LINNAEUS UNIVERSITY

Lab report

Time measurements

Christoffer Lundström

07/03-2019

IDV507 - Programming and Data Structures

Examinator: Dr Jonas Lundberg

1. INTRODUCTION

It is my firm belief that Science and education bring out the best of humanity. They enable us to push the boundaries of humanity and present new opportunities for all aspects of life. The sciences of Engineering has enabled instant communication across the globe and revolutionized industry, transportation, energy, medicine and many other fields.

One particular dear to me is computer science and software engineering. Over decades computers have taken a stronger foothold in our lives and are now a part of everyday interaction and more subfields of computer science has sprung from our usage of this technology.

Big data is now its own field where extreme amounts of data must be constantly translocated. One important foundation which enable this constant flow of information is data structures. The goal of this report is to analyse old algorithms that still have a large impact on today's communication. Namely insertion-sort and merge sort, the latter invented by Neumann in 1945. We will also look at the concatenation and appending of Strings, comparing the time complexities of each.

1.1. Purpose

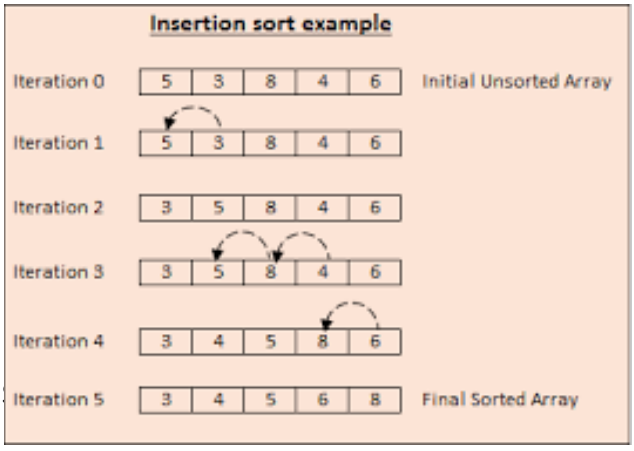
The purpose of this project is to analyse the average times of String concatenation and also comparing the sorting times of the algorithms insertion-sort and merge-sort using Java and IntelliJ. The results will then be presented in an appropriate manner.

1.2 Theory

Insertion Sort

A comparison based algorithm which iterates through a list according to the example in figure 1. The insertion sorts has a time complexity of $O(n^2)$ in the worst case scenario. In the best case scenario insertion sort is $O(n)$ which is a little bit faster than merge sort up to a certain number of comparisons.

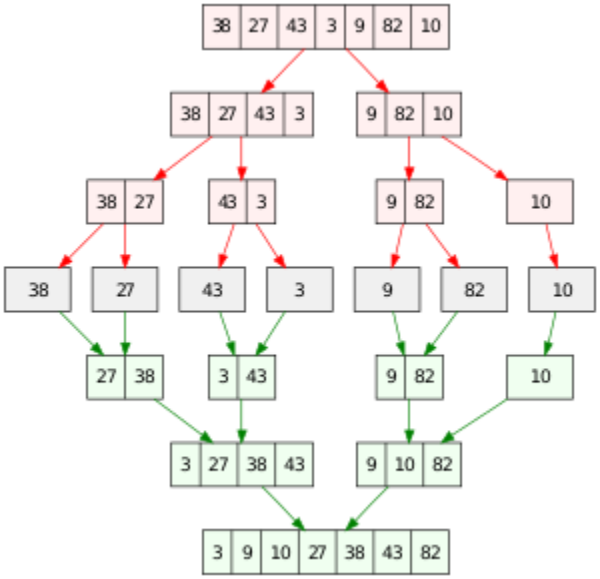
Fig. 1



Merge Sort

The merge sort works by recursively splitting each array in two pieces until there are only one element left in each branch. A comparison is then between each branch as they are merged. This approach's time complexity follows the curve of O(n log(n) which should in most cases be faster than Insertion Sort.

