

# Assignment 2

Archer

---

Romano Asciutto (500801794)

Jesse van Bree (500801418)

Class IS201

9th October, 2019

# Table of contents

<b>Table of contents</b>	<b>1</b>
<b>1 Introduction</b>	<b>2</b>
<b>2 Development</b>	<b>3</b>
2.1 Classes	3
2.1.1 Archer	3
2.1.2 ArcherComparator	3
2.1.3 ArcherLauncher	3
2.1.4 ChampionSelector	4
2.1.5 Names	4
2.1.6 SortingMeasurement	4
2.2 Most important code	5
2.2.1 Comparator compare	5
2.2.2 Measure sorting time	6
2.2.3 Generate archers	6
2.2.3 Weighed score	7
2.3 Efficiency	8
2.3.1 Insertion Sort	9
2.3.2 QuickSort	12
2.3.3 Collections Sort	16
2.4 Output	19
2.4.1 SortingMeasurement	19
2.4.2 Testing	20
2.4.2.1 ExtendedArcherTest	20
2.4.2.1 ExtendedChampionSelectorTest	20
<b>3 Conclusion</b>	<b>21</b>
<b>Sources</b>	<b>22</b>

# 1 Introduction

In this assignment, we have built an application for an archery competition. This program can generate a list of archers and their scores, after which their total (weighed) score can be calculated. This way, the application can determine a winner by sorting the scores from high to low. For this we have used three different sorting algorithms and determined their efficiency.

The purpose of this report is to go over our archery competition assignment and explain how we implemented the required functionalities. The main goal is to explain how we implemented the sorting algorithms and determined their efficiency. We will do so using graphs and mathematical constructs.

We will begin by describing the purpose of each class and how they work together. This will give an overall view of the structure of our application. With this discussed, we will go over the most important code that we wrote and explain how it works along with code snippets. This will mostly consist of sorting algorithms along with their efficiency, for which we measured the time.

At the end of this report, we will conclude it by reflecting on our solution and determine what sorting algorithm is the most efficient in the case of this assignment along with a graphical representation.

## 2 Development

In this chapter we will discuss the different parts of the development of the project. This includes the different classes, the most important code of the program, the efficiency of the three algorithms Insert, Quick and Collections sort and lastly the output of the program and tests.

### 2.1 Classes

In this section we will discuss the different classes that we have used in this assignment and describe how they work.

#### 2.1.1 Archer

The archer class is our central class which contains the logic to create new archer instances using a so-called factory method. This is a static method with an integer parameter indicating the amount of archers to create. After calling this method, it will generate the archers in a List and determine their score by simulating a competition. This class also implements Comparable so the total score of an archer can be compared to another which is used to sort them and to determine a winner.

#### 2.1.2 ArcherComparator

The archer class has a compare method in which it creates a new instance of this class. This class implements the Comparator interface and overrides the compare method which has custom logic to determine a winner among 2 archers. The logic is as follows: if the total scores are equal, compare the weighed score. If the weighed scores are also equal compare the ID's. As lower experienced archers have a higher ID, the one with the higher ID would win. This way the comparator always returns either -1 or 1, so there will always be a winner among the 2 archers.

#### 2.1.3 ArcherLauncher

This is a simple convenient test class containing a List with a number of generated archers by executing the static factory method of the archer class. As we progressed in this project, we have used this class to test our sorting algorithms and output the result in the console using the archers toString method. This way we can see whether we get the expected result or not.

### 2.1.4 ChampionSelector

The ChampionSelector class contains three public methods which implement the sorting algorithms: Insertion, Quick and Collections.sort (which uses merge sort as algorithm). In this class there are some private helper methods for quick sort. These helper methods are: Partition, swap and an overloaded method for quickSort. Partition is used in Quicksort to split up the list, swap is used to swap items in a list **and finally the overloaded quicksort method is used recursively because quicksort needs to calculate a pivot point based on high and low values.**

### 2.1.5 Names

The Names class is a small helper class which returns a random first name or surname. There are two arrays: the first array contains a bunch of first names, where the second is filled with surnames. Then there are two methods which return a random first name or surname based on a randomizer. This class is used when generating random archers in the Archer class.

### 2.1.6 SortingMeasurement

The SortingMeasurement [1] class contains different methods. Which either test or calculate parts of the efficiency of the three sorting methods: sellnsSort, quickSort and collectionSort of the ChampionSelector class. The main method starts the test and repeats itself x amount of times based on a variable TIMES\_TO\_RUN. During each run the 3 methods mentioned above will be tested. At the start of the test a List of archers is generated. This list of archers starts at a 100 and will be sorted. The time taken will be recorded in an array. The array of archers will double in size and be sorted again. This will repeat until the size of the list exceeds 5000000 archers or the time taken exceeds the limit of 20 seconds. After the main method is done with testing each sorting method and repeating this x amount of times. The results will be calculated and printed in the console, by the printResult method. With the help of private helper methods to calculate the average and total.

## 2.2 Most important code

In this chapter we will provide code snippets of the most important parts of our assignment along with an explanation of why these are important and how they work.

### 2.2.1 Comparator compare

The compare method is important, because it is used in all the sorting methods. This method compares 2 archers based on their total score, weighed score or their id. The method works as follows: The method first checks if the total scores are equal, if so than it looks if the weighed score are equal, if that is also true the method will compare the id.

```
@Override
/**
 * Compares 2 archers on total score, Weighed score or id.
 * Returns -1 if archer 2 is ahead, 0 if they are equal, 1 if archer 1 is ahead
 */
public int compare(Archer o1, Archer o2) {
    if(o1.getTotalScore() == o2.getTotalScore()){
        if(o1.getWeighedScore() == o2.getWeighedScore()){
            return Integer.compare(o1.getId(), o2.getId());
        } else {
            return Integer.compare(o1.getWeighedScore(), o2.getWeighedScore());
        }
    } else {
        return Integer.compare(o1.getTotalScore(), o2.getTotalScore());
    }
}
```

