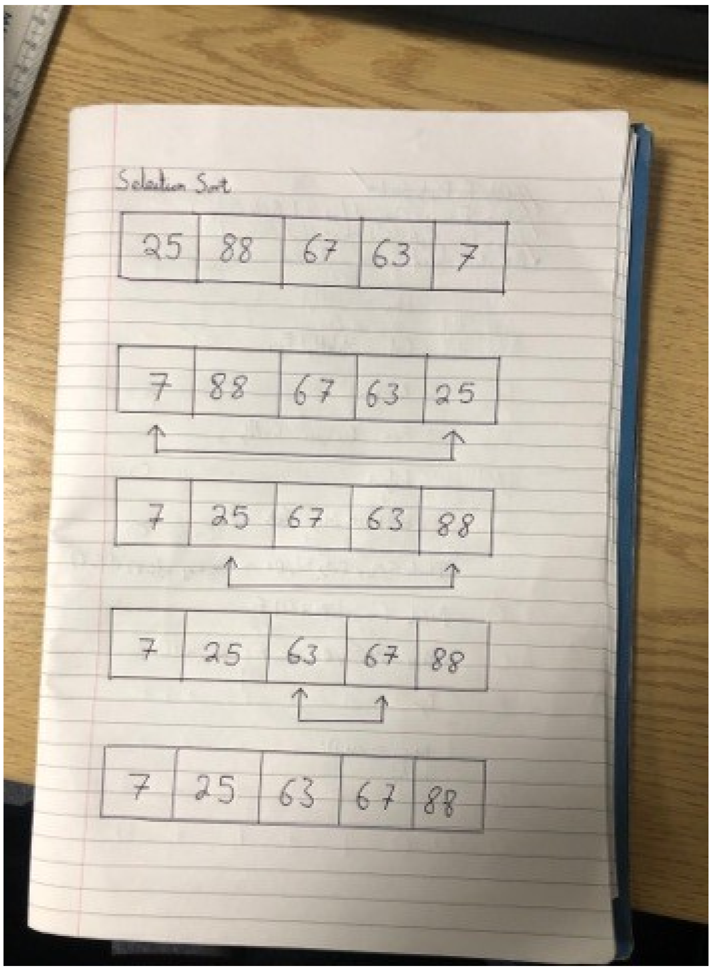
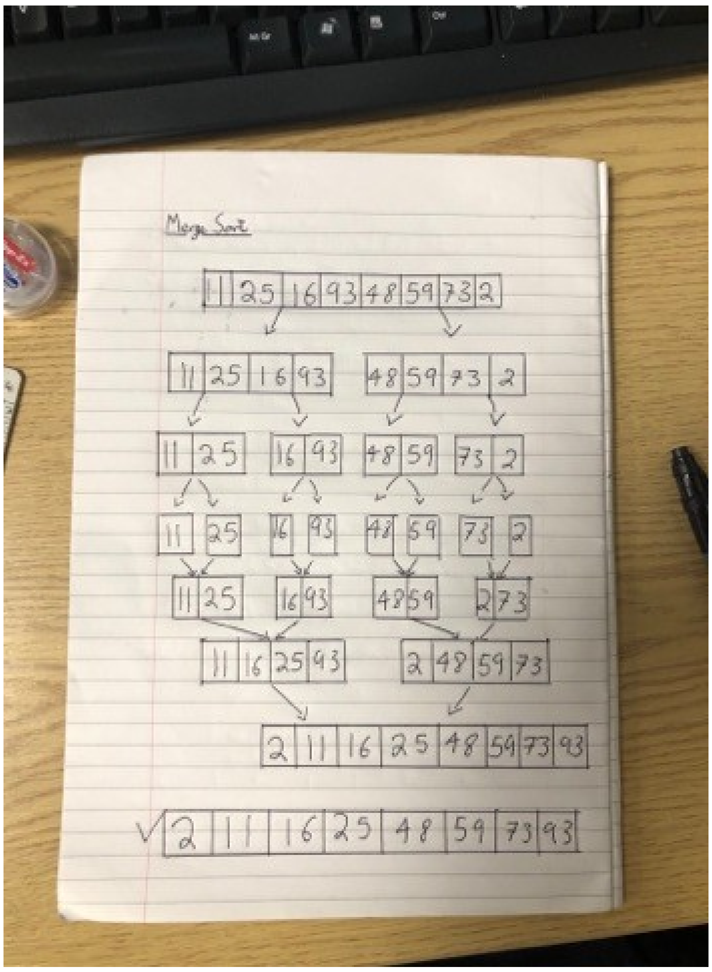
CS2001 W10 Practical **180012514** **19 November 2019**

Design & Implementation

Before starting the implementation, I thought about the two sorts. I drew out how both sorts work to give me a better understanding of how my code should be. The drawings look as follows:





Before designing the sorts, I first wanted to partially create the main class that will be doing everything other than the sorts. I named this class 'Testing' and began with the createArray method as this is the base level of the sorts (I.e you need an array of random numbers for the sorts to work on in the first place). I learned of the Random class which I used to add a random integer as I incremented through the array. I chose to make the maximum number to be sorted 1000 to offer a wide, but not massive, variety of numbers.

