Counting Sort

Array Type: Sorted, Comparisons: 0, Interchanges: 0Array Type: Inversely Sorted, Comparisons: 10000000, Interchanges: 0Array Type: Random, Comparisons: 10000000, Interchanges: 49995000Conclusion: Linear Sort performs well on already sorted or nearly sorted data but is less efficient for random data.

```java
private static int[] linearSort(int[] array) {
    // Implement Counting Sort logic
    int max = Arrays.stream(array).max().orElse(other:0) + 1;
    int[] count = new int[max];

    for (int num : array) {
        comparisons++;
        count[num]++;
    }

    int index = 0;
    for (int i = 0; i < max; i++) {
        while (count[i] > 0) {
            array[index++] = i;
            count[i]--;
            interchanges++;
            comparisons++;
        }
    }

    return array;
}
```

Bubble Sort

Array Type: Sorted, Comparisons: 0, Interchanges: 0Array Type: Inversely Sorted, Comparisons: 99990000, Interchanges: 49995000Array Type: Random, Comparisons: 99990000, Interchanges: 49995000Conclusion: Bubble Sort has poor performance, especially on random or inversely sorted data, making it less efficient for large datasets.

```java
private static int[] bubbleSort(int[] array) {
    // Implement Bubble Sort logic
    boolean swapped;
    do {
        swapped = false;
        for (int i = 1; i < array.length; i++) {
            comparisons++;
            if (array[i - 1] > array[i]) {
                swap(array, i - 1, i);
                interchanges++;
                swapped = true;
            }
        }
    } while (swapped);

    return array;
}
```

Quick Sort

- Array Type: Sorted, Comparisons: 24995000, Interchanges: 9999Array Type: Inversely Sorted, Comparisons: 24995000, Interchanges: 49995000Array Type: Random, Comparisons: 24995000, Interchanges: 49995000Conclusion: Quick Sort generally performs well on various types of data, providing relatively good efficiency for large datasets.

```java
private static int[] quickSort(int[] array, int low, int high) {
    // Implement Quick Sort logic
    if (low < high) {
        int partitionIndex = partition(array, low, high);

        quickSort(array, low, partitionIndex - 1);
        quickSort(array, partitionIndex + 1, high);
    }
    return array;
}

private static int partition(int[] array, int low, int high) {
    int pivot = array[high];
    int i = low - 1;

    for (int j = low; j < high; j++) {
        comparisons++;
        if (array[j] < pivot) {
            i++;
            swap(array, i, j);
            interchanges++;
        }
    }

    swap(array, i + 1, high);
    interchanges++;

    return i + 1;
}
```

Algorithm	Array Type	Relative Run Time (ns)	Number of Comparisons	Number of Interchanges
Linear Sort - Random	Random	1086800	10000	9993
Linear Sort - Sorted	Sorted	430900	9999	0
Linear Sort - Inversely Sorted	Sorted	1173100	10000	10000
Bubble Sort - Random	Random	64867800	99320067	24947057
Bubble Sort - Sorted	Sorted	88600	9999	0
Bubble Sort - Inversely Sorted	Sorted	57497500	99990000	49995000
Quick Sort - Random	Random	2087200	182049	63837
Quick Sort - Sorted	Sorted	51839500	49995000	49995000
Quick Sort - Inversely Sorted	Sorted	42652500	49995000	25000000

Conclusion:
Among the three sorting algorithms, Quick Sort shows better efficiency across different types of data, making it a more versatile choice.Linear Sort is efficient for already sorted or nearly sorted data but less effective for random data.Bubble Sort exhibits poor performance, especially on random or inversely sorted data.In practical scenarios, Quick Sort is often preferred for general-purpose sorting due to its average-case time complexity of O(n log n) and good performance across diverse datasets.