Bubble Sort:

For bubble sorting I ran a traditional algorithm. I ran ‘n’ passes and each pass pushed the largest element to the end of the array. The number of comparisons is computed as 1+2+3+…+(n-q). There should be more movements than comparisons. This is because each swap requires three moves and not all comparisons lead to movement.

Selection Sort:

For selection sort I used a traditional method. I ran ‘n’ passes, each pass would swap the beginning of the unsorted array with the next smallest element. The time complexity of this algorithm is O(N2). There should be more comparisons than movement. This is because each loop goes through the entire unsorted array before it makes one movement.

Insertion Sort:

For Insertion sort I used a traditional method. I ran ‘n’ passes, each pass compare elements until one was found out of order. This made it so where there wouldn’t be any redundant movement if the array happened to be sorted in segments. . The time complexity of this is O(N2). There should be more comparisons than movement. The algorithm will always compare at least 2 elements to each movement.

Merge Sort:

For Merge sort I used a traditional method. I ran ‘n’ passes, each pass would split the array until it was down to individual elements. It would then merge the elements and sort them. This continues until the array is fully merged back together. The time complexity of this is O(N log(N)). There should be more movement than comparisons.

Quick Sort:

For Quick sort I used a traditional method. I ran ‘n’ passes, each pass would select a pivot element and move all element greater than or less than it to its left or right respectively. This will place the pivot in or closer to its correct position. The time complexity of this algorithm is O(N log(N)). There should be more comparisons than movement in most cases. In the worst case of this algorithm, it will have equal comparisons and movements.