Joey Ferguson - 21720040
Rohail Mall - 21620052
Fin Msiska - 21720014
Jonathan Phiri - 21720042

Team Project 2

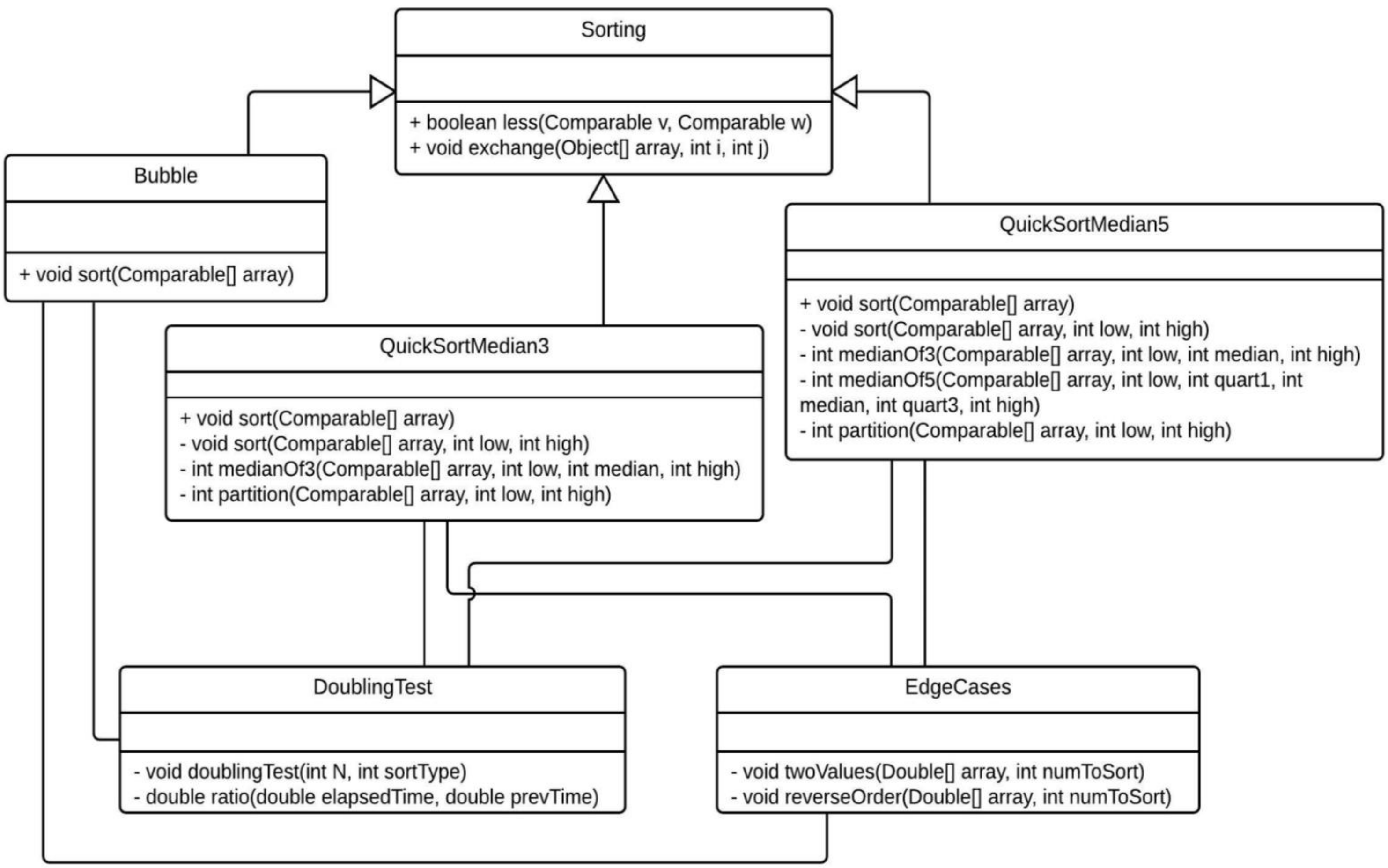
27 May 2017

Documentation

The Problem

Sorting is a fundamental part of computer science. Lists, dictionaries, arrays, and more all use sorting to make data easier to access or arrange data so humans can interpret it. Most of the time, efficiency isn't relevant, because the number of items is so small. But as the number of items grows, there becomes an increasing need to sort the items with an efficient algorithm. After all, you don't want to be waiting hours for your computer to sort something that could be sorted in a few seconds with a better algorithm.