## 2.2.2 Measure sorting time

To measure the efficiency of an algorithm you need to measure the time it took for the algorithm to finish. To calculate this, we wrote the following method. Firstly it will get the current nanotime from the system. Then it calls the sort method to sort the list. When the sort is finished, the time is calculated by subtracting the current system nanotime with the starttime. The difference is the time it took in nanoseconds and is recorded into an array for later use in other methods.

```
private static void measureInsSortingTime(List<Archer> archers, int index) {
    long startTime;
    startTime = System.nanoTime();
    ChampionSelector.selInsSort(archers, archerComparator);
    sortTimes[timeRun][0][index] = (double) (System.nanoTime() - startTime);
}
```

## 2.2.3 Generate archers

The generate archers method is helpful to quickly generate a list of archers. Especially when testing sorting algorithms. You can quickly increase the list of archers by just calling a method with the number of archers you need. The method uses the names class to get a random firstname and a surname. and after this it will call the letArcherShoot method to generate some values. Which are you used to generate a score.

```
/**
 * This methods creates a List of archers.
 * @param nrOfArchers the number of archers in the list.
 * @return
 */
public static List<Archer> generateArchers(int nrOfArchers) {
    List<Archer> archers = new ArrayList<>(nrOfArchers);
    for (int i = 0; i < nrOfArchers; i++) {
        Archer archer = new Archer(Names.nextFirstName(), Names.nextSurname());
        letArcherShoot(archer, nrOfArchers % 100 == 0);
        archers.add(archer);
    }
    return archers;
}
```

### 2.2.3 Weighed score

The weighed score is one of the variables which can be used to sort the list of archers. This is also used as a tiebreaker in the compare method if the total score is the same. It generates points based on the following rules: There are 0 to 10 points on the board. 1 to 10 is worth its value + 1 and 0 subtract 7 from the total score. Example: 10 is hit twice, 4 is hit once and 0 is hit once. The weighed score will be  $((10+1)*2) + ((4+1)*1) - (1*7) = 22 + 5 - 7 = 20$ .

```
public int getWeighedScore() {  
    int weighedTotal = 0;  
    for (int i = scores.length - 1; i > 0; i--) {  
        weighedTotal += (i + 1) * scores[i];  
    }  
    return weighedTotal - scores[0] * 7;  
}
```



## 2.3 Efficiency

In this chapter we will be going over the three sorting algorithms that we have implemented in our application to sort the archers. We have used the following three algorithms:

- Insertion sort [2]
- QuickSort [3]
- Collections sort [4]

We will explain our implementation of each algorithm and graphically display their efficiency so we can compare them in our conclusion to find out what the most efficient algorithm is during our tests. For this, we have measured the time it took to sort the data, increasing the amount of data by 2 with every iteration. We have done this for a total of 10 runs for each algorithm so we can get the average time of all runs to improve the accuracy.

As we tested the efficiency of our algorithms, we found out that Java uses the Just In Time (JIT) compiler by default. This improves the performance of Java applications at runtime which is normally preferred, but in the case of our time measurement it would give inconsistent results. By using the `-Xint` parameter in the JVM settings of our IDE, we have disabled this feature so we get consistent results.

### 2.3.1 Insertion Sort

The first sorting algorithm that we implemented is the insertion sort. This is a simple sorting algorithm that goes through an unsorted array, takes each item it finds and puts it in the right place, in our case from high to low. This algorithm is simple to code and works fine on small sets of data, but as we found out, its efficiency quickly degrades once the amount of objects to sort increases. We will explain this in the next section.

Let's first look at the code. We loop through the archer List and get the previous index of the current archer in the iteration. We then loop backwards through the List and check if the previous archer has a lower score than the current one. If so it switches that archer with the one in front of it and lowers the index. This keeps going until the beginning of the List has been reached. After this, it moves on to the next archer and so on. This way the whole List gets sorted from high to low.

```
public static List<Archer> selInsSort(List<Archer> archers, Comparator<Archer>
scoringScheme) {
    // Loop to all archers in the List
    for (Archer archer: archers) {
        int previousArcherIndex = archers.indexOf(archer) - 1;

        // Loop backwards through the List and check if previousArcher has a higher
score
        // If the previousArcher score is lower than set the previousArcherIndex
subtracted by 1 and try again until
        // the previousArcher has a higher score then switch the position of archer
and previous archer
        while (previousArcherIndex >= 0 &&
scoringScheme.compare(archers.get(previousArcherIndex), archer) < 0) {
            archers.set(previousArcherIndex + 1, archers.get(previousArcherIndex));
            previousArcherIndex = previousArcherIndex - 1;
        }
        archers.set(previousArcherIndex + 1, archer);
    }

    // Return the sorted List
    return archers;
}
```

