Exploration and Presentation Assignment 3

Optimization

Simon Schønberg Bojesen - cph-sb339@cphbusiness.dk
Kenneth Hansen- cph-kh415@cphbusiness.dk
Frederik Blem - cph-fb114@cphbusiness.dk
Martin Høigaard Cupello - cph-mr221@cphbusiness.dk

April 21, 2021

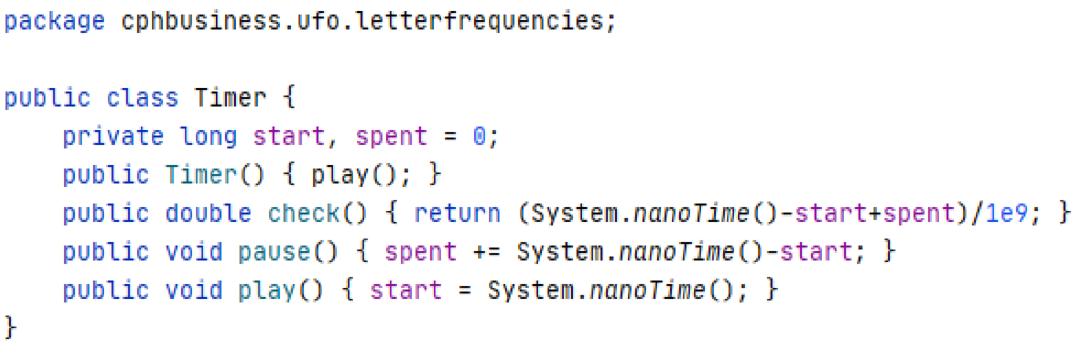
1 Introduction

We chose to optimize the program letterfrequencies. This program opens any textfile and counts all instances of each letter occuring in the textfile, and prints the results. We had as assignment to optimize the programs performance by at least 50%. As part of the assignment we will figure out where we can make the most impactful changes. After we have optimzed the program we will compare the two programs to see if we are able to achieve a better result.
Better optimized programs will save significantly more time on slower systems and will give a better user experience.

2 Documentation of the current performance

First we needed to document the current performance to be able to show how the current program performed. To do that we had to add a Timer class to time the programs runtime.

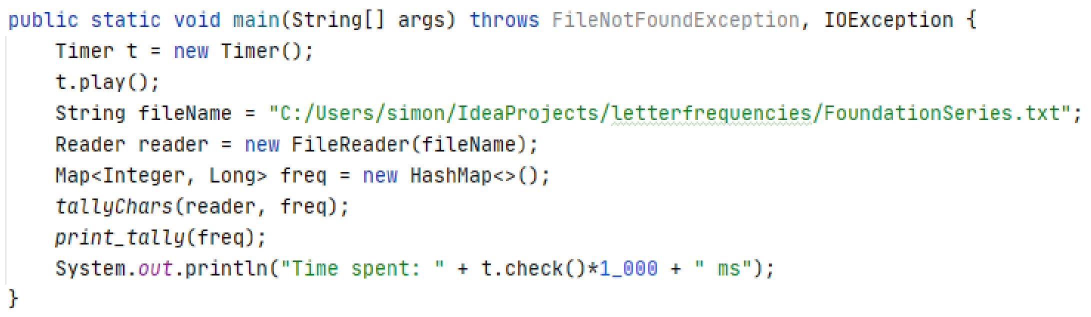
Figure 1: Timer.java



```
package cphbusiness.ufo.letterfrequencies;

public class Timer {
    private long start, spent = 0;
    public Timer() { play(); }
    public double check() { return (System.nanoTime()-start+spent)/1e9; }
    public void pause() { spent += System.nanoTime()-start; }
    public void play() { start = System.nanoTime(); }
}
```

Figure 2: Main.java

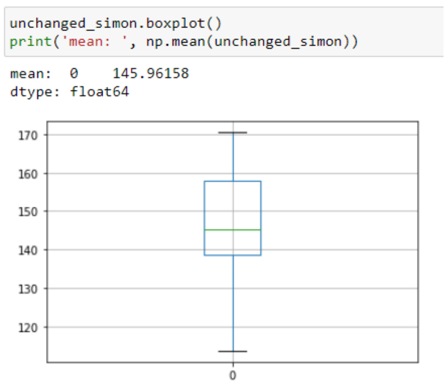


```
public static void main(String[] args) throws FileNotFoundException, IOException {
    Timer t = new Timer();
    t.play();
    String fileName = "C:/Users/simon/IdeaProjects/letterfrequencies/FoundationSeries.txt";
    Reader reader = new FileReader(fileName);
    Map<Integer, Long> freq = new HashMap<>();
    tallyChars(reader, freq);
    print_tally(freq);
    System.out.println("Time spent: " + t.check()*1_000 + " ms");
}
```

After making these changes each person in our group ran 10 individual speedtests of the program on their systems. These test results were then put in our own named files in the testnotes folder. We then made a boxplot showing the results of our tests, and calculated the means. From these means we calculated the goal performances each of our systems needed to reach by optimization.

Figure 3: Boxplot of all test results

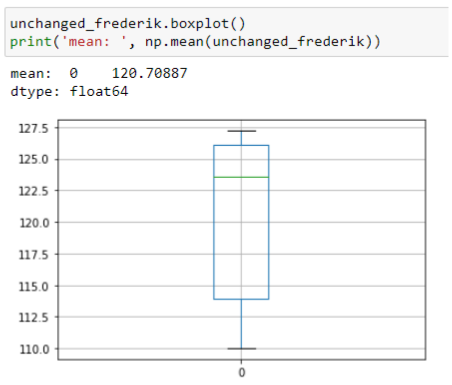
(a) Simon's tests.



```
unchanged_simon.boxplot()
print('mean: ', np.mean(unchanged_simon))
```

```
mean:  0    145.96158
dtype: float64
```

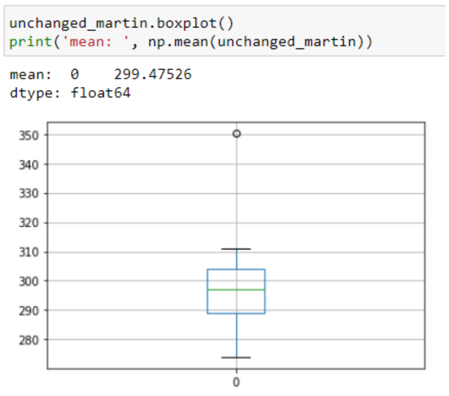
(b) Frederik's tests.



```
unchanged_frederik.boxplot()
print('mean: ', np.mean(unchanged_frederik))
```

```
mean:  0    120.70887
dtype: float64
```

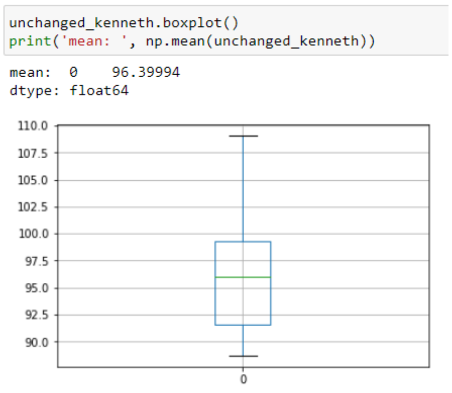
(c) Martin's tests.



```
unchanged_martin.boxplot()
print('mean: ', np.mean(unchanged_martin))
```

```
mean:  0    299.47526
dtype: float64
```

(d) Kenneth's tests.



```
unchanged_kenneth.boxplot()
print('mean: ', np.mean(unchanged_kenneth))
```

```
mean:  0    96.39994
dtype: float64
```

As we can see our systems performance varied greatly. We then calculated minimum goals for each of us from these means. The goal is to increase performance as measured by time.

Simon mean: 145.96158
Frederik mean: 120.70887
Martin mean: 299.47526
Kenneth mean: 96.39994

These numbers needs to be reduced by 50% as per the assignment. We have calculated the goal means to be:

Simon: 108.75
Frederik: 90.5316525
Martin: 224.606445
Kenneth: 72.299955

3 Bottlenecks and hypothesis of issues

3.1 Finding bottlenecks

First thing we needed to do was figuring out what method would give us better performance by optimization. Therefore we ran the program in IntelliJ Ultimate, with a profiler to see which methods were the most expensive.

Figure 4: Profiler Flame Chart

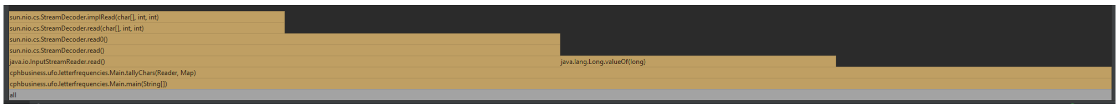
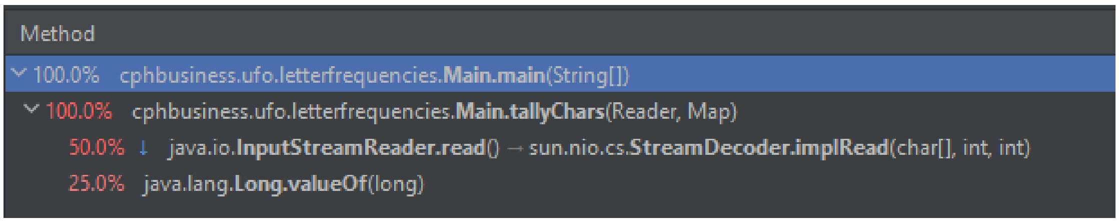


Figure 5: Profiler Call Tree



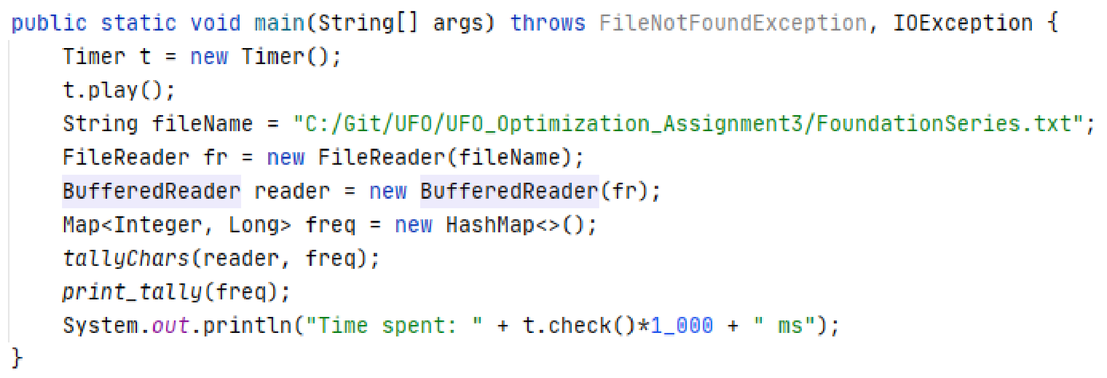
```
Method
⌄ 100.0%   cphbusiness.ufo.letterfrequencies.Main.main(String[])
  ⌄ 100.0%   cphbusiness.ufo.letterfrequencies.Main.tallyChars(Reader, Map)
      50.0% ↓  java.io.InputStreamReader.read() → sun.nio.cs.StreamDecoder.implRead(char[], int, int)
      25.0%  java.lang.Long.valueOf(long)
```

These charts clearly state that all of the costly functionality is in the method tallyChars. We then moved the timer around to check how fast the other function: "print_tally" was as this method was not showing up in our profiler. It averaged at about 5 ms for simon, and we therefore concluded it wasnt worth optimizing.

3.2 Why is tallyChars so expensive

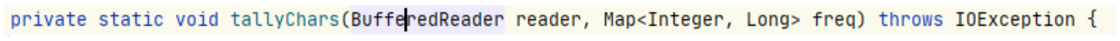
As we looked at he profiler it was quite clear to us that we needed to improve the speed of reading the file. After searching google, we figured out using a BufferedReader instead might be the way to go.

Figure 6: Changed Reader Type



```
public static void main(String[] args) throws FileNotFoundException, IOException {
    Timer t = new Timer();
    t.play();
    String fileName = "C:/Git/UFO/UFO_Optimization_Assignment3/FoundationSeries.txt";
    FileReader fr = new FileReader(fileName);
    BufferedReader reader = new BufferedReader(fr);
    Map<Integer, Long> freq = new HashMap<>();
    tallyChars(reader, freq);
    print_tally(freq);
    System.out.println("Time spent: " + t.check()*1_000 + " ms");
}
```

Figure 7: Changed tallyChars Arguments