Class Diagram



The class diagram is self-explanatory. The Bubble, QuickSortMedian3, and QuickSortMedian5 classes inherit the Sorting class, mainly so there's no code reuse. Then, all three sorting algorithms are related by association to DoublingTest and EdgeCases for testing and printing out results.

Joey Ferguson - 21720040
Rohail Mall - 21620052
Fin Msiska - 21720014
Jonathan Phiri - 21720042

Team Project 2

27 May 2017

<u>Code</u> - Note: main() functions are included in source code, but not necessary here.

Sorting Class

We knew we would need to do a lot of comparisons and exchanges while implementing our sorting algorithms, so we decided to create a class that all the other classes could inherit so we wouldn't have to reuse code.

```
public class Sorting {

    // Returns true if v is less than w, false if w is less than v
    static boolean less(Comparable v, Comparable w) {
        return (v.compareTo(w) < 0);
    }

    // Swaps array[i] and array[j]
    static void exchange(Object[] array, int i, int j) {
        Object swap = array[i];
        array[i] = array[j];
        array[j] = swap;
    }
}
```

Bubble Class

This is a very basic implementation of a Bubble Sort. It's not supposed to be efficient. It was solely used as a control to show what a better sorting algorithm can do as opposed to a lazy approach.

```
public class Bubble extends Sorting {

    public static void sort(Comparable[] array) {
        boolean sorted = false;
        while (!sorted) {
            sorted = true;
            for (int i = 0; i < array.length - 1; i++) {
                if (less(array[i + 1], array[i])) {
                    sorted = false;
                    exchange(array, i, i + 1);
                }
            }
        }
    }
}
```

Quick Sort Median 3 Class

This class is a modification of Quick.java, which is provided in the standard library. When sort gets called, it will first do a sampling of 3 indices and find the median, and use that as the partition value. Then it repeats this process recursively, finding a new median with each iteration. This greatly improved the performance since it wasn't creating lopsided arrays to sort.

Joey Ferguson - 21720040
Rohail Mall - 21620052
Fin Msiska - 21720014
Jonathan Phiri - 21720042

Team Project 2

27 May 2017

```
public class QuickSortMedian3 extends Sorting {

    // Public sort method callable from outside the class
    static void sort(Comparable[] array) {
        sort(array, 0, array.length - 1);
    }

    // Sorts the array from low to high indices
    private static void sort(Comparable[] array, int low, int high) {
        if (high <= low) { return; }

        int median = low + (high - low)/2;

        median = medianOf3(array, low, median, high);
        exchange(array, low, median);

        int i = partition(array, low, high);        // partition data into parts
        sort(array, low, i-1);                       // recursively sort lower part
        sort(array,i+1, high);                       // recursively sort higher part
    }

     // Returns the median of three indices passed in
     private static int medianOf3(Comparable[] array, int low, int median, int high) {
        if (less(array[low], array[median])) {
            if (less(array[median], array[high])) {
                return median;
            }
            else {
                if (less(array[low], array[high])) {
                    return high;
                }
                else {
                    return low;
                }
            }
        } else {
            if (less(array[low], array[high])) {
                return low;
            }
            else {
                if (less(array[median], array[high])) {
                    return high;
                }
                else {
                    return median;
                }
            }
        }
    }

    // Partition the sub-array so that array[lo..j-1] < array[j] < array[j+1..hi]
    private static int partition(Comparable[] array, int low, int high) {
        int i = low;
        int j = high + 1;

        Comparable partition = array[low];       // pivot

        while(true) {

            while(less(array[++i], partition)) {
                if (i == high) break;
            }
```

```
            while(less(partition, array[--j])) {
                if (j == low) break;
            }

            if(i >= j) { break; }

            exchange(array, i, j);
        }

        exchange(array, low, j);
        return j;
    }
}
```