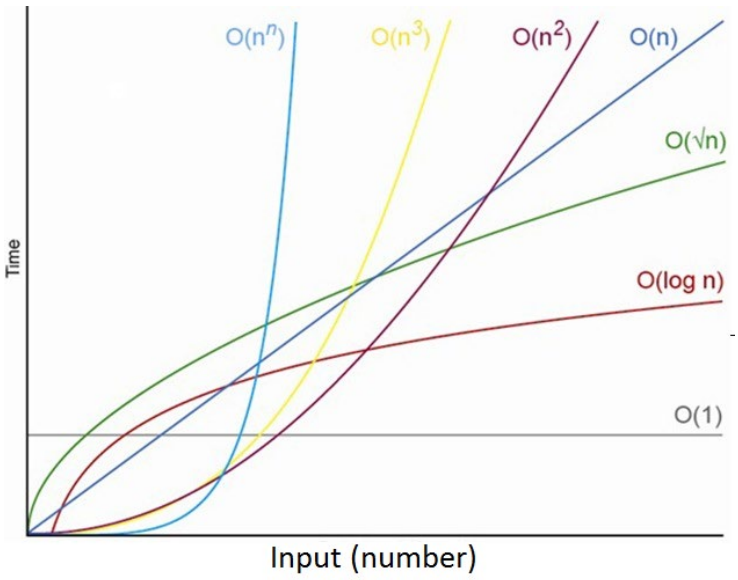
Fig. 2



Time Complexities

In figure 3 the most common time complexities growth curves are shown.

Fig. 3



2. PROCEDURE & MATERIALS

2.1 Experimental Setup

This experiment is run on Windows 10 PRO with the following PC specifications.

Intel Core I7-8700k @ 4.7Ghz (1.2v vCore).
16GB DDR4 @ 3200mhz.
Samsung EVO 960 M.2.

Furthermore the experiment is done using Java 11SE and IntelliJ Community Edition 2018.3.

Measurement

All functions in the experiment are measured using the function in Fig.4 which uses System.currentTimeMillis.

Length value is reset and garbage is collected prior to each measurement.
Each sorting function is run 20 times and an average time is calculated.

Fig. 4

```
public static long measureAverageFuncTime(Runnable func){
    long tmp = 0;
    length = 0;
    Runtime r = Runtime.getRuntime();
    r.gc();
    long start = System.currentTimeMillis();
    for(int i = 0; i < 20; i++){
        func.run();
    }
    tmp = System.currentTimeMillis()-start;
    System.out.println("Average Time: " + tmp/20 + " ms");
    return tmp;
}
```

Each measurement is also preceded by a Warmup used to optimize the JVM for the different sorting and concatenations.

Concatenation of strings

One of the two functions written for concatenation is supplied with a String value which will be concatenated specified amount of times. This string parameter will either be a string consisting of 80 characters or 1.

The other approach is near identical however it uses the StringBuilder class in Java and instead of manually concatenating strings the function append is called.

Sort-algorithms

Two types of sorting algorithms are tested during this phase with two overloads each.
InsertionSort(int[] array);
InsertionSort(String[] array, Comparator comp);
MergeSort(int[] array);
MergeSort(String[] array, Comparator comp);

The string-parameter functions will take a comparator forcing the array to be sorted in the natural order. These functions will all be fed with a randomly generated array of Strings or Integers.

Warmup-routine

Below is the warmup run before each test.

```
StringBuilder ab = new StringBuilder( capacity: 800000);
// Turns out warmup slows down concat and append ALOT
for (int i = 0; i < 100; i++) {
    longStr = longStr + longStr;
    ab.append(longStr);
}
```

3. RESULTS

A table of all data can be found here.

3.1 Concatenation

Below are the results of the tests. You can also find scaled up versions here and here.
Each concatenation of long is the amount of concatenations*80.
Each short concatenation increase the string by 1.
For example 2 long concats = 160 characters.