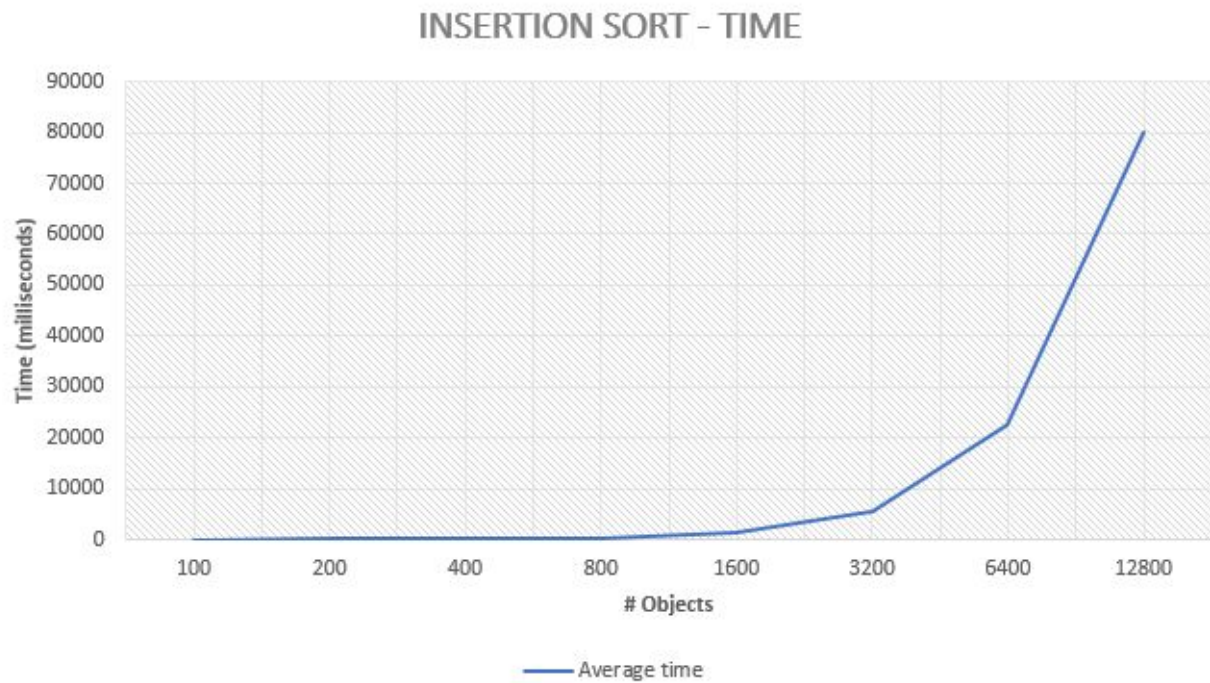
Below we have measured the time for the sorts over 10 runs with doubling object counts for each iteration, until it takes longer than 20 seconds to sort. As we can observe, only 2 runs made it past 6400 objects, taking about 80 seconds each to sort 12800 objects. What can be clearly concluded from this, is that this algorithm is very inefficient with a high object count. With the worst and average case efficiency being  $O(n^2)$ , the time to sort in most cases exponentially increases.

# Objects	Time (milliseconds)										Average
	Run 1	Run 2	Run 3	Run 4	Run 5	Run 6	Run 7	Run 8	Run 9	Run 10	
100	7.87	6.56	8.89	6.06	5.36	5.47	4.78	5.16	4.7	4.65	5.95
200	26.17	24.32	33.49	24.64	21.25	21.77	18.3	17.77	20.14	19.17	22.7
400	111.21	100.25	123.8	86.39	80.46	89.91	78.23	72.06	73.3	79.8	89.54
800	533.65	399.19	591.22	331.16	309.3	303.38	317.77	302.21	307.8	306.65	370.23
1600	1760.29	1578.67	2159.4	1471.14	1200.17	1210.03	1195.34	1251.67	1234.53	1181.19	1424.24
3200	6673.58	6666.85	7504.33	5721.06	4994.64	4862.07	4896.78	5167.58	4936.35	4798.37	5622.16
6400	26163.39	26555.56	29368.58	22417.28	20615.38	19500.91	19742.03	20210.33	20266.6	20120.59	22496.07
12800	0	0	0	0	0	80272.72	79542.78	0	0	0	79907.75

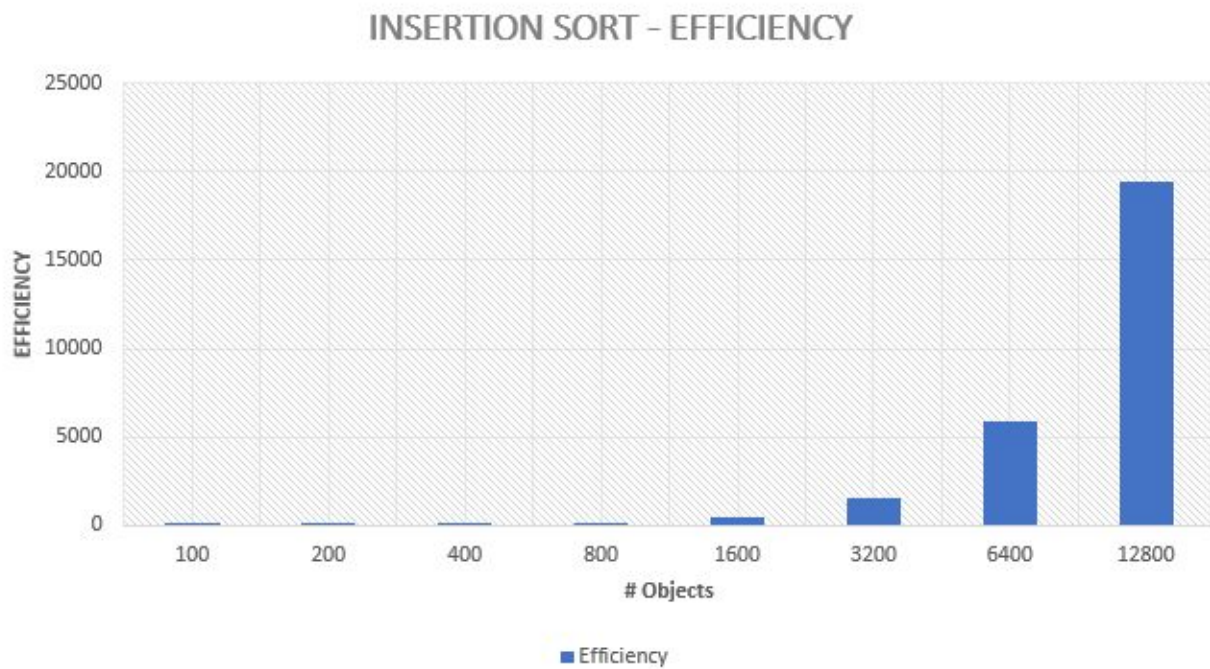
Table 1 - Run time in milliseconds (insertion sort)

# Objects	Average time (milliseconds)	Efficiency $T/\log(n)$
100	5.95	2.98
200	22.7	9.87
400	89.54	34.41
800	370.23	127.53
1600	1424.24	444.5
3200	5622.16	1603.97
6400	22496.07	5910.41
12800	79907.75	19455.48

Table 2 - Time efficiency (insertion sort)



Graph 1 - Average run time for number of objects



Graph 2 - Efficiency for number of objects

## 2.3.2 QuickSort

This is the second algorithm we wrote to sort the archers. This algorithm is a divide and conquer algorithm. It sorts the Archer list by calling itself recursively and calls the partition method to divide the list into smaller lists. The dividing point also known as the pivot and is in our case always the last element in the list. Because the algorithm is logarithmic, the list needs to be sorted. this happens when the quicksort gets called the first time. Based on the value of the pivot the elements get split up in 2 parts. First the part that has lower values than the pivot and the second part which has values higher than the pivot. this gets repeated until the array is sorted.