Quick Sort Median 5 Class

This class was very similar to Quick Sort Median 3, but instead of picking three indices and finding the median, it finds the median of five indices (and then repeats for every sub-array recursively). Functions that were the same as the Quick Sort Median 3 class were not shown again.

```
public abstract class QuickSortMedian5 extends Sorting {

    // Public sort method callable from outside the class
    static void sort(Comparable[] array) {
        sort(array, 0, array.length - 1);
    }

    // Sorts the array from low to high indices
    private static void sort(Comparable[] array, int low, int high) {
        if (high <= low) { return; }

        int median = low + (high - low)/2;

        median = medianOf5(array, low, low + (median - low)/2, median,
                                      median + (high - median)/2, high);
        exchange(array, low, median);

        int j = partition(array, low, high);        // partition data into parts
        sort(array, low, j-1);                      // recursively sort lower part
        sort(array, j+1, high);                     // recursively sort higher part
    }

    private static int medianOf3(Comparable[] array, int low, int median, int high) {
         /* same as QuickSortMedian3 class */
    }

    // Returns the best three candidates for the medianOf3 function.
    private static int medianOf5(Comparable[] a, int l, int q1, int m, int q3, int h){
        int leftMin, leftMax;
        int rightMin, rightMax;
        int c1, c2, c3;

        if (less(array[low], array[quart1])) {
            leftMin = low;
            leftMax = quart1;
        } else {
            leftMin = quart1;
            leftMax = low;
        }
```

```
        if (less(array[high], array[quart3])) {
            rightMin = high;
            rightMax = quart3;
        } else {
            rightMin = quart3;
            rightMax = high;
        }

        if (less(array[rightMax], array[leftMax])) {
            c1 = rightMax;
        } else {
            c1 = leftMax;
        }

        if (less(array[leftMin], array[rightMin])) {
            c2 = rightMin;
        } else {
            c2 = leftMin;
        }

        c3 = median;

        return medianOf3(array, c1, c2, c3);
    }

    private static int partition(Comparable[] array, int low, int high) {
         /* same as QuickSortMedian3 class */
    }
}
```

Doubling Test Class

This is a pretty straightforward class that helped to test all sorting algorithms at once. It mainly keeps track of timers, calculations, and creates the array of doubles (which doubles each time through).

```
private static void doublingTest(int N, int sortType) {
    DecimalFormat rt = new DecimalFormat("#.####");
    DecimalFormat et = new DecimalFormat("##.##");

    double elapsedTime;
    double prevTime = 0.0;

    for (int numToSort = N; numToSort <= (64 * N); numToSort *= 2) {
        Double[] randomArray = new Double[numToSort];
        for (int randoms = 0; randoms < numToSort; randoms++) {
            randomArray[randoms] = StdRandom.uniform();
        }

        Stopwatch timer = new Stopwatch();
        if      (sortType == 0) {        Insertion.sort(randomArray); }
        else if (sortType == 1) {        Selection.sort(randomArray); }
        else if (sortType == 2) {            Shell.sort(randomArray); }
        else if (sortType == 3) {           Bubble.sort(randomArray); }
        else if (sortType == 4) { QuickSortMedian3.sort(randomArray); }
        else if (sortType == 5) { QuickSortMedian5.sort(randomArray); }
        else if (sortType == 6) {            Quick.sort(randomArray); }
        elapsedTime = timer.elapsedTime();

        double ratio = 0.0;
```

```
        // Printing stats and headers
        if (numToSort != N) { ratio = ratio(elapsedTime, prevTime); }
        else { StdOut.println("Items\t\t Time\t\t Ratio"); }

        int numLength = String.valueOf(numToSort).length();
        if (numLength <= 7) {
            StdOut.println(numToSort + "\t\t" + " " + et.format(elapsedTime) +
                                       "\t\t" + " " + rt.format(ratio));
        } else {
            StdOut.println(numToSort + "\t" + " " + et.format(elapsedTime) +
                                       "\t\t" + " " + rt.format(ratio));
        }

        prevTime = elapsedTime;
    }
    StdOut.println();
}

private static double ratio(double elapsedTime, double prevTime) {
    return (elapsedTime/prevTime);
}
```

Edge Case Class

This class was helpful for comparing how the algorithms held up when put under specific conditions. We tested if the array was in the exact opposite (reverse) order and if the array only had 2 possible values in it (useful for sorting true/false, 1/0). We made this as modular as possible, so you can easily add more edge cases if you wanted to.

```
private static void twoValues(Double[] array, int numToSort) {
    for (int i = 0; i < numToSort; i++) {
        if (i % 2 == 0) {
            array[i] = 1.0;
        }
        else {
            array[i] = 2.0;
        }
    }
}

// Arranges the array in the exact opposite order of sorted
private static void reverseOrder(Double[] array, int numToSort) {
    for (int i = 0; i < numToSort; i++) {
        array[i] = StdRandom.uniform();
        // StdOut.println(array[i]);
    }

    Arrays.sort(array, Collections.reverseOrder());
}
```

Joey Ferguson - 21720040
Rohail Mall - 21620052
Fin Msiska - 21720014
Jonathan Phiri - 21720042

Team Project 2

27 May 2017

Results

On the left, you can see the results of the three algorithms available for use in the standard library. Clearly, Insertion sort and Selection sort aren't the best algorithms out there, but they're good enough if you're sorting a small number of items. Shell sort gets much better compared to Insertion and Selection, especially as the number of items goes up.