Fig. 5

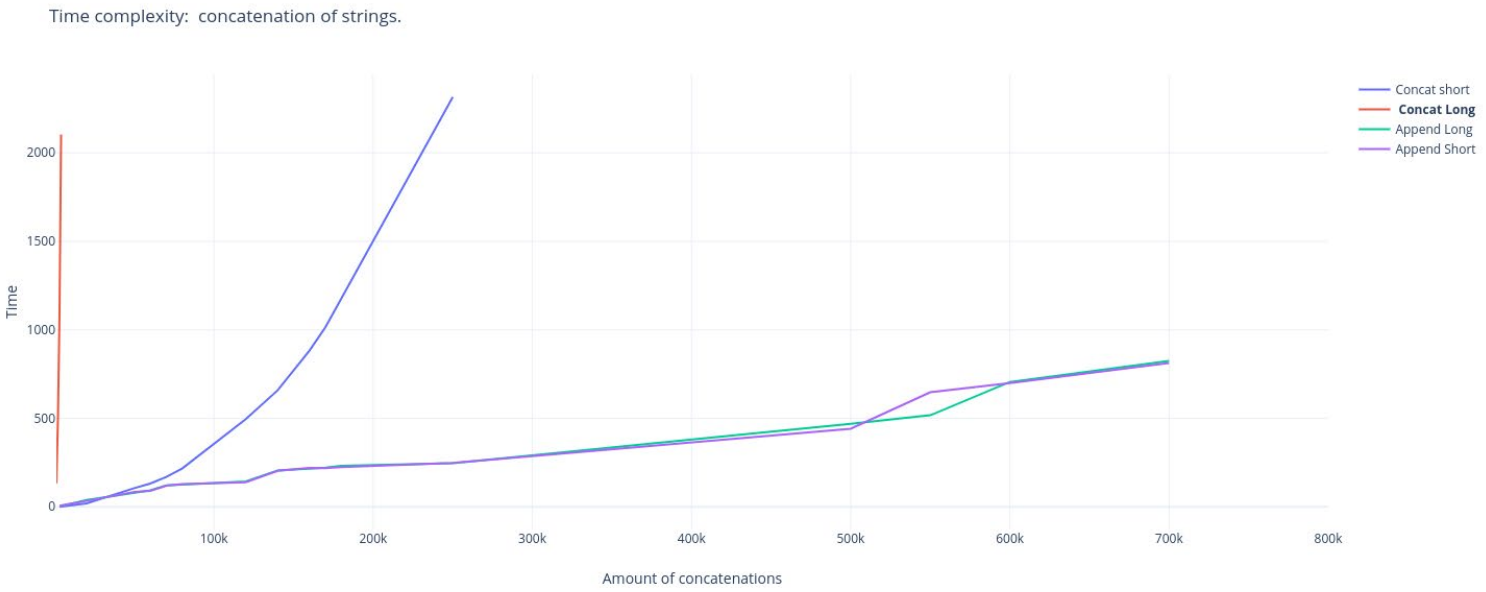


Fig. 6