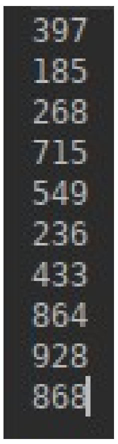
The first of the two sorts that I implemented was the selection sort as it's definitely the easiest to understand and implement. I made use of three long variables; one that stored the time just before the sort begins, one that stores the time just after the sort has been completed and the final once stores the difference between the two – all in nano seconds. My selection sort makes use a nested for loop, the first for loop grabs the first number in the unsorted section of the array, the second for loop works its way through the rest of the unsorted section of the array and if it finds a smaller number, assigns that as the position that will be getting swapped to the end of the sorted section (in its correct place). To do the swap, I make use of a temp value that is used to hold one of the swap values whilst the swap is happening so that it is not lost. At the end of the sort, I record the end time and and calculate the time taken by taking away the start time from the end time. The time taken is then returned to the Testing class.

For the merge sort, I again made use of three long variables that track time. However, this time I require two arrays instead of just one as it is from the original array that the other array is filled with numbers that are in order. As with all merge sorts, my merge sort comes in two parts (plus a third method which instantiates values and calls the first part before returning the timeTaken): divide & conquer. In the divide method, I make use of recursion to split the array into two halves. These halves are continually broken down until they can't be any further. Then the conquer method builds up the array again. In the conquer method, two halves of the array have their first values compared, whatever value is smallest gets put into the sorted array and that half looks at the next number in it. This process is repeated until one half has been fully put into the array. In this case, the remainder of the leftover half is put into the array. This produces a sorted array. Since the first array originally consists of the first half then the second half, if the first half is the first to be fully put back into the array, then nothing else is needed as the remainder of the second half is already in place. However, if the second half is the first to be fully put into the array, then I needed another while loop to put whatever is left in the first half back into the array, at the end.

Once I had my sorts implemented, I moved onto finishing the implementation of the Testing method. For this method, I realised I would be using three key numbers; the maximum array size I would be using, by how many sizes I increment by and how many times I test each array size before I get the median. I decided it would be better to calculate the mean time for each array size rather than the average time as by sorting and picking the mean (middle value), I remove extreme values for times that are outliers from the standard. This gives a more accurate graph. To store the times before they are output to the file, I use arrayLists as they are of no fixed size so I can chose however many array sized I'd like on the fly. I decided to use a nested for loops for running the sorts; the first for loop would run for until the maximum size chosen by me is reached and the second for loop would run until the number required for the median is reached. In the second loop, I simply call findSortMedian for both sorts. I differentiate between the two sorts with a boolean with merge sort being true. When calling findSortMedian, I pass into it the differentiating boolean, as well as the time taken for the sort which is called. I use an if to check which sort is being focused on, before adding the time to one of another pair of arrayLists. Once the size of the arrayList equals that of the size required to calculate the median, I make use of the collections method sort() on the arrayList so that the median value is in fact the middle time. The median is then added to the appropriate sort's time arrayList before the smaller arrayList is cleared for the next array size. Once a median is acquired for each array size, the writeToFile method is called. In said method I make use of a file writer that writes the times for both sorts at a specific array size.

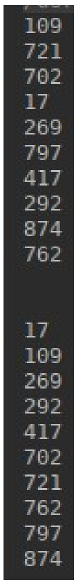
Testing

The first thing I tested was that I could produce an array composed of randomly generated numbers. For this (and for all other first tests) I kept the array to a size of 10 and once the integers were assigned to their position in the array, I printed them out. When I saw a list of 10 completely unrelated numbers within the range I set (0-1000) I knew I had successfully implemented the random aspect of this practical and could move on.



```
397
185
268
715
549
236
433
864
928
868
```

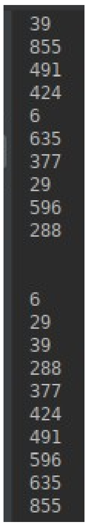
I tested my selection sort by adding a for loop that works through the sorted array, printing out each value. When each number was in ascending order, I knew I had sorted the array and could move onto the merge sort:



```
109
721
702
17
269
797
417
292
874
762


17
109
269
292
417
702
721
762
797
874
```