But what we find even more interesting is how drastically different the results on the right are. Bubble sort is clearly the worst implementation out of the six. It took more than 4 minutes to sort 160,000 numbers, where both the Quick Sort implementations could do it in less than a tenth of a second.

Insertion Sort

Items	Time	Ratio
10000	0.25	0
20000	0.91	3.6855
40000	2.69	2.9387
80000	12.46	4.6392
160000	64.95	5.2123

Selection Sort

Items	Time	Ratio
10000	0.14	0
20000	0.62	4.4643
40000	2.31	3.6912
80000	9.98	4.3251
160000	57.61	5.7741

Shell Sort

Items	Time	Ratio
10000	0.01	0
20000	0.01	0.6667
40000	0.02	2.25
80000	0.04	2.3889
160000	0.1	2.4419

Bubble Sort

Items	Time	Ratio
10000	0.56	0
20000	2.71	4.8551
40000	11.47	4.2255
80000	49.47	4.3133
160000	267.53	5.4086

Quick Sort Median 3

Items	Time	Ratio
10000	0.01	0
20000	0.01	1.125
40000	0.04	4.4444
80000	0.02	0.425
160000	0.04	2.1765

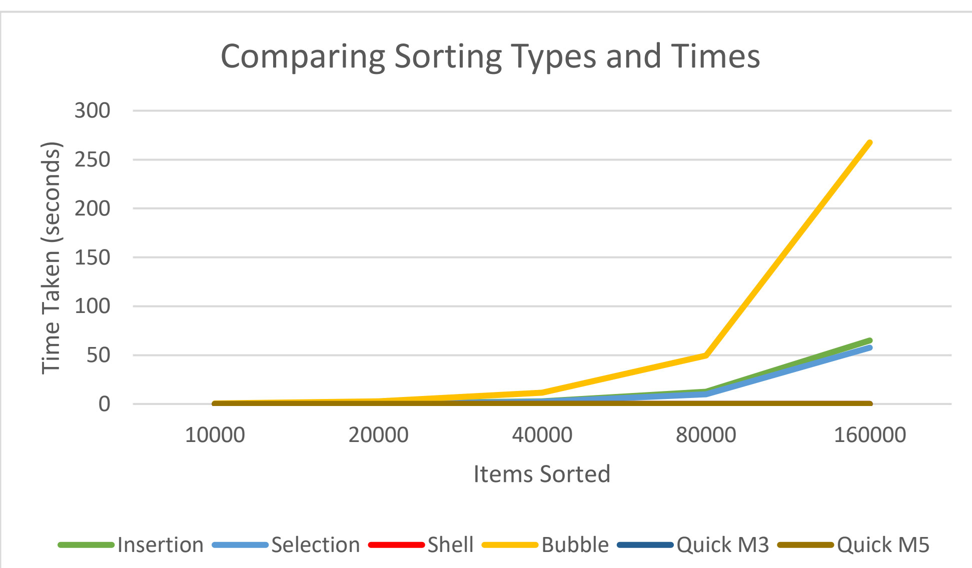
Quick Sort Median 5

Items	Time	Ratio
10000	0.01	0
20000	0.02	3.3333
40000	0.05	2.65
80000	0.03	0.5283
160000	0.06	2.0714

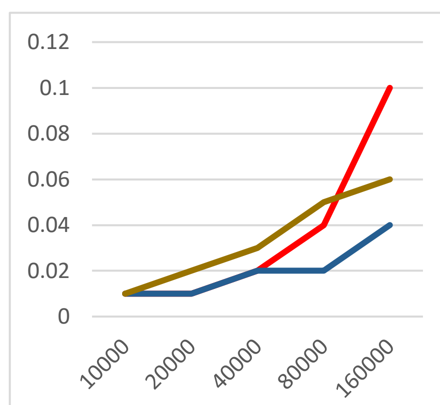
Above, we can see that the Insertion, Selection, and Bubble Sorts are at their limits. They could be used in some situations, but it's impractical to test them further now because they would take too long.

Also, the ratios of Insertion, Selection, and Bubble Sorts is consistently 4 or 5, which means that doubling the number of items sorted takes 4 or 5 times longer than the previous amount. This is unacceptable for a sorting algorithm, especially with many items.

As you can see, the Bubble Sort is clearly much slower than the other algorithms, and the Quick Sorts and Shell Sort don't even register on this graph scale.



The zoomed in version shows the other results here, but they the time they take to sort these are negligible compared to the others.



Joey Ferguson - 21720040
Rohail Mall - 21620052
Fin Msiska - 21720014
Jonathan Phiri - 21720042

Comparing Shell Sort, Quick Sort Median 3, and Quick Sort Median 5

It's also clear that Shell, Quick Median 3, and Quick Median 5 are not even close to their limits. Let's see where those limits are.

