# Time Measurements

Dr Jonas Lundberg

Software Technology Group
Department of Computer Science, Linnaeus University
Veides Plats 7, SE 351 95 V¨axj¨o, Sweden
Jonas.Lundberg@lnu.se

*Abstract*— **This is a short report intended as an example for the reports to be submitted in Exercises 3 and 4 in Assignment 4 in the course 1DV507 – Programming and Data Structures. The task handled in this report is to find how many integers can be sorted in 1 second when using: a) integer arrays and the method sort in class java.util.Arrays, and b) ArrayList<Integer> and the method sort in class java.util.Collections.**

**Our experiments show that about 12.2 millions integers can be sorted in 1 second when using integer arrays, and about 3.1 millions when using an ArrayList.**

## I. Exercises

Students are required to submit a short report in Assignment 4 in the course 1DV507 – Programming and Data Structures. The purpose of this report is to show an example of what a such a report might look like.

The problem we handle in this report is the following: How many integers can be sorted in 1 second when using: a) integer arrays and the method `sort` in class java.util.Arrays, and b) ArrayList<Integer> and the method `sort` in class java.util.Collections.

## II. Experimental Setup

All experiments was done on a MacBook Pro with an Intel Core i7 processor (2.2GHz) with 8GB of memory. We used JavaSE-1.8 and during the experiments we also allowed the Java Virtual Machine to use a heap space of 4GB using the VM arguments: `-Xmx4096m -Xms4096m`. Furthermore, we closed all other applications while performing the experiments and made sure that neither input data generation nor print-outs was done during the time measurements. We used the clock `System.currentTimeMillis()` ion all our measurements.

We used the same approach in all experiments. Each time measurement is an average of 10 consecutive runs (i.e., sorting an array or a list) and we always make sure to generate our input data (random arrays/lists) before the time measurements start. We also run a garbage collection just before each time measurement to avoid having memory problems (and forced garbage collections) while we are sorting. As an example, the code below shows the code used for measuring the time of sorting integer arrays:

```
public static int oneArrayRun(int size, int runs) {
   Timer timer = new Timer();
   ArrayList<int[]> arrays = randomArrays(size,runs);
   timer.gc(); // Garbage collection timer.tic();
        // Start clock
   for (int i=0;i<arrays.size();i++) {
      int[] a = arrays.get(i);
     Arrays.sort(a);   // Actual sorting
   }
 int milliSec = timer.toc();// Stop clock
  return milliSec/runs;      // Average time
}
```

Here `Timer` is our own class to handle time and memory measurements and `randomArrays` is a help method used to generate random arrays.

For each experiment we started with a try-and-error approach to find a size interval (Min,Max) that covers an execution time of about 1 second. We then divided this interval into about 10 subintervals and performed an experiment for each of these subintervals. As an example, for sorting integer arrays we found that sizes around 12 million elements can be sorted in about 1 second and that the size interval [11,13] millions definitely covers the size we are looking for. A print-out when handling integer arrays using method `sort` in class java.util.Arrays:

```
Array Experiments
 Warming up
```

| Variable Type | Character amount | Repeats | String Length |
|---|---|---|---|
| String | 1 | 69147 | 69147 |
| String | 100 | 74149 | 74149 |
| StringBuilder | 1 | 464560 | 123248679 |
| StringBuilder | 100 | 456400 | 150994943 |

We always run a few warm-up runs to make sure that the JVM is fully optimized when we start the actual time measurements. That is, to avoid that the JVM stops the execution, rewrites the code to improve the performance, and then continues. This stop-and-rewrite process costs a bit of time in the first runs but results in slightly faster execution in the following runs. Also, we print the amount of used memory after each run to give us an indicator of the memory situation.



