UFO - Assignment 2

Adam Lass, Rasmus Helsgaun, Stephan Djurhuus, Pernille Lørup 24/11/2020

Objectives

You need to make it run at least 50% faster than it is now, and should be able to get to around 100% faster.
Hand-in: should contain the following:

- Documentation of the current performance (remember mean and standard deviation – see the Sestoft paper!)
- An explanation of the bottlenecks in the program.
- A hypothesis of what is causing the problem,
- A changed program which is improved to solve the problem.
- Documentation of the new performance.

Notice: there might be more than one optimization needed to achieve optimal performance.

Hypothesis & Bottlenecks

A call is made to read a system file each time **FileReader** is called. A **FileReader** will make 256 calls for reading 256 characters from file.
try/catch is reducing the performance due to its check and preparation of potential exceptions.

```
1  private static void tallyChars(Reader reader, Map<Integer, Long> freq) ↩
       throws IOException {
2      int b;
3      while ((b = reader.read()) != -1) {
4          try {
5              freq.put(b, freq.get(b) + 1);
6          } catch (NullPointerException np) {
7              freq.put(b, 1L);
8          };
9      }
10 }
```

Optimization

Tally Chars Custom 1

This method uses **freq.getOrDefault** instead of using **try/catch** that checks if the **Key** exists and then throws an exception if it doesn't.

```
1 private static void tallyCharsCustom1(Reader reader, Map<Integer, Long> ↩
      freq) throws IOException {
2     int b;
3     while ((b = reader.read()) != -1) {
4         freq.put(b, freq.getOrDefault(b, 0L) + 1L);
5     }
6 }
```

Tally Chars Custom 2

In addition to **tallyCharsCustom1** this method makes use of the **BufferedReader** which uses a memory buffer to optimize reading time.

```
1 private static void tallyCharsCustom2(Reader reader, Map<Integer, Long> ↩
      freq) throws IOException {
2     int b;
3     BufferedReader br = new BufferedReader(reader);
4
5     while ((b = br.read()) != -1) {
6         freq.put(b, freq.getOrDefault(b, 0L) + 1L);
7     }
8 }
```

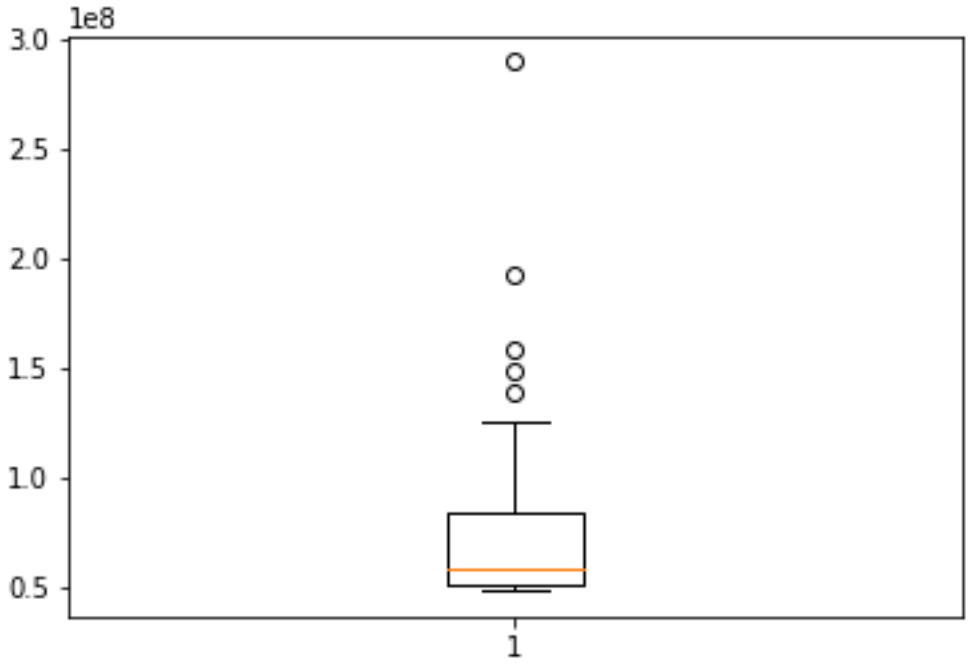
Tally Chars Custom 3

Using **byte[]** optimizes performance because it is a simple data structure. This reduces unnecessary operations and therefore optimizes performance time.

```
1 private static void tallyCharsCustom3(byte [] fileBytes, Map<Integer, Long↩
    > freq) throws IOException {
2     int singleInt;
3     for(byte b : fileBytes) {
4         singleInt = (int) b;
5         freq.put(singleInt, freq.getOrDefault(singleInt, 0L) + 1);
6     }
7 }
```

Performance

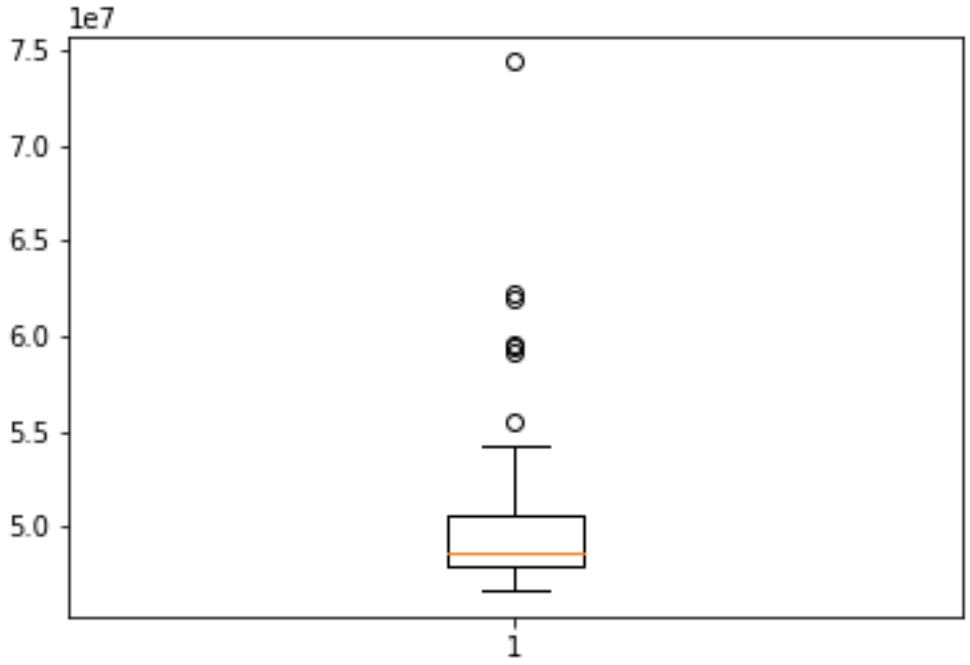
Tally Chars



std: 44704672.3019

mean: 77245155.5800

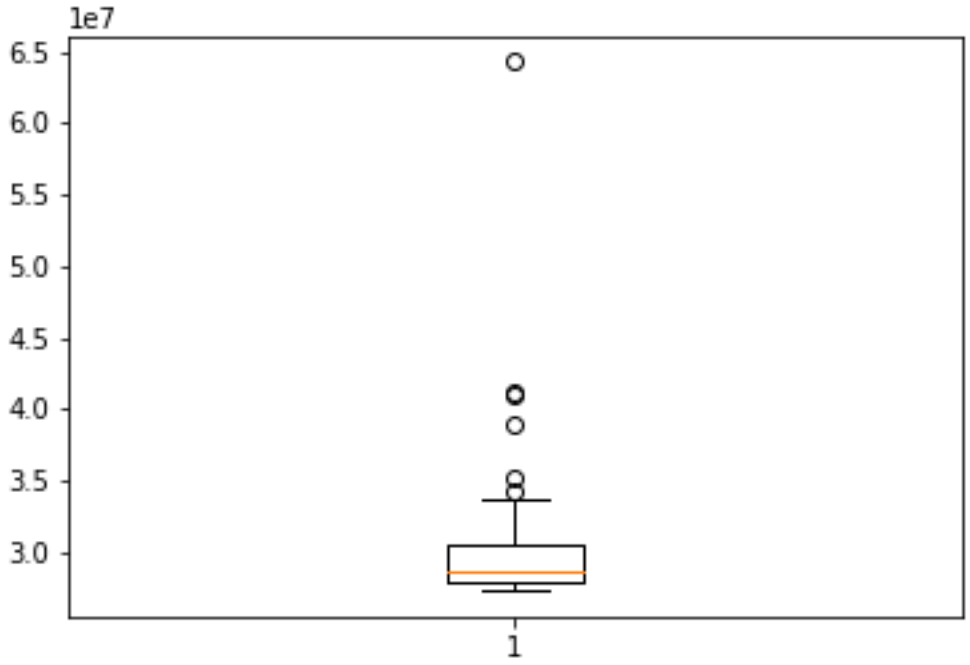
Tally Chars Custom 1



std: 5212312.7555

mean: 50741955.4000

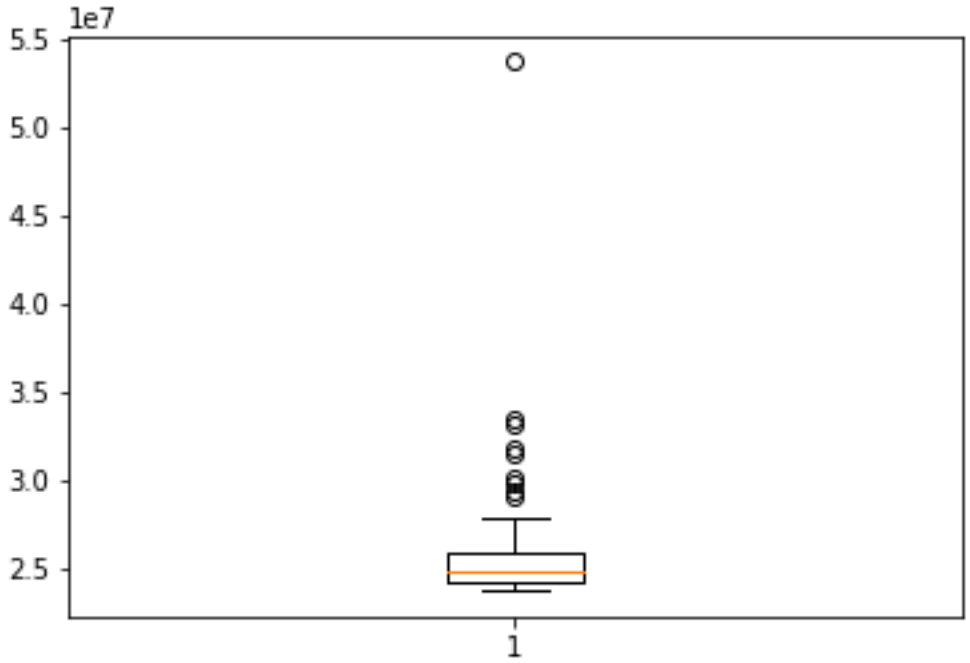
Tally Chars Custom 2



std: 5884498.2695

mean: 30583595.9400

Tally Chars Custom 3



std: 4698149.1470
mean: 26328980.2800

Specifications

The code was executed on a MacBook Pro (Retina, 13-inch, Early 2015) with a 2,9 GHz Dual-Core Intel Core i5 Processor. The MacBook has a memory of 8 GB 1867 MHz DDR3.
The code was executed with Java version 13.0.2.

Conclusion

The third alteration of the the **tallyChars** method has the best performance with approximately **293.38%**

Resources

We used a Jupyter Notebook to analyse the collected performance data.
Java code solution