Following the same format as earlier, we ran the same tests, but this time starting with 1 million and doubling until reaching 32 million items sorted (as opposed to 10,000 to 160,000 before). This one really pushed all the sorting algorithms to the point where it wasn't nearly as fast.

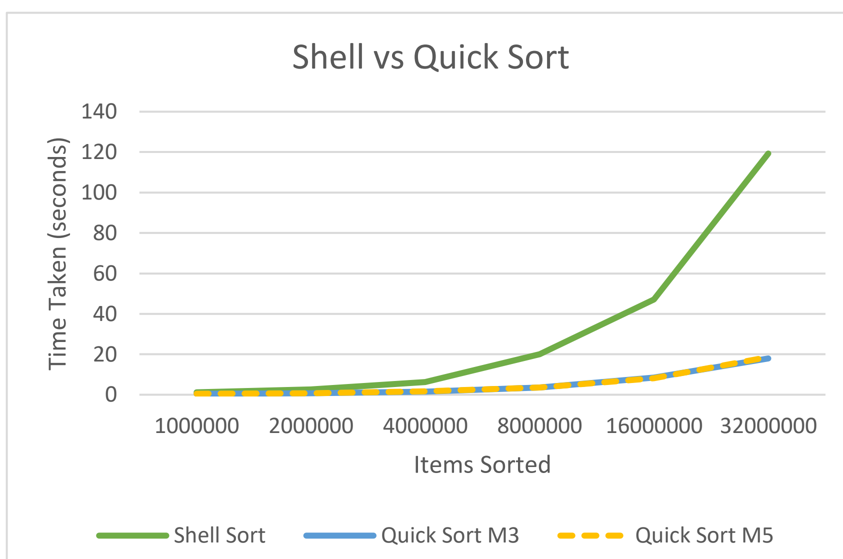
```
Shell Sort
Items        Time      Ratio
1000000      1.19      0
2000000      2.68      2.25
4000000      6.23      2.3214
8000000      19.99     3.2102
16000000     47.1      2.3565
32000000     119.31    2.5331

Quick Sort Median 3
Items        Time      Ratio
1000000      0.44      0
2000000      0.66      1.508
4000000      1.52      2.3065
8000000      3.63      2.3895
16000000     8.43      2.3219
32000000     17.93     2.1264

Quick Sort Median 5
Items        Time      Ratio
1000000      0.48      0
2000000      0.72      1.4804
4000000      1.6       2.2354
8000000      3.62      2.2523
16000000     8.12      2.2459
32000000     18.59     2.2896
```

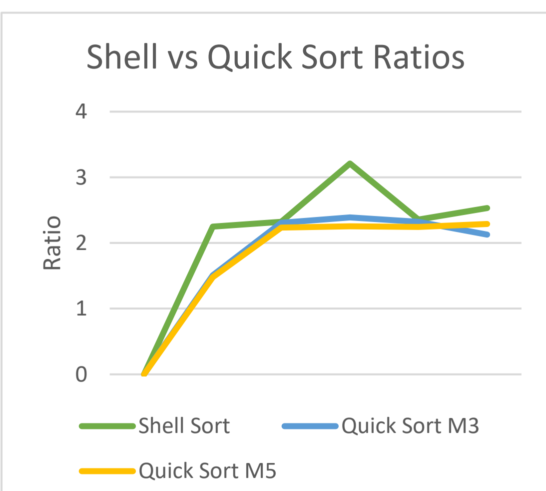
As you can see, comparing these algorithms gets a lot more interesting when you sort millions of items instead of only tens of thousands.

We ran this test several times, and Quick Sort Median 3 was the clear winner most of the time, with Quick Sort Median 5 not far behind. This surprised us, because we thought creating more evenly-spaced partitions would result in an algorithm that's faster.

But, we suppose the Median of 3 method was sufficient to where it found a "good enough" median to partition with, and it has considerably less calls than the Median of 5, which explains why it's faster.

Finally, we thought it was quite interesting how similar the ratios were for these algorithms. All of the algorithms consistently had ratios of 2.2 – 2.3. Considering the amount of work is much more than 2.2 or 2.3 times more than the previous amount, we'd say the Quick Sort algorithms scale pretty well.