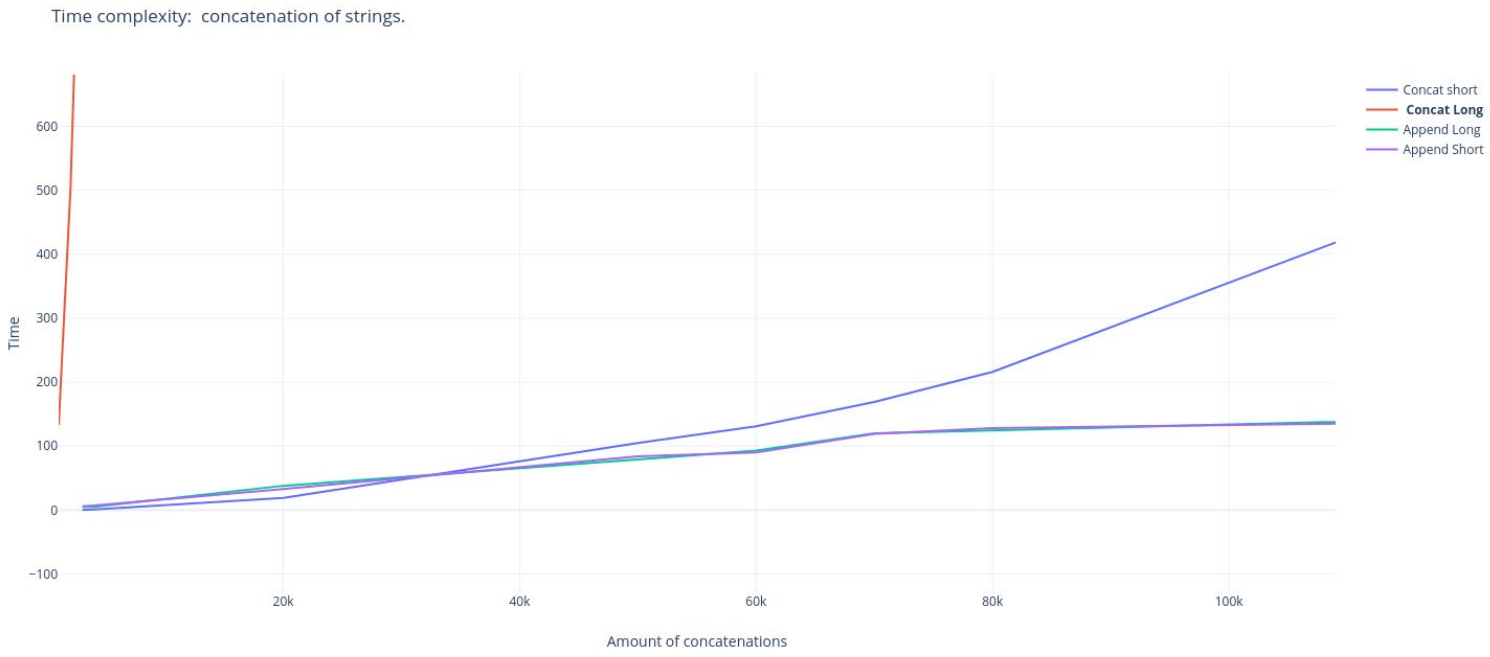
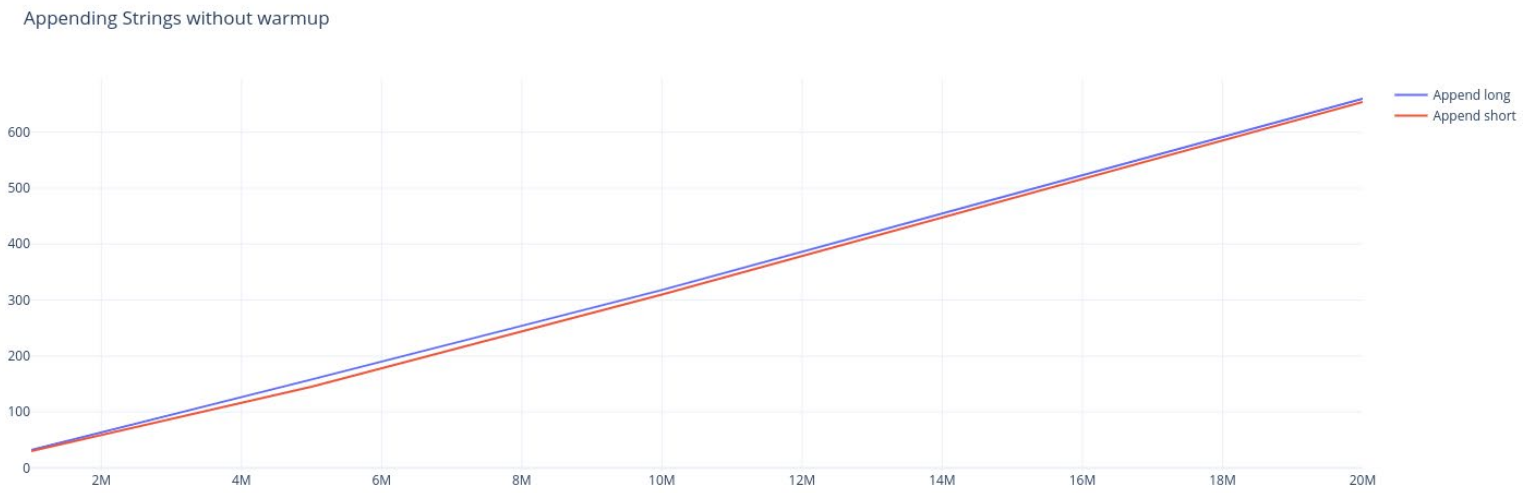