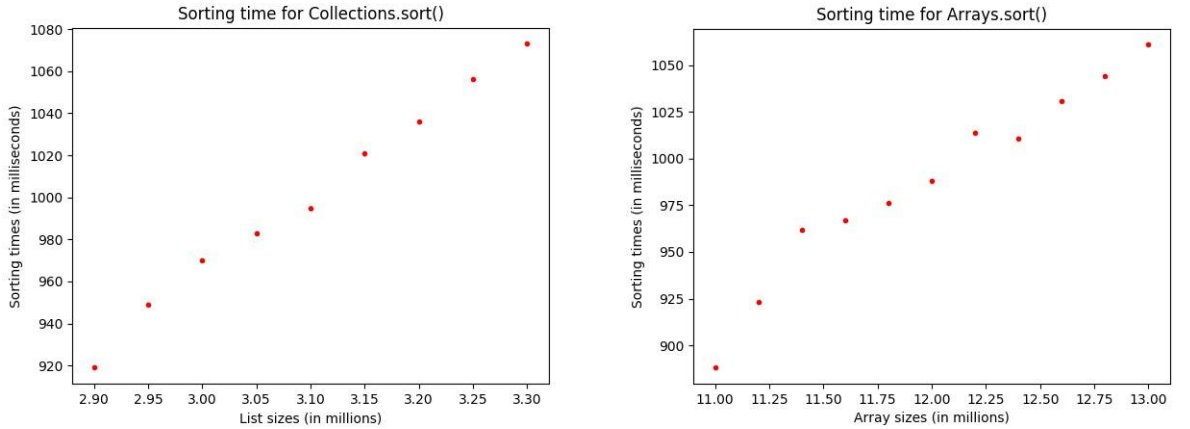


Fig. 1
List experiments (left) and array experiments (right).

## III. Sorting Integer Lists

The table below shows the time (milliseconds), size (millions), and memory (Mb) for each experiment, and Figure 1 plots the results.

| Sorting Lists | | |
|---|---|---|
| Size | Time | Memory |
| 2.9 | 919 | 724 |
| 2.95 | 949 | 734 |
| 3.0 | 970 | 745 |
| 3.05 | 983 | 757 |
| 3.1 | 995 | 767 |
| 3.15 | 1021 | 778 |
| 3.2 | 1036 | 789 |
| 3.25 | 1056 | 801 |
| 3.3 | 1073 | 811 |
| | | |
| | | |

| Sorting Lists | | |
|---|---|---|
| Size | Time | Memory |
| 11.0 | 888 | 449 |
| 11.2 | 923 | 457 |
| 11.4 | 962 | 465 |
| 11.6 | 967 | 473 |
| 11.8 | 976 | 481 |
| 12.0 | 988 | 489 |
| 12.2 | 1014 | 497 |
| 12.4 | 1011 | 505 |
| 12.6 | 1031 | 513 |
| 12.8 | 1044 | 521 |
| 13.0 | 1061 | 529 |

The first thing to observe is that list numbers look OK. The left-hand side plot indicates a steady increase in time when increasing the list sizes (no outliers) and the rather stable memory numbers in the left-hand side table (well below the heap size memory) indicate that we did not have any severe memory problems during the experiments. Hence, we conclude that the experiments can be trusted.

An estimate of number of elements that can be sorted in 1 second can be done if we imagine a line fitted to the plotted data, and try read out the size which corresponds to 1000 milliseconds. That gives about 3.1 million elements.

## IV. Sorting Integer Arrays

The array experiment (right-hand side table and figure) is not as good as the list experiments. We have certain data points (around 11.4 and 12.2 millions) that looks a bit off. The problem at 12.2 millions is especially problematic since it is close to where we expect to find the result (elements sorted in 1 second). In any case, using the ”imagine a line fitted” approach gives about 12.2 million elements.

It is also worth noting the difference in the amount of used memory in the two experiments. That is, around 750MB (size 3 millions) for lists and 500MB (size 12 millions) for arrays. This shows that using arrays is much more memory efficient compared to ArrayList.

## V. Possible Improvements

Our approach of estimating the sizes which correspond to 1 second by ”imagine a line fitted” is not very trustworthy. A much better result can be obtained by linear regression (see course 2DV516 - Introduction to Machine Learning) which also would give us a confidence interval.

Another way to improve the experiments is of course to increase the number of measurements, especially more measurements close to the estimated values (3.1 and 12.2 millions). The need for more experiments is most urgent in the array experiment where the plot indicates measurement problems around 12.2 million elements. One way to do it would be to repeat the experiments using the same number of measurements but with smaller intervals (e.g., [11.5, 12.5] for the integer arrays).

In conclusion, from a performance perspective, sorting arrays is both faster and more memory efficient.