This method calls the private helper method quickSort which starts a recursive quicksort algorithm.

```
public static List<Archer> quickSort(List<Archer> archers, Comparator<Archer>
scoringScheme) {
    quickSort(archers, scoringScheme, 0, archers.size() - 1);
    return archers;
}
```

This is the method that actually does the quicksorting. It calls itself recursively until the value of low is higher than high.

```
private static void quickSort(List<Archer> archers, Comparator<Archer>
scoringScheme, int low, int high) {
    // Check if there are items left to sort
    if (low < high) {
        // Execute the partitioning code and store the index of the pivot
        int partitioningIndex = partition(archers, scoringScheme, low, high);

        // Sort left and right side of the pivot
        quickSort(archers, scoringScheme, low, partitioningIndex - 1);
        quickSort(archers, scoringScheme, partitioningIndex + 1, high);
    }
}
```

The partition method takes the last element of the list as the pivot. It checks each element and swaps it before the pivot if the score is higher than the pivot. This way, the partition being sorted will be sorted from high to low.

```
private static int partition(List<Archer> archers, Comparator<Archer> scoringScheme,
int low, int high) {
    // Take the last archer as the pivot
    Archer pivot = archers.get(high);
    int i = low - 1;

    // Loop from beginning of the partition to the end
    for (int j = low; j < high; j++) {
        // If the archer left of the pivot has a higher score than the right, swap
        them
        if (scoringScheme.compare(archers.get(j), pivot) > 0) {
            i++;
            swap(archers, i, j);
        }
    }

    swap(archers, i + 1, high);
    return i + 1;
}
```

This method swaps the two archers in the list archers

```
/**
 * @param archers List of archers
 * @param firstIndex index of the first item
 * @param secondIndex index of the second item
 */
private static void swap(List<Archer> archers, int firstIndex, int secondIndex) {
    Archer temp = archers.get(firstIndex);
    archers.set(firstIndex, archers.get(secondIndex));
    archers.set(secondIndex, temp);
}
```

The efficiency of quick sort is different for the best and worst cases. The average and best case scenarios take  $O(n \log(n))$  amount of time. But for the worst case scenario the time taken will be  $O(n^2)$  this happens when the array is sorted perfectly in the reverse direction. To solve this the algorithm shuffles the list beforehand which takes  $O(n)$  amount of time. Which means this gets added to the original time complexity of  $O(\log(n))$ . Which is also why they can guarantee the average sort takes  $O(n \log(n))$  time, because the likelihood of a shuffle being the worst case scenario is almost nonexistent.