Fig. 7



When we analyse the graphs of fig 5 and 6 we can conclude that append is a generally faster method to use for concatenation. In this experiment Java heap error appeared after around 700k concatenations with the first approach and would not be resolved by increasing heap size of the JVM.

It did however resolve when the warmup routine was removed. That enabled appending up to 20 million Strings before receiving Java heap error. It seems the warmup routine was faulty or not working as intended, therefore it tested both with and without.

Immutable Strings

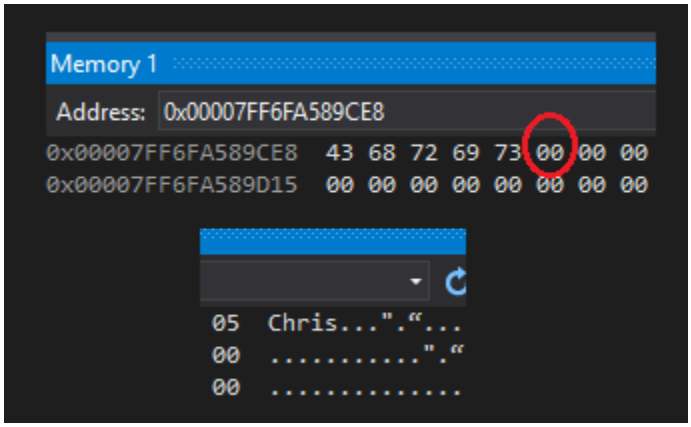
One reason for the Java heap error is that the process of concatenation includes creating a new String variable for each concatenation. This is an extremely CPU-heavy task compared to appending using a *Stringbuilder* and gets heavier the longer the String gets. Thus makes it completely non-viable in a big-data setting.

Strings are also created because Strings in Java are immutable, meaning they may not be altered after instantiation. This behaviour is common among high-level languages such as Java or C# and it provides a basis for encapsulation and safety for the internals of an object.

If Strings were to be mutable they could be subject to change and manipulation through a simple get-call as is the case in languages such as C++ where strings are *const* char pointers or essentially arrays of chars. Even though the *const* keyword is a promise to the compiler that the Object will not change it is still possible to manipulate it in memory. It is still not a safe approach in these languages because of the way a String is read.

Reading strings

Fig. 8
Null termination byte followed after 5 Chars



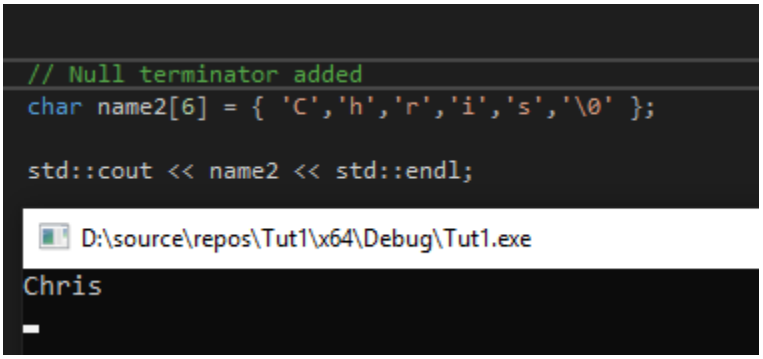
With this short demonstration in C++ we can see that should the manipulation of the String change the size or exclude the final null termination character (fig 8), the result would be as shown in Fig 9.

