Time Measurements

Lab 4 Exercise 3-4
Love Samuelsson
ls223qx@student.lnu.se

Abstract:
On my hardware, ~800 000 integers can be sorted in one second using my own implementation of insertion sort, while around 14 M million integers can be sorted in one second using my implementation of merge sort. The performance difference between merge sort and insertion sort is around 17 times.

The difference between concatenating strings directly and using a StringBuilder is quite apparent, with a greatly improved efficiency (over 240 000%) for the StringBuilder when comparing the results between short string appending on concatenation.

Introduction:
The purpose of these exercises are to examine the amount of work that can be done in one second during different operations when handling data. In this lab, different ways of combining strings and different ways of sorting arrays are examined for their efficiency.

Setup:
Experiments were done on a custom-built desktop with an i5-4690k (OC to 4.5GHz), 16GB of RAM and a modern SSD. To avoid heap space issues, the programs were partially split between different java files.

System.currentTimeMillis() was used as timing for the String experiments, a self-built Timer class were used for the sorting experiments which slims down the main code, but essentially provides the same functionality. The java files all had a few 'dry-runs' before actual data is collected, in order for the JVM to optimise itself. Any unnecessary background programs were closed as to not interfere with the test.

Experiments with String concatenation:
A generic version of the code used for the tests can be seen below:

```
end = System.currentTimeMillis() + 1000; // One second delay.
            while (end > System.currentTimeMillis()) {
                  string += "c";  // String to be concatenated
                  counter++;
            }
```

At runtime, the counter will count the number of concatenations done while the condition is true, which means that 'counter' will record the amount of concatenations done within one second.

Results for short Strings (Concatenation)

N	1	2	3	4	5
Count	36721	38319	36811	36696	37323
Final length	36721	38319	36811	36696	37323

Average count: 37 147

For long strings, an 80-character string of semi-random characters was used.
Specifically, this string was used:
"lkjtnllkhvopqwefjfpoqwepofjvåpiqwepojlkfjpoqwepojöfklvpihqwelkfjrphqwepolnv.,nas"

Results for long Strings (Concatenation)

N:	1	2	3	4	5
Count:	5846	5892	5799	5825	5780
Final length	467680	471360	463920	466000	462400

Average length: 466 272
Average count: 5 828

Experiments with append (StringBuilder):
With a StringBuilder, the final StringBuilder.toString() call also takes a non-trivial amount of time, which is accounted for in the loop condition by simply reducing the amount of time available to append strings until total runtime is 1000ms ∓30ms.

In the short string test, the final string length is equal to the number of appendages. Final convert time is the time it takes for the final toString() call on the StringBuilder object.

Results for short Strings (StringBuilder)

N:	1	2	3	4	5
Count	93890835	87007747	91738747	92013384	91733685
Final convert time (ms)	80	75	84	80	81

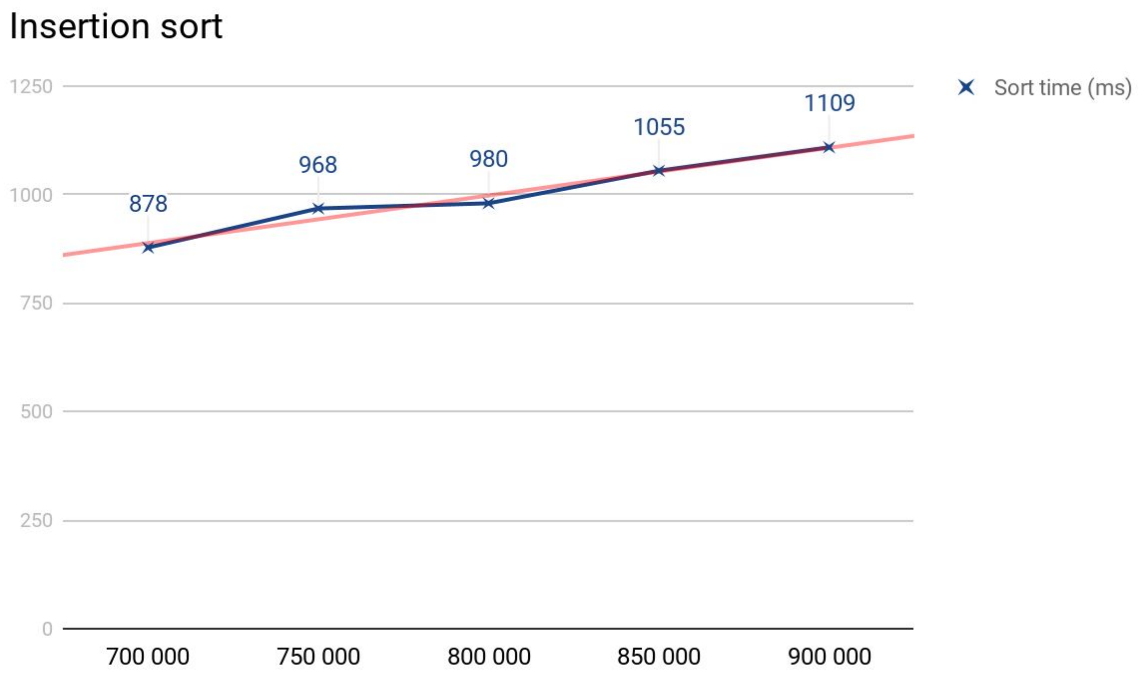
Average count: 91 276 880

Longer strings:
Unfortunately the test for longer strings were inconsistent, it looks like that the conditional as shown in the generic code snippet does not work for some reason, making the loop take longer than intended. The behaviour that was observed on my machine was that then directing the conditional to be true for 650 ms or more, actual runtime would vary wildly from 1200ms to 1600ms.