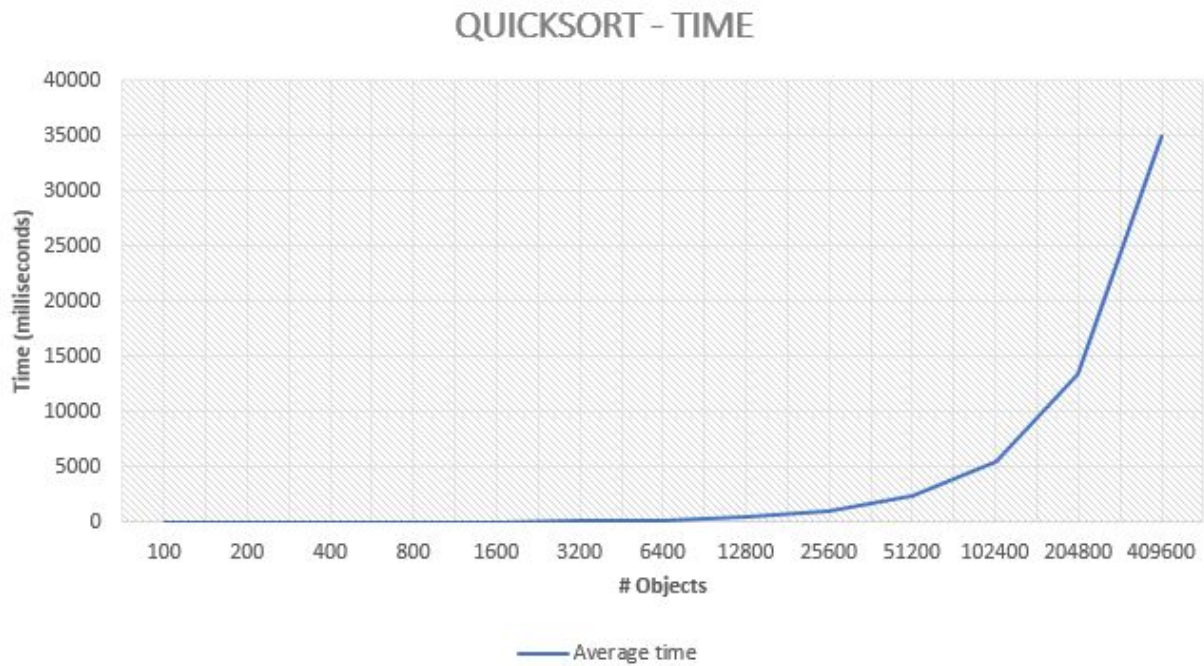


# Objects	Time (milliseconds)										Average
	Run 1	Run 2	Run 3	Run 4	Run 5	Run 6	Run 7	Run 8	Run 9	Run 10	
100	1.81	1.68	1.68	1.68	1.14	1.29	1.35	1.19	1.24	1.4	1.45
200	3.74	4.02	4.51	3.63	3.44	3.27	3.22	3.18	2.86	3.37	3.52
400	9.21	9.75	11.36	10.53	7.05	9.42	6.89	6.64	6.43	6.91	8.42
800	23.52	24.52	22.7	21.64	17.32	16.81	19.23	17.91	16.68	18.61	19.89
1600	47.53	60.8	64.66	45.82	36.66	40.33	38.76	36.66	38.76	35.72	44.57
3200	109.08	113.42	126.47	111.54	83.88	120.96	87.81	83.06	83.55	104.74	102.45
6400	259.36	240.64	298.58	217.52	175.67	217.95	187	174.26	196.21	191.84	215.9
12800	572.5	522.94	626.86	579.66	521.35	566.04	418.3	428.86	422.7	405.67	506.49
25600	1387.95	1257.91	1356.09	1130.74	1015.62	1085.1	938.58	1002.83	946.31	899.7	1102.08
51200	2950.72	2841	2924.46	2176.52	2386.78	2291.72	2273.39	2554.23	2110.84	2064.09	2457.38
102400	6486.27	7064.26	5299.69	5100.49	5323.77	5459.12	5238.95	5373.57	4899.12	4802.82	5504.81
204800	15509.04	17975.43	12312.24	13903.05	12649.06	13097.27	11640.49	12352	12538.71	11760.28	13373.76
409600	44969.02	53096.9	33929.9	36267.94	30544.65	28560.1	32934.21	30839.13	31141.76	28011.32	35029.49

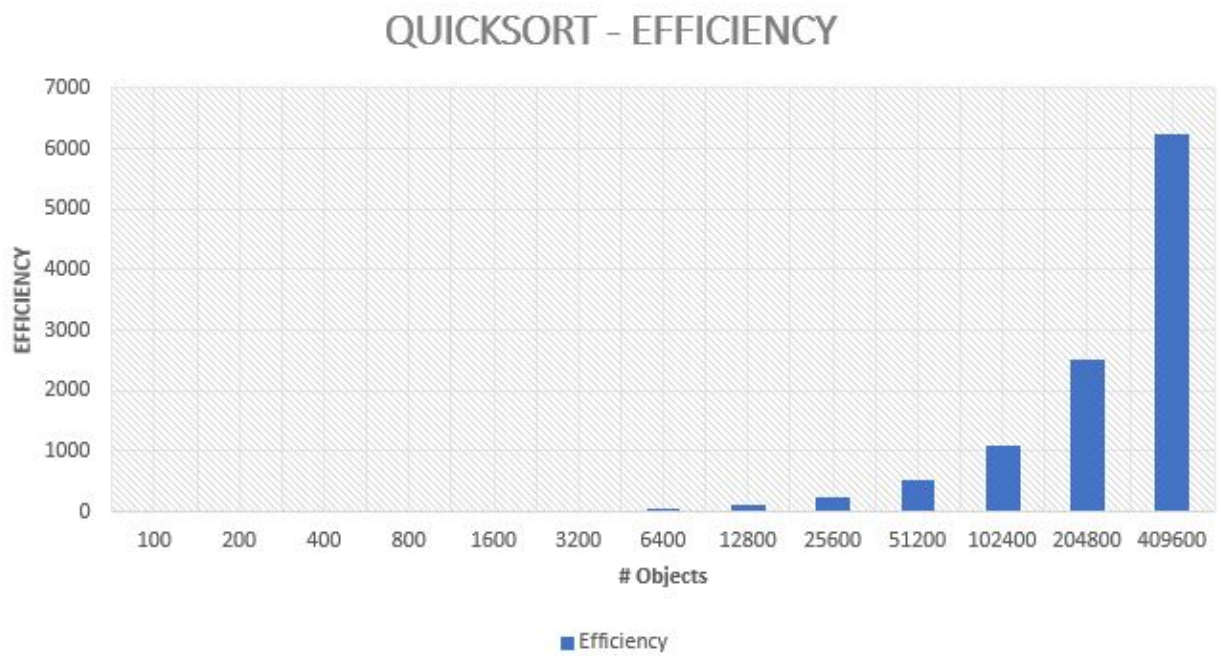
Table 1 - Run time in milliseconds (quicksort)

# Objects	Average time	Efficiency $T/\log(n)$
100	1.45	0.73
200	3.52	1.53
400	8.42	3.24
800	19.89	6.85
1600	44.57	13.91
3200	102.45	29.23
6400	215.9	56.72
12800	506.49	123.32
25600	1102.08	250
51200	2457.38	521.82
102400	5504.81	1098.7
204800	13373.76	2517.97
409600	35029.49	6241.49

Table 2 - Time efficiency (quicksort)



Graph 1 - Average run time for number of objects



Graph 2 - Efficiency for number of objects



### 2.3.3 Collections Sort

The third and last algorithm we implemented is collections sort. The class Collections is provided by java JDK. Which contains methods like shuffle and sort, which the latter one we use in this method. Collections.sort uses a modified version of the merge algorithm to sort a list. Because the standard implementation sorts the list from low to high. We need an extra parameter to reverse the sort and sort from high to low. we also provide a comparator we wrote to get the archer with the highest points (See chapter 2.2.1 for more info).

```
/**
 * This method uses the Java collections sort algorithm for sorting the archers.
 */
public static List<Archer> collectionSort(List<Archer> archers, Comparator<Archer>
scoringScheme) {
    // Sort the archers List backwards because the highest score needs to be the
    first value
    Collections.sort(archers, Collections.reverseOrder(scoringScheme));
    return archers;
}
```