Joey Ferguson - 21720040
Rohail Mall - 21620052
Fin Msiska - 21720014
Jonathan Phiri - 21720042

Finding the Limit of Quick Sort

Sorting 32 million items in 18 seconds seems pretty crazy. But, we decided to take it one step further and find the true limit of Quick Sort. We found that Quick Sort could sort 64 million elements in a somewhat reasonable amount of time. However, when trying to test any of the algorithms further, we get an Out of Memory Error. So, for the hardware available to us, we found the limit of these algorithms.

```
Exception in thread "main" java.lang.OutOfMemoryError: GC overhead limit exceeded
    at java.lang.Double.valueOf(Double.java:519)
    at DoublingTest.doublingTest(DoublingTest.java:19)
    at DoublingTest.main(DoublingTest.java:68)
```

So, if we can't go further than 64 million items, we should find out which algorithm sorts the 64 million the fastest. We decided to add the original Quick Sort in (available in algs4) to see which of the three Quick Sorts would perform best.

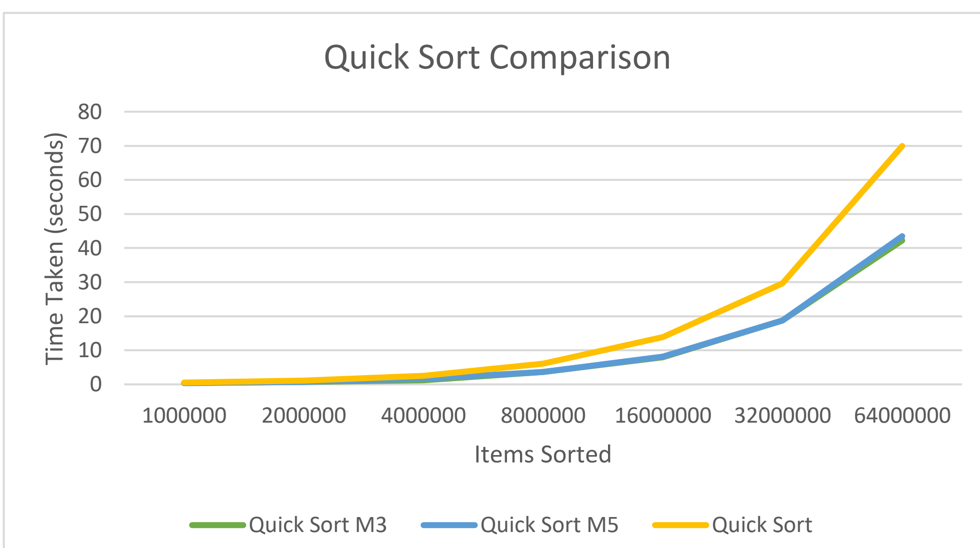
```
Quick Sort Median 3
Items       Time      Ratio
1000000     0.42      0
2000000     0.73      1.7644
4000000     1.23      1.6703
8000000     3.65      2.9731
16000000    7.94      2.1789
32000000    18.74     2.3601
64000000    42.22     2.2523

Quick Sort Median 5
Items       Time      Ratio
1000000     0.35      0
2000000     0.72      2.034
4000000     1.63      2.2716
8000000     3.64      2.2342
16000000    8.15      2.2377
32000000    18.7      2.2936
64000000    43.47     2.3244

Quick Sort
Items       Time      Ratio
1000000     0.53      0
2000000     1.09      2.0546
4000000     2.54      2.33
8000000     6.06      2.3824
16000000    13.83     2.2839
32000000    29.61     2.1412
64000000    69.93     2.3612
```

Surprisingly, the modifications made to the Quick Sort class ended up making it considerably faster. This experiment was repeated several times, and we got similar results each time.

Obviously, the reason for it being faster was because it was able to find a more viable partition each time. Since the Quick Sort provided in the JDK just picks a random index and uses that for the partition, there's a possibility of it being very lopsided each time. As the number of items grows, this becomes an increasing reality.

However, by finding three numbers and getting the median of those, we can partition in a way that's a lot smarter, and therefore save the CPU a lot of work.

We expected the Quick Sort Median 5 to perform better than the Quick Sort Median 3, mainly because it had a better chance of getting the median closer to the actual center. This proved not to be the case, and it seems like the extra time spent trying to find a viable median showed up in the results.



Joey Ferguson - 21720040
Rohail Mall - 21620052
Fin Msiska - 21720014
Jonathan Phiri - 21720042

Team Project 2

27 May 2017

Edge Cases

For a sorting algorithm to truly be efficient and considered "good," it should work in all scenarios. So, we decided to come up with some edge cases that would test most of the weird things that could occur with an array of doubles.

First, we came up with the scenario that the array could have only **two possible values** (1.0 or 2.0). We started testing them with values of 100,000 and 200,000. The Bubble Sort clearly performed the worst in this situation. Selection Sort was also pretty bad, but Insertion Sort performed surprisingly well (considering with the randomly sorted array it was comparable to the Selection Sort times).