This is because the compiler cannot know where the String (char array) ends.

Moving the null termination character would also result in a shorter or longer String.

Fig. 9

Fig. 10
In this example the null terminator is added manually to the char array and the String is read correctly.



Stringbuilder

Using a *Stringbuilder* to append a String essentially creates a mutable String object which is a lot faster to manipulate than using standard concatenation.

Because the *Stringbuilder* handle these operations for us we never need to worry about *null* termination characters missing or other errors because they are handled within the class.

Sorting comparison

Below is the comparison between the sorting algorithms. Also a scaled up version here and here.

Fig. 11

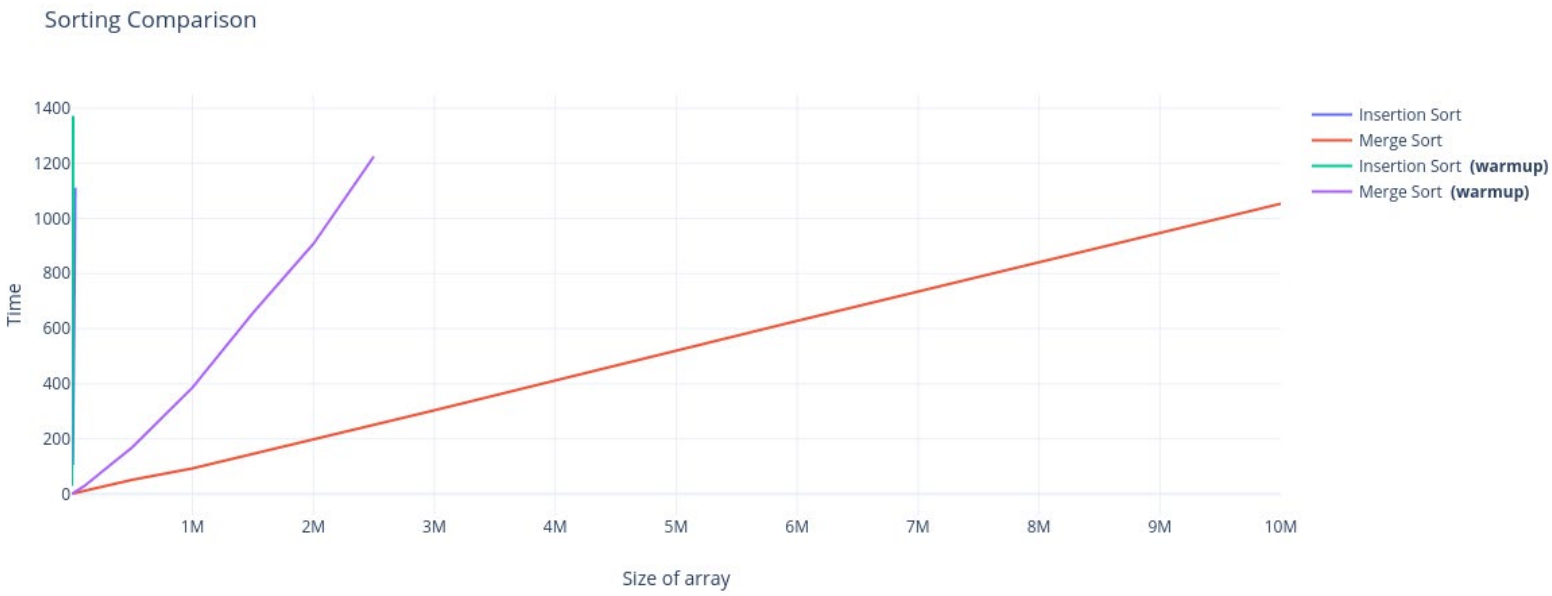
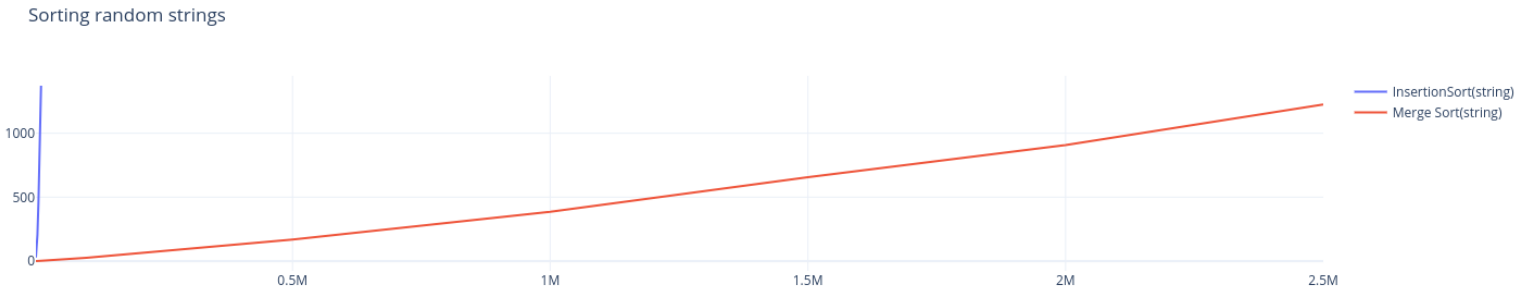


Fig. 12



Analysing these graphs it seems that using a warmup prior to starting the sort has increased the time taken to sort quite substantially. This might be a subject to dig further into.

We can also conclude as expected that merge sort is indeed much faster. Insertion sort seem to follow $O(n^2)$ in this case whilst merge sort looks to increase linearly, more or less $O(n)$ or $O(\log n)$ in both the String and the Integer case.

In the case of the string sorting there can still be room for error since the Strings used for sorting have been randomly generated. Even though each function was run 20 times and averaged it's not a guarantee for accurate measurements.

4. CONCLUSION & DISCUSSION

This project presented a few challenges. One of these were obviously the Java heap errors. It made it very difficult to get accurate readings of the results. I spent hours on trying to optimize the code and increasing the java heap size before I realised that the warmup function was a root cause.

My conclusions to this experiment confirms the prior time complexity notions. Merge sort is superior over Insertion sort in most cases. However implementing a merge sort might be overly complex for small data structures. As in most cases a requirement analysis should serve as a base for the decision of sorting algorithm together with a weighing of advantages and disadvantages.

As for concatenation of Strings we can conclude that any situation where a String needs concatenation numerous times a Stringbuilder is the preferred approach. Editing an already existing object will most likely be faster than allocating a new object before editing. This includes Strings as well, however the way Strings are immutable makes them unsuitable for concatenation, hence Stringbuilder should serve as a workaround for the immutable String problem.

Furthermore the experiment could be enhanced further by delving deeper into how objects are allocated behind the scenes and looking at the specifics of each sorting algorithm, but this is a task more suitable when working with other languages such as Assembler or C.

5. REFERENCES

Java 11 SE

https://www.java.com/sv/download/

IntelliJ Idea Community Edition

https://www.jetbrains.com/idea/download/#section=windows

Fig 5-6

https://plot.ly/~LuRRE/9

Fig 7

https://plot.ly/~LuRRE/13/

Fig 11

https://plot.ly/~LuRRE/11

Fig 12

https://plot.ly/~LuRRE/15/

Time Complexity

http://codedreaming.com/wp-content/uploads/growth-of-function.png

Merge Sort

https://upload.wikimedia.org/wikipedia/commons/thumb/e/e6/Merge_sort_algorithm_diagram.svg/300px-Merge_sort_algorithm_diagram.svg.png

Insertion Sort

1. https://www.opentechguides.com/images/howto/howto_5001.png

Table of data

https://drive.google.com/file/d/1likK7WP0qgTr8pFxFnoXnGxnmxilgxrT/view?usp=sharing