Code Review Rory McDermid 2566302M:
Git Link (If code isn't working): https://github.com/RoryMcDermid/ADS2_AE1.git
Code is run from the main in TimeSortingAlgorithms, the data files should be stored in a folder called Test_Data in the project folder

The sorts are referred to as 1A,1B,1C and 1D. This order is the way they are referenced in the Assessed Exercise 1 document.

int10 :

Outputs:

```
(k=4)
Execution time for Insertion Sort is : 2700 nanoseconds
Execution time for Sort 1A is : 5200 nanoseconds
Execution time for Sort 1B is : 4900 nanoseconds
Execution time for Sort 1C is : 7000 nanoseconds
Execution time for Sort 1D is : 6200 nanoseconds
(k=5)
Execution time for Sort 1B is : 6200 nanoseconds
(k=3)
Execution time for Sort 1B is : 6300 nanoseconds
```

Overall in int10.txt the insertion sort is the fastest by a decent margin, though this was to be expected as an insertion sort works best on a small list. As far as the quicksorts go A and fine tuned B are close to fastest, with the less tuned B and D in second place and C taking third place.

int100 :

Outputs:

```
(k=30)
Execution time for Insertion Sort is : 35400 nanoseconds
Execution time for Sort 1A is : 23300 nanoseconds
Execution time for Sort 1B is : 69900 nanoseconds
Execution time for Sort 1C is : 35100 nanoseconds
Execution time for Sort 1D is : 27000 nanoseconds
(k=50)
Execution time for Sort 1B is : 25700 nanoseconds
(k=70)
Execution time for Sort 1B is : 26300 nanoseconds
```

In int100.txt the insertion sort falls drastically behind compared to the smaller list, with sort C (the largest logic wise) barely beating it out. A and D are close along with a fine tuned B though a poorly tuned B nearly triples the fastest times.

int1000:

Outputs:

```
(k=300)
Execution time for Insertion Sort is : 1454900 nanoseconds
Execution time for Sort 1A is : 263000 nanoseconds
Execution time for Sort 1B is : 2922200 nanoseconds
Execution time for Sort 1C is : 697600 nanoseconds
Execution time for Sort 1D is : 288800 nanoseconds
(k=600)
Execution time for Sort 1B is : 1396200 nanoseconds
(k=800)
Execution time for Sort 1B is : 1189500 nanoseconds
```

In int1000.txt the speed clearly show a defined list of speed, with A and D tied for first, C taking second place, tuned B taking third, Insertion taking fourth and poorly tuned B taking fifth.

The outliers can be mostly accounted for, with the insertion sort's long run time being down to it's poor optimisation for longer lists, B's time draws attention to what is likely coding issues on my end, as it shouldn't have been much longer than the other quicksorts.

C takes a longer time mostly due to the extra code of Median adding extra complexity to the code, this may also have lead to an unseen coding oversight which has caused such an increase in time but this is unlikely due to the times seen being relatively close to the leaders of the pack. It also from testing during development manages to get the outcome in a smaller number of cycles, but this precision is unable to make up for the extra code that must be run to get the same results.

As for A and D, the only difference between them is how their partition code works (QuickSorts.Partition and QuickSorts.Partition_3_Way), and with multiple tests they both usually ended up very close, with A usually but not always beating D. This slight difference is likely due to D's purpose being sorting lists with duplicates, and the intx.txt lists have none. However when 1000 long lists of numbers 0-99 (inclusive) were ran through A and D, the results were unsurprising:

Outputs:

```
Execution time for Sort 1A is : 312800 nanoseconds
Execution time for Sort 1D is : 214000 nanoseconds

Execution time for Sort 1A is : 328300 nanoseconds
Execution time for Sort 1D is : 212000 nanoseconds
```

In this scenario D was able to run at two thirds the speed of A, or about 75% of it's speed on int1000.txt, this is entirely due to it's code being designed to remove extra recursions due to duplicate values, this can be taken to it's extreme by setting the list to 0-9 and the values become:

Outputs:

```
Execution time for Sort 1A is : 887800 nanoseconds
Execution time for Sort 1D is : 119300 nanoseconds

Execution time for Sort 1A is : 745400 nanoseconds
Execution time for Sort 1D is : 105700 nanoseconds
```

This shows that A, while being faster with no duplicates, cannot compare to D when it comes to lists with duplicates involved, with A's time shooting up past C and D's time being cut in half. Overall this shows that while A has a shorter time in some lists, it's worst time complexity is far greater than D's, while D has a best time complexity far below that of A.