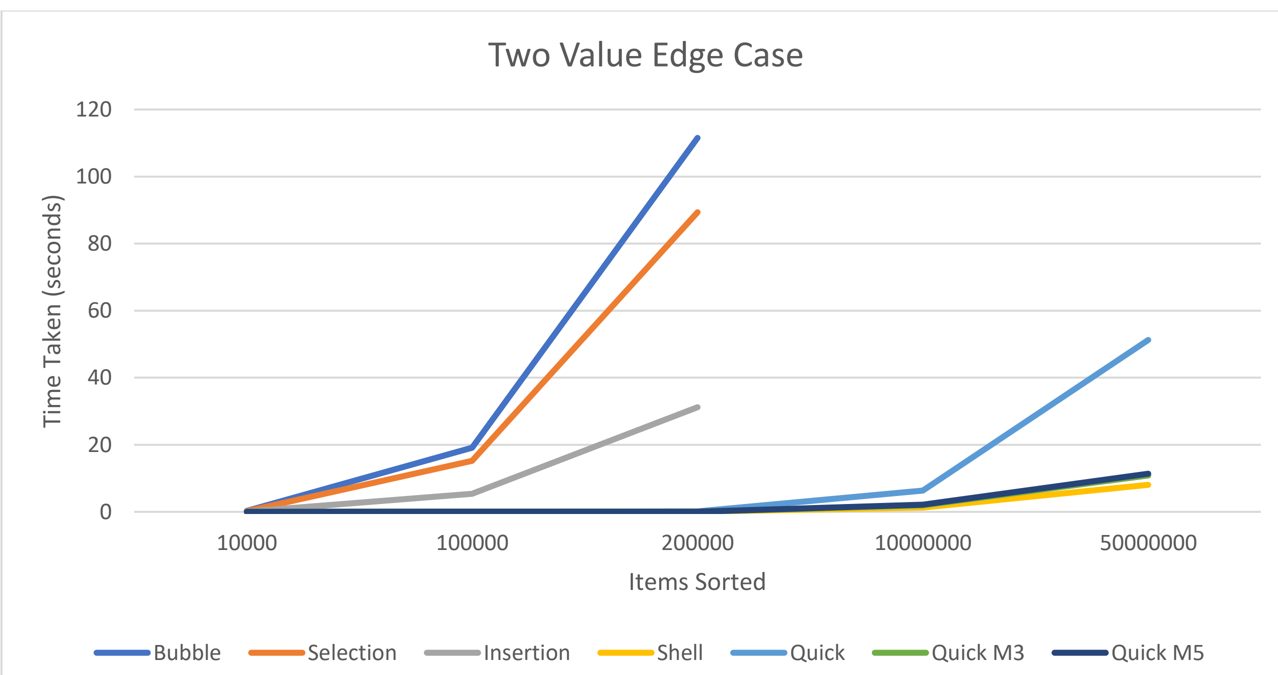
We then moved on to testing 1,000,000 and 5,000,000 to see how the better algorithms would stack up. Clearly, finding a median for the partition worked a lot better than picking a point arbitrarily, especially for the 50 million items.

```
Items to sort: 100000

Bubble Sort
19.081
Selection Sort
15.238
Insertion Sort
5.343
Shell Sort
0.015
Quick Sort
0.04
Quick Sort Median 3
0.032
Quick Sort Median 5
0.043
```

```
Items to sort: 200000

Bubble Sort
111.531
Selection Sort
89.352
Insertion Sort
31.208
Shell Sort
0.031
Quick Sort
0.133
Quick Sort Median 3
0.074
Quick Sort Median 5
0.085
```

```
Items to sort: 10000000

Shell Sort
1.32
Quick Sort
6.364
Quick Sort Median 3
1.96
Quick Sort Median 5
2.14
```

```
Items to sort: 50000000

Shell Sort
8.06
Quick Sort
51.266
Quick Sort Median 3
10.851
Quick Sort Median 5
11.378
```



Joey Ferguson - 21720040
Rohail Mall - 21620052
Fin Msiska - 21720014
Jonathan Phiri - 21720042

We thought it would be interesting to see how the algorithms compared while sorting an array that has been generated in **<u>exact reverse order</u>**.