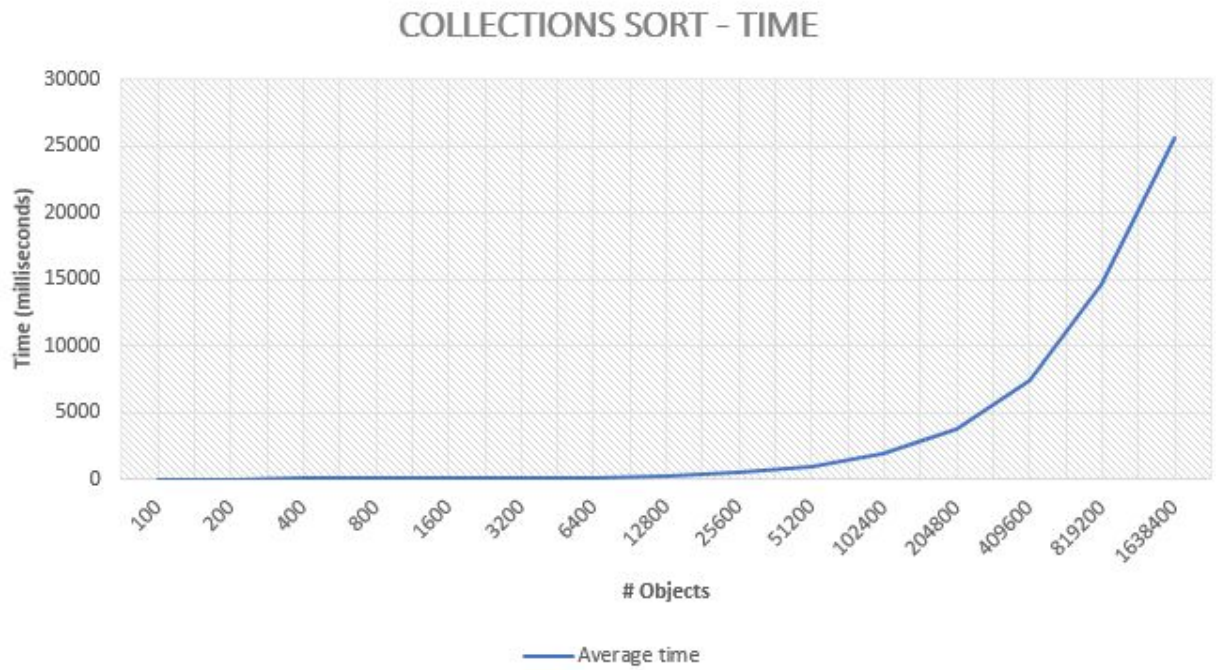
On the next pages we have a graphical representation of the measurements of the collections sort method. As described in paragraph 2.3.1 we run each test 10 times with increasing object count. As you can see this method of sorting sorts scales really well with lower and increasing object sizes. The downside is that there is some overhead with using merge as an algorithm. This includes creating multiple temporary arrays while sorting and as a result will use more memory. But considering the above the speed is still remarkably fast. The efficiency of this algorithm is  $O(n \log n)$  and not  $O(\log n)$  because it divides the array in sub chunks until there is only one element left. After it merges them together which takes  $O(n)$  amount of times. which adds to  $O(n \log n)$ .

# Objects	Time (milliseconds)										Average
	Run 1	Run 2	Run 3	Run 4	Run 5	Run 6	Run 7	Run 8	Run 9	Run 10	
100	1.45	1.03	0.63	0.84	0.62	0.59	0.61	0.6	0.94	0.98	0.83
200	1.9	2.24	1.41	1.48	1.86	1.51	1.63	1.41	2.14	2.28	1.79
400	4.44	5.66	3.51	4.28	5.41	3.42	3.68	3.31	3.52	4.67	4.19
800	10.4	11.65	8.21	10.55	7.78	8.45	8.52	7.57	8.22	10.18	9.15
1600	22.69	27.69	17.29	24.87	17.48	17.88	22.95	17.19	18.71	18.59	20.53
3200	50.51	63.27	39.74	41.11	40.61	50.55	44.8	38.26	39.91	41.84	45.06
6400	111.12	136.34	88.77	87.18	87.07	85.04	89.02	88.88	85.19	86.64	94.52
12800	234.48	312.61	224.75	248.74	178.8	198.23	218.85	196.16	192.42	233.95	223.9
25600	485.53	597.74	466.79	449.48	421.34	371.57	397.18	423.69	408.24	435.07	445.66
51200	1091.29	1245.39	922.18	813.13	871.51	784.58	875.6	875.98	814.47	753.04	904.72
102400	2162.39	2487.3	1909.26	1857.34	1663.69	1578.44	1745.82	1766.21	1576.17	1734.07	1848.07
204800	4449.43	5272.38	3415.8	3778.28	3522.93	3320.03	3538.72	3274.16	3194.07	3594.08	3735.99
409600	8933.52	10044.77	6735.62	7416.72	6872.4	6662.37	6865.08	6816.15	6528.45	6821.58	7369.66
819200	17953.12	20021.35	12886.32	13477.98	13306.53	13792.98	14195.87	13368.95	12850.32	13703.12	14555.65
1638400	36444.22	0	28596.58	27817.76	26930.19	27555.21	28965.63	26472.12	25557.86	28037.34	25637.69