Experiments with sorting algorithms:
Each result is an average of 5 runs. Memory is calculated using following formula: (32 + 4 * (length of array) / 1024. This formula consists of 32 bytes needed for the object header and 4 bytes for each integer stored in the array. Division by 1024 will give the result in KB. Dividing by another 1024 gives the result in MB. This is a pretty rough estimate, and does not take into account memory usage while using the sorting algorithm. It is simply another way to estimate array size.

Results for *'Insertion Sort'*

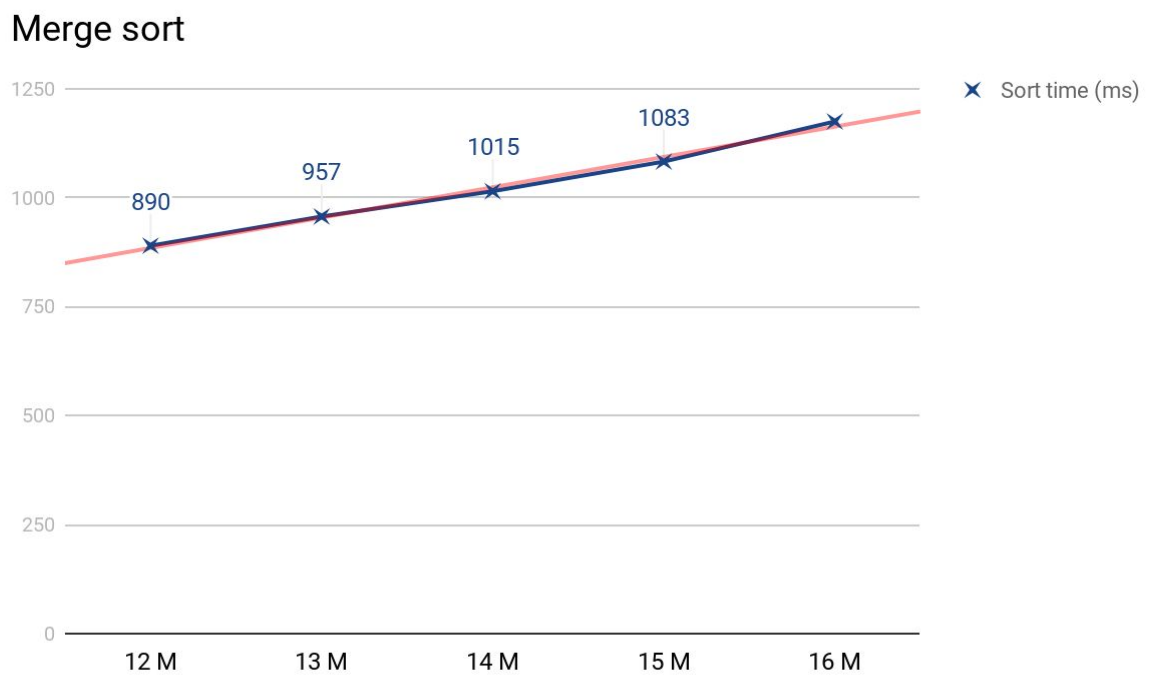
Length (N integers)	Memory (KB)	Sort time (ms)
700 000	2737	878
750 000	2929	968
800 000	3125	980
850 000	3320	1055
900 000	3515	1109



Results for 'Merge Sort'

Length (N integers)	Memory (MB)	Sort time (ms)
12 M	45,7	890
13 M	49,6	957
14 M	53,4	1015
15 M	57,2	1083
16 M	61,0	1175

M is for Mega (10^9)



Final thoughts:
It's unfortunate that the String append test didn't yield any useful data. Attempts were made to remedy this issue by using heap space JVM arguments, which only yielded small improvements at best. "-Xms10G and -Xmx10G" was used, although experimentation with smaller and larger numbers yielded similar results. Perhaps the reason has something to do with the internal implementation of StringBuilder. Maybe the issue is a large array resize that happens internally in the StringBuilder class after a certain point.

It seems quite plausible that if this processor did not have a 1GHz overclock, the StringBuilder performance might drop considerably enough that the test may work as it is supposed to, circumventing the array resize by simply going slowly enough that it doesn't get to that point.

In the String tests, a clear improvement in performance is seen when moving from concatenation to the StringBuilder append. In a one second duration, it was observed that 2457 times as many String combinations can occur with StringBuilder rather than with concatenations. As for the reason why there might be a large difference in efficiency between the methods of combining, the main reason might be that strings are inherently immutable, meaning that for every concatenation the JVM has to continually build a new string every time. With a StringBuilder object, only one string is created during the final call. Internally, the StringBuilder class instead maintains a character array containing all characters of the String, which is then combined and translated into a String during the final toString() call.

As for the sorting algorithms, it appears that my implementation of insertion sort is rather slow compared to my implementation of merge sort. ~820 000 compared to ~14M is an efficiency difference of around 17 times. The clear conclusion to draw here is that merge sort is the preferred option for larger amounts of data. This is a pretty expected result, since the time complexity of insertion sort is O(n^2), compared to the relatively slim time complexity of O(nlog(n)) for merge sort.