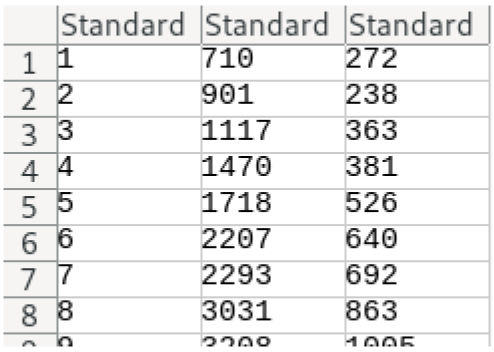
I tested my merge sort by adding a for loop that went through the sorted array just before the time was returned. When each number was in ascending order, I knew I had sorted the array and could move onto the merge sort:



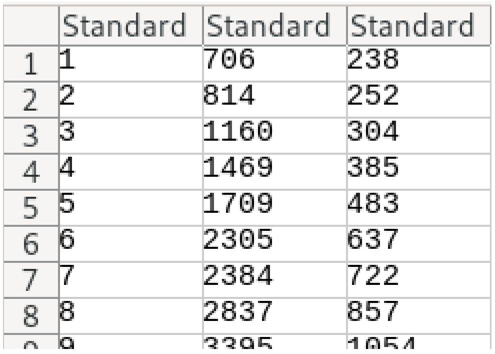
```
39
855
491
424
6
635
377
29
596
288


6
29
39
288
377
424
491
596
635
855
```

I tested to see if my file writer worked by running the program and checking to see if the new ‘data.csv’ file had data in it, once I saw the two sets of time, I knew it was working. I then tested it again to see if the new data overwrote the old data instead of just appending to it:



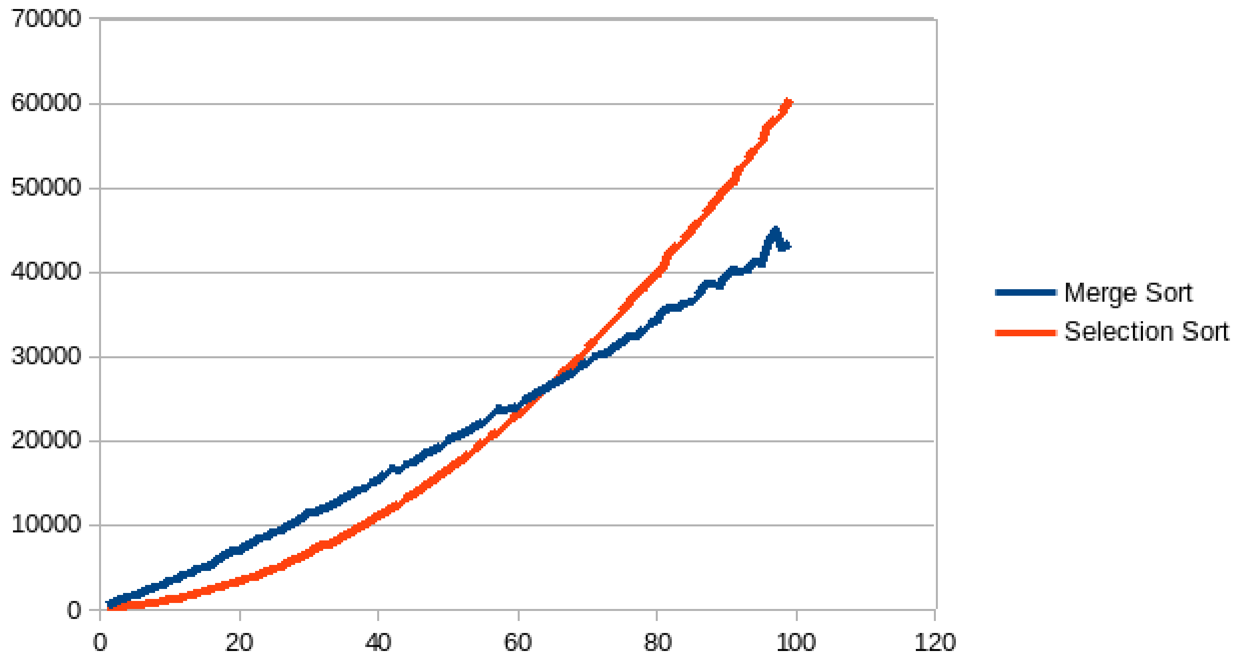
	Standard	Standard	Standard
1	1	710	272
2	2	901	238
3	3	1117	363
4	4	1470	381
5	5	1718	526
6	6	2207	640
7	7	2293	692
8	8	3031	863



	Standard	Standard	Standard
1	1	706	238
2	2	814	252
3	3	1160	304
4	4	1469	385
5	5	1709	483
6	6	2305	637
7	7	2384	722
8	8	2837	857

Evaluation

When setting up the constant variables to loop until an array list of size 100, the following graph is produced:

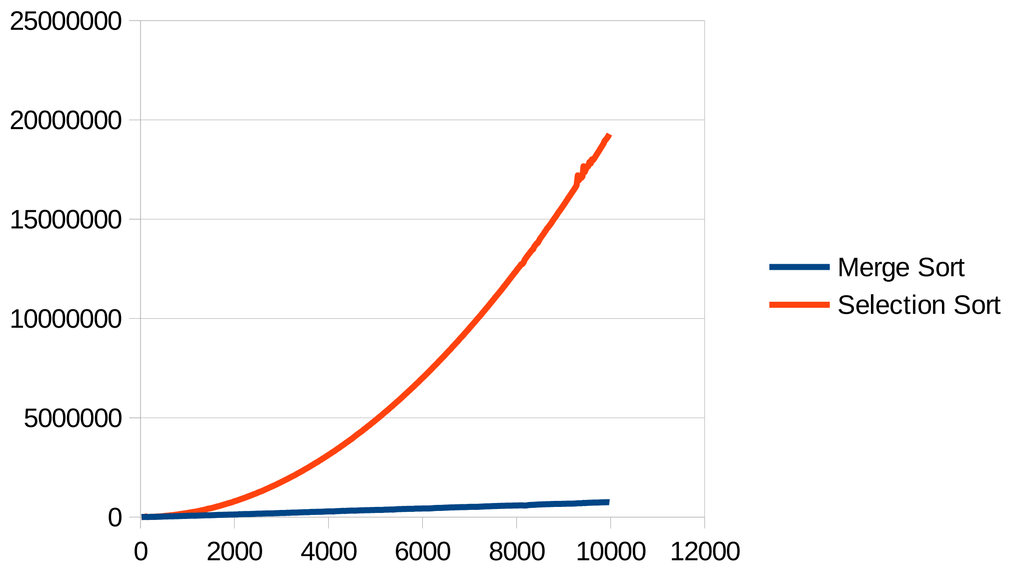


This shows a crossover point between 60 and 70. Upon multiple repeats, I got the same exact crossover so I am pretty confident with this result.

When running the program I used the commands -Xint -Djava.compiler=NONE in IntelliJ's VM options as without them, once a certain number of operations are repeated a number of times, byte code is converted to machine code to run faster, but it produces strange spikes at specific values.

Conclusion

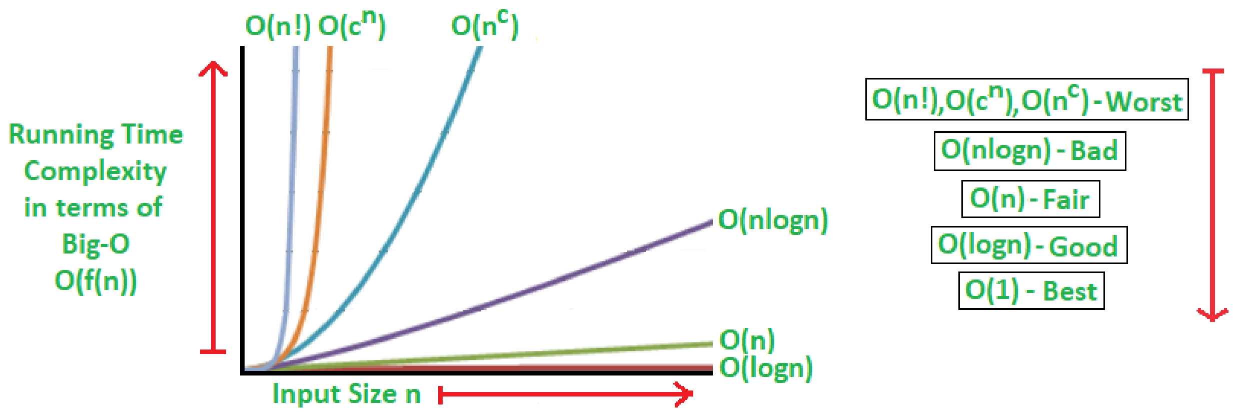
When setting up the constant variables to loop until an array list of size 10000 is reached, then the graph below is produced.



This combined with the graph at a smaller scale shows that my results can be trusted as the selection sort starts out as the quicker sort, but as the scale increases the merge sort becomes the faster sort, agreeing with the conclusion that:

‘selection sort, whose time complexity is O(n2), is faster than merge sort, which has complexity O(n log n). At some point, however, for some n, the times taken will “cross over”, making merge sort faster. ’.

The shapes of the two lines also match with the complexities stated in the conclusion, further proving my result’s liability. The complete complexity graph can be seen here (courtesty of https://www.geeksforgeeks.org/analysis-algorithms-big-o-analysis/):



References

Help with filling the array with random numbers-
https://docs.oracle.com/javase/8/docs/api/java/util/Random.html

Help with making use of Collections to get the real median time-
https://docs.oracle.com/javase/8/docs/api/?java/util/Collections.html