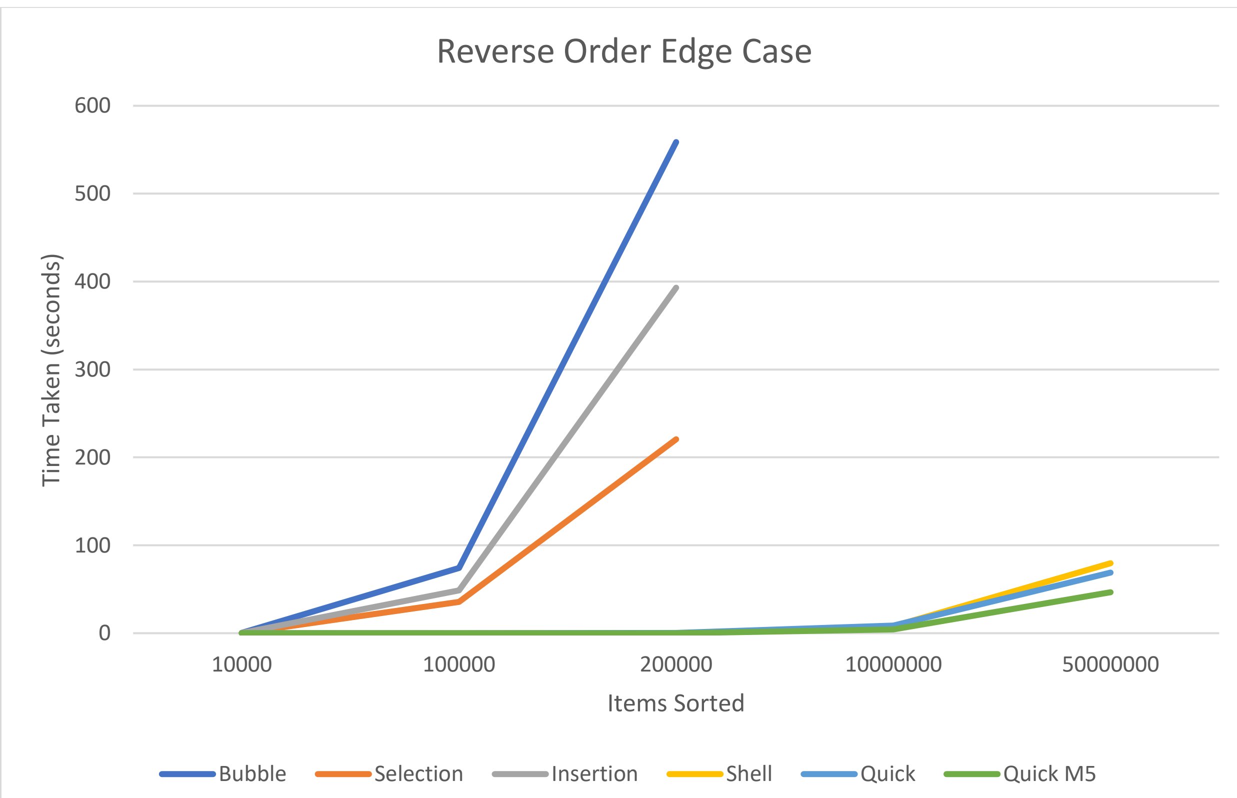
As expected, the Bubble Sort took way longer in this edge case since it is the worst possible case (it takes the smallest at the back and swims it all the way up to the front). It was especially terrible as it grew in number of items. Shell Sort performed extremely well too, since it starts from the outside and can place the items where they should be with minimum comparison. Finally, the Quick Sorts unsurprisingly performed well again – the median finding clearly helps with partitions, even in a worst-case scenario like this one. However, it was interesting to compare how well the quicksort could do in regular scenarios for 50 million items, but it clearly is lacking for the exactly reversed order case.

```
Items to sort: 100000

Bubble Sort
74.113
Selection Sort
35.688
Insertion Sort
48.702
Shell Sort
0.027
Quick Sort
0.071
Quick Sort Median 5
0.067
```

```
Items to sort: 200000

Bubble Sort
558.72
Selection Sort
220.61
Insertion Sort
393.032
Shell Sort
0.077
Quick Sort
0.135
Quick Sort Median 5
0.116
```

```
Items to sort: 10000000

Shell Sort
7.906
Quick Sort
8.575
Quick Sort Median 5
4.347
```

```
Items to sort: 50000000

Shell Sort
79.576
Quick Sort
68.937
Quick Sort Median 5
46.597
```



Fun Facts

We left Bubble Sort running overnight to see how long it would take, and it look 6.25 hours to sort 1,000,000 items – something that our Quick Sorts could do in half a second.

The computer we were using to develop this crashed seven times during this project – perhaps we may have pushed the limit with the number of items stored at instances of IntelliJ we were running.

We learned a ton from this assignment and it was very interesting to us to see the importance of efficiency when it comes to something like sorting.