Task 1:

Average runtime of randomly generated arrays:

```
Array Quick Sort has an average runtime of 17 ms
Array Quick Sort(2) has an average runtime of 17 ms
Array Merge Sort has an average runtime of 158 ms
Array Merge Sort(2) has an average runtime of 17 ms
```

The only standout of the group was the original Merge Sort, with a runtime much greater than all the others. Running multiple trials showed a slight difference between Quick Sort, Quick Sort 2, and Merge Sort 2. Merge Sort 2 seemed to be a few ms slower than both Quick Sorts, while Quick Sort and Quick Sort 2 were almost always equal. Quick Sort 2 would sometimes be slightly faster(1ms) than Quick Sort, but nothing noticeable.

Task 2:

-Average runtime of randomly generated arrays-

```
Array Quick Sort has an average runtime of 128 ms
Array Quick Sort(2) has an average runtime of 11 ms
Array Quick Sort(3) has an average runtime of 20 ms
Array Quick Sort(4) has an average runtime of 9 ms
```

-Average runtime of sorted arrays-

```
Array Quick Sort has an average runtime of 344 ms
Array Quick Sort(2) has an average runtime of 115 ms
Array Quick Sort(3) has an average runtime of 152 ms
Array Quick Sort(4) has an average runtime of 101 ms
```

-Average runtime of reversely sorted arrays-

```
Array Quick Sort has an average runtime of 352 ms
Array Quick Sort(2) has an average runtime of 121 ms
Array Quick Sort(3) has an average runtime of 164 ms
Array Quick Sort(4) has an average runtime of 99 ms
```

The original Quick Sort was much slower than the rest, which was the only one that did not check if the subarray had been sorted at all. This feature, common to Quick Sort 3 and 4, caused a significant speed up. Quick Sort 2, which was missing the sub array feature, but would use insertion sort if the sorting problem is small, performed very well. It performed better than Quick Sort 3, which only had the sub array feature. The best of the group was Quick Sort 4, which contained both features and caused a significant speed up compared to the rest. These patterns held true for all three kinds of arrays, showing the performance was independent on the type of input array.

Task 3:

-Average runtime of randomly generated arrays-

Array Quick Sort has an average runtime of 128 ms
Array Quick Sort(4) has an average runtime of 10 ms
Array Quick Sort(5) has an average runtime of 8 ms
Array Quick Sort(6) has an average runtime of 10 ms
Array Heap Sort has an average runtime of 136 ms

-Average runtime of reversely sorted arrays-

Array Quick Sort has an average runtime of 17 ms
Array Quick Sort(4) has an average runtime of 7 ms
Array Quick Sort(5) has an average runtime of 7 ms
Array Quick Sort(6) has an average runtime of 7 ms
Array Heap Sort has an average runtime of 131 ms

-Average runtime of Organ-Pipe shaped arrays-

Array Quick Sort has an average runtime of 139 ms
Array Quick Sort(4) has an average runtime of 10 ms
Array Quick Sort(5) has an average runtime of 8 ms
Array Quick Sort(6) has an average runtime of 11 ms
Array Heap Sort has an average runtime of 149 ms

I will preface this section by pointing out an issue with the addition of Quick Sort 5 and Quick Sort 6. Changing the pivots very slightly caused Stack Overflow Errors that could not be dealt with. Simple changes such as changing int pivot = arr[(high+low)/2] to arr[(high+low)/3]; would cause a stack overflow, so the more elaborate pivots would only cause overflow easier.

The patterns shown were still clear, showing that Heap Sort and the original Quick Sort were much slower than the upgraded Quick Sorts. The upgraded Quick Sorts were some of the fastest running times the project will see, due to the addition of so many features to speed them up that had been accumulated from the previous tasks. These results disagree with sorting-algorithms.com, which claims that heap sort is often the winner for reversely sorted arrays. With an input of this size, it will spend way too much time sorting compared to the upgraded Quick Sorts.

Task 4:

--K Exchange Sorting--K = 100

```
Testing for Array 1:Quick Sort: 149 ms|Heap Sort: 19 ms|Merge Sort: 224 ms|nMerge Sort: 6 ms|Insertion Sort: 6278361 ms|
BUILD STOPPED (total time: 188 minutes 3 seconds)
```

The datasheet will show an insertion sort time of 0 seconds for all trials. This is due to the fact that insertion sort was terrible for all large data sets, having a running time of over an hour per data point. Instead of runnning the task 4 for an unrealistic amount of time, it was commented out and deemed unusable and too slow.

*Other K trials omitted from here, but are attached in the datasheet.

-K Exchange Sorting, K = 100-

```
Array Quick Sort has an average runtime of 1 ms
Array Heap Sort has an average runtime of 16 ms
Array Merge Sort has an average runtime of 235 ms
Array Natural Merge Sort has an average runtime of 7 ms
Array Insertion Sort has an average runtime of 0 ms
```

-K Distance Sorting, K = 320-

```
Array Quick Sort has an average runtime of 1 ms
Array Heap Sort has an average runtime of 16 ms
Array Merge Sort has an average runtime of 219 ms
Array Natural Merge Sort has an average runtime of 7 ms
Array Insertion Sort has an average runtime of 0 ms
```

Each K exchange trial had very similar patterns, showing that Natural Merge Sort in second, and the Quick Sort being the clear winner over all others. The speed shown here almost matched the quickest speeds shown from the upgraded Quick Sort. Regular Merge Sort and regular Quick Sort were slow, and almost unusable compared to the speed of the other 3. The only sorting algorithm that came close to the upgraded Quick Sort was the Natural Merge Sort, which while fast, was more than double the time of the upgraded Quick Sort. This matches the sorting-algorithms.com order vaguely, although their Quick Sort is not an upgraded one like ours is.