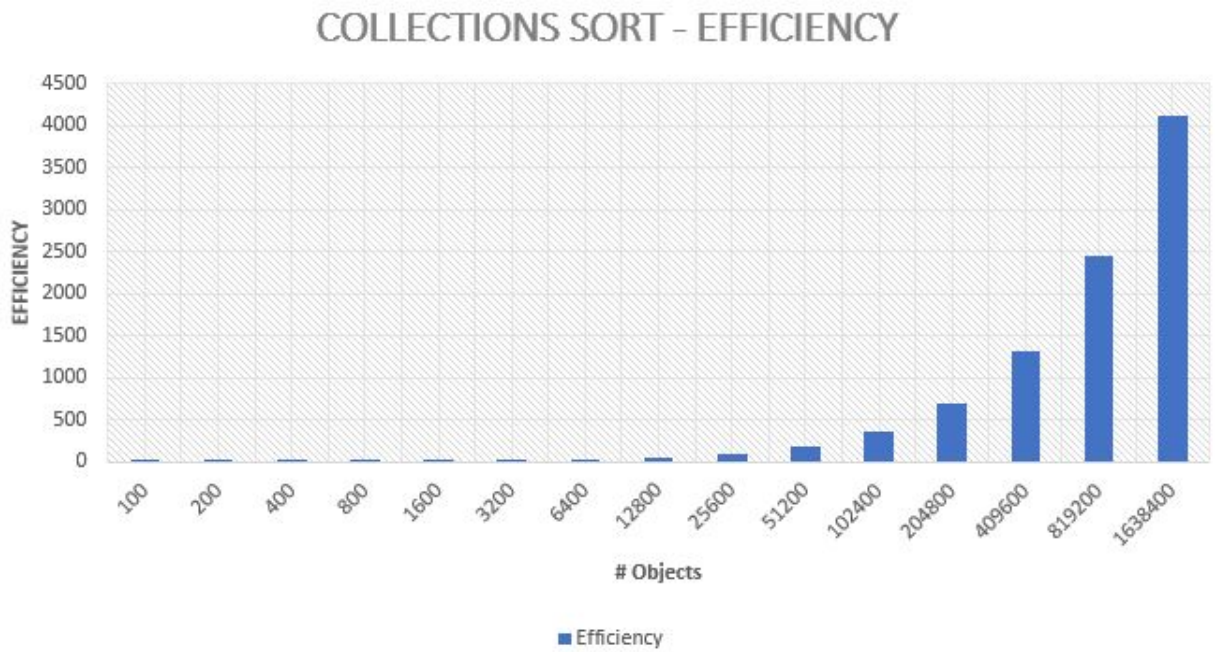
Table 1 - Run time in milliseconds (collections sort)

# Objects	Average time	Efficiency $T/\log(n)$
100	0.83	0.42
200	1.79	0.78
400	4.19	1.61
800	9.15	3.15
1600	20.53	6.41
3200	45.06	12.86
6400	94.52	24.83
12800	223.9	54.51
25600	445.66	101.1
51200	904.72	192.11
102400	1848.07	368.85
204800	3735.99	703.4
409600	7369.66	1313.11
819200	14555.65	2461.47
1638400	25637.69	4125.52

Table 2 - Time efficiency (collections sort)



Graph 1 - Average run time for number of objects



Graph 2 - Efficiency for number of objects

## 2.4 Output

In this chapter we will discuss the output of the tests and the output of our sorting measurement class.

### 2.4.1 SortingMeasurement

In the code snippet below you can see the output of our sorting measurements. The first lines that get printed in the console are the stats of each individual sort. This includes the name of the algorithm, the amount of archers, the time in nanoseconds and the time in seconds. Because this gets printed over 300 times only the last line of this is present in the code snippet. When each sort print is printed, the total time, total iterations and the averages are calculated. This is then printed as seen below.

```
// Repeating for every sort
Algorithm: Collections sort Archers count: 3276800 Time in nanoseconds: 5250113200,000000 Time in
seconds: 5,250113

|-----| Totals and averages |-----|
Total of insertion: 55.2356774
Iterations of insertion: 9
Average of insertion: 6.137297488888889

Total of quick: 32.1028785
Iterations of quick: 15
Average of quick: 2.1401919

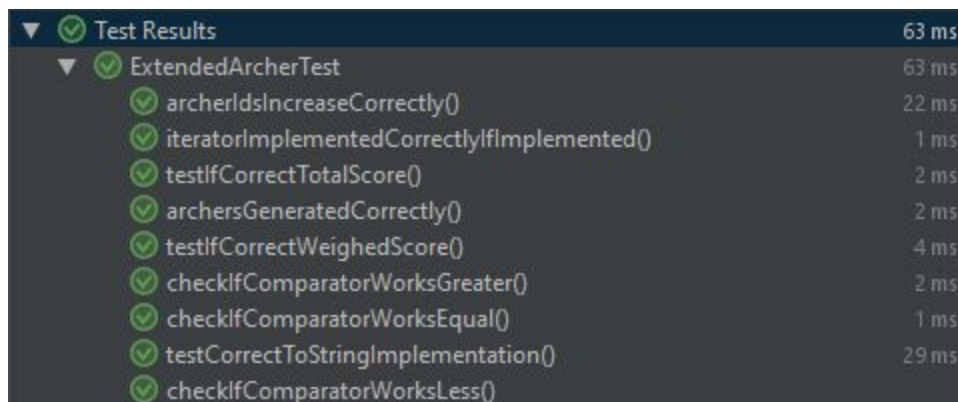
Total of collection: 12.196270300000002
Iterations of collection: 16
Average of collection: 0.7622668937500001
```

## 2.4.2 Testing

To test our application we created some unit tests using JUnit. In this chapter we will discuss our extra tests we created.

### 2.4.2.1 ExtendedArcherTest

To test the archer class there were some pre written tests created beforehand. In addition to these tests. We wrote extra tests to make sure the archer class is working correctly. In the image below there is an output of our Extra tests with the pre written tests included.

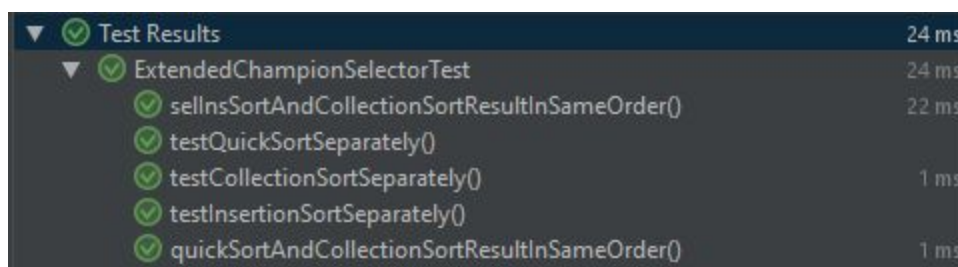


A screenshot of a JUnit test results window. The window has a dark background with a light blue header bar. The header bar contains a green checkmark icon, the text 'Test Results', and '63 ms'. Below the header bar, there is a tree view showing the test results. The tree view has a green checkmark icon next to 'ExtendedArcherTest' and '63 ms'. Under 'ExtendedArcherTest', there are ten test methods, each with a green checkmark icon and its execution time in milliseconds.

Test Method	Time
archerIdsIncreaseCorrectly()	22 ms
iteratorImplementedCorrectlyIfImplemented()	1 ms
testIfCorrectTotalScore()	2 ms
archersGeneratedCorrectly()	2 ms
testIfCorrectWeighedScore()	4 ms
checkIfComparatorWorksGreater()	2 ms
checkIfComparatorWorksEqual()	1 ms
testCorrectToStringImplementation()	29 ms
checkIfComparatorWorksLess()	

### 2.4.2.1 ExtendedChampionSelectorTest

In addition to testing the Archer class we need to test if the sorting algorithms work correctly. For this we added a couple of tests to the batch of unit tests. In the image below there is an output of our tests with the pre written tests included.



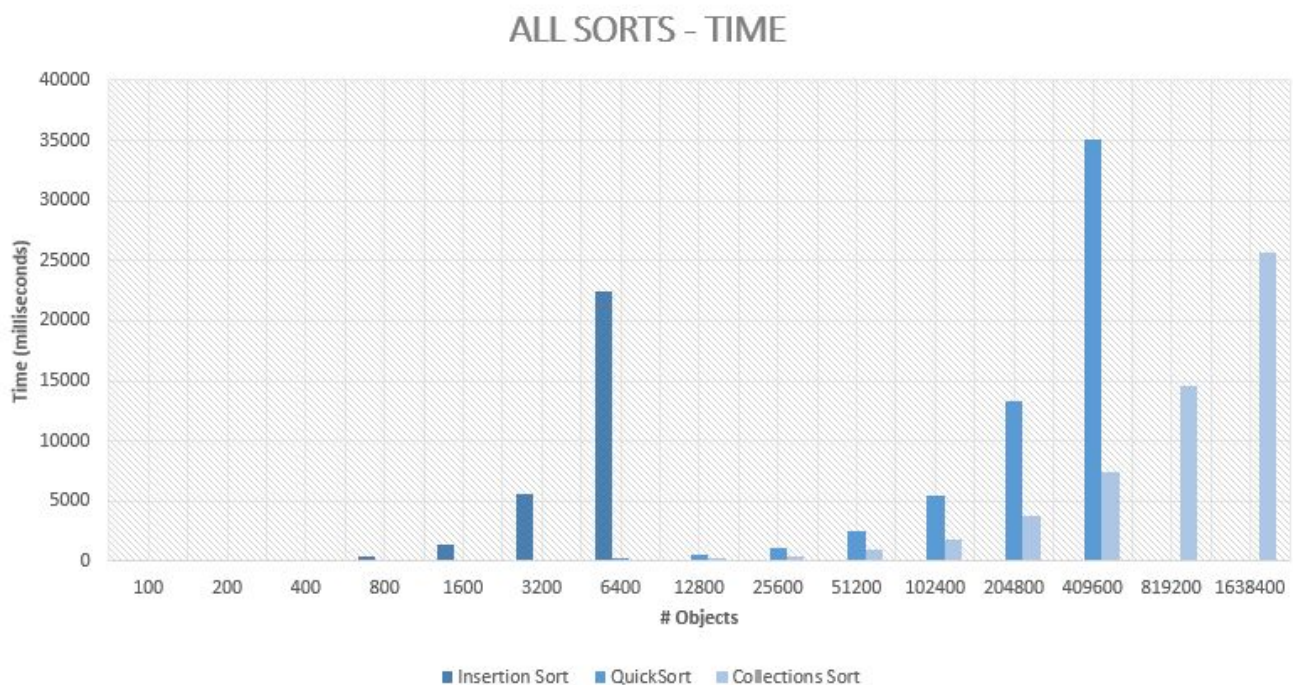
A screenshot of a JUnit test results window. The window has a dark background with a light blue header bar. The header bar contains a green checkmark icon, the text 'Test Results', and '24 ms'. Below the header bar, there is a tree view showing the test results. The tree view has a green checkmark icon next to 'ExtendedChampionSelectorTest' and '24 ms'. Under 'ExtendedChampionSelectorTest', there are five test methods, each with a green checkmark icon and its execution time in milliseconds.

Test Method	Time
sellInSortAndCollectionSortResultInSameOrder()	22 ms
testQuickSortSeparately()	
testCollectionSortSeparately()	1 ms
testInsertionSortSeparately()	
quickSortAndCollectionSortResultInSameOrder()	1 ms



### 3 Conclusion

What we have learned from this assignment is how the three sorting algorithms we implemented work and how efficient they are at sorting the scores using a custom comparator implementation. From sorting small amounts of data to very high numbers, we have measured the time it took to completely sort each set. With this data, we can compare the efficiency of each sorting algorithm to see which is best to use in what case.



What we can conclude is that the built in Collections sort is the most efficient, as can be seen above. It was the only algorithm that could sort over 1.6 million archers in under 20 seconds. The quicksort however, was also efficient overall but needed more time as it progressed so it couldn't sort the same amount under 20 seconds. When we look at the insertion sort, it was already over the 20 second limit at 6400 objects, making it very inefficient with a higher amount of objects.

Though the insertion sort is inefficient with higher amounts of objects, it is still worthy to use when it only has to sort a lower amount of objects because it's easy to implement and uses no extra memory as it sorts in-place.

This concludes this report of assignment 2 Archer.

---

## Sources

[1] `SortingMeasurement.java`

[2] <https://www.geeksforgeeks.org/insertion-sort/>

[3] <https://www.geeksforgeeks.org/quick-sort/>

[4] <https://www.geeksforgeeks.org/collections-sort-java-examples/>