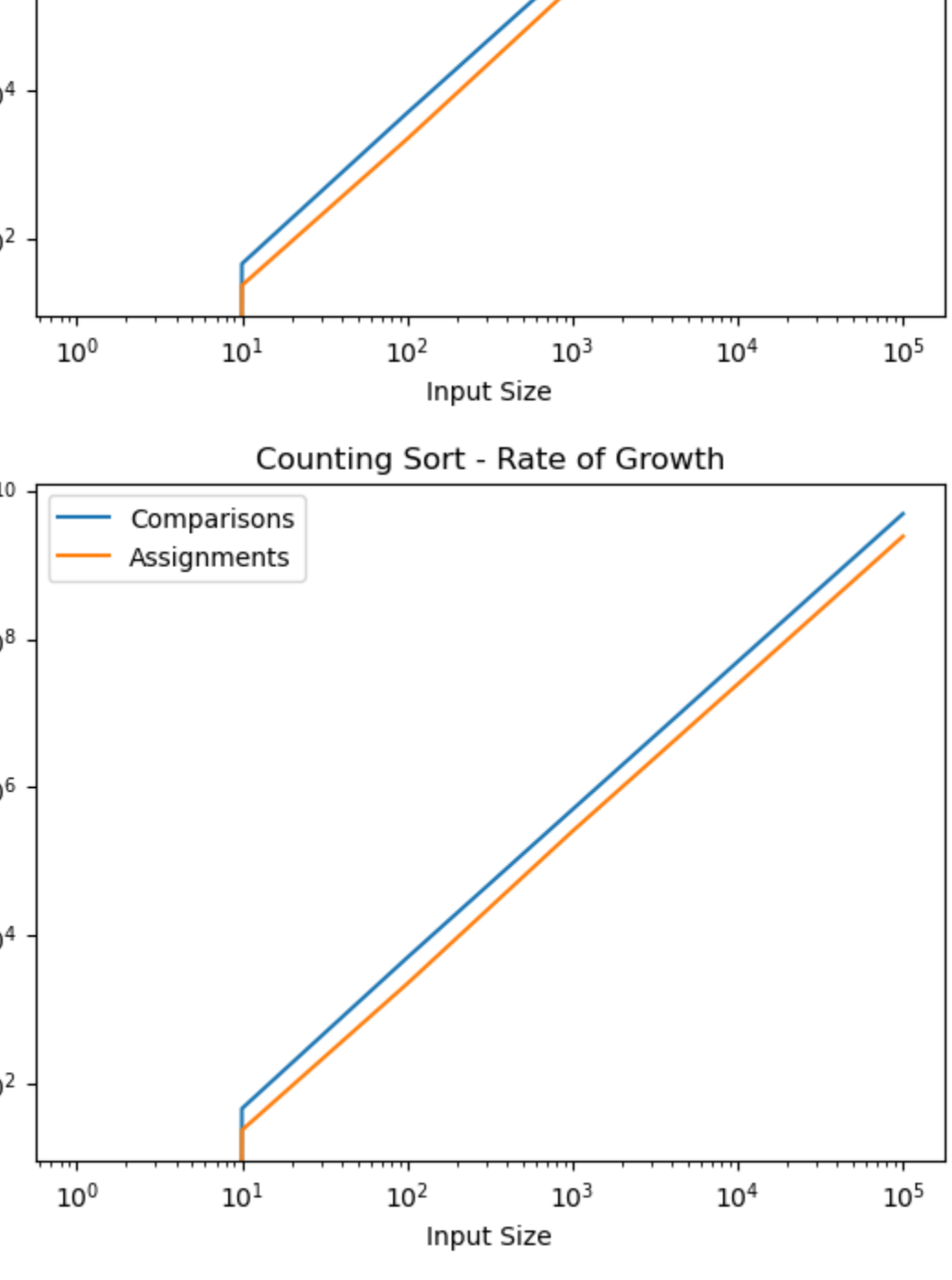
Part 5 - Analysis

In [8]:

```python
import matplotlib.pyplot as plt

# Function to generate data for plotting
def generate_data(filename):
    description, data = read_file(filename)
    sizes = [1, 10, 100, 1000, 10000, 100000]
    comparisons_data = []
    assignments_data = []

    for size in sizes:
        subset = data[:size]  # Extract subset of data for the given size
        # Perform sorting and store comparisons and assignments
        comparisons, assignments = bubble_sort(subset.copy())
        comparisons_data.append(comparisons)
        assignments_data.append(assignments)

    return sizes, comparisons_data, assignments_data

# Function to plot the data
def plot_data(x_values, y1_values, y2_values, algorithm_name):
    plt.plot(x_values, y1_values, label="Comparisons")
    plt.plot(x_values, y2_values, label="Assignments")
    plt.xscale('log')  # Use logarithmic scale for x-axis
    plt.yscale('log')  # Use logarithmic scale for y-axis
    plt.xlabel("Input Size")
    plt.ylabel("Count")
    plt.title(f"{algorithm_name} - Rate of Growth")
    plt.legend()
    plt.show()

# Perform analysis for each algorithm
for algorithm_name, sorting_function in [("Bubble Sort", bubble_sort), ("Merge Sort", merge_sort), ("Counting Sort", counting_sort)]:
    # Generate data for plotting
    sizes, comparisons_data, assignments_data = generate_data("one-million-randoms.txt")
    # Plot comparisons and assignments
    plot_data(sizes, comparisons_data, assignments_data, algorithm_name)
```







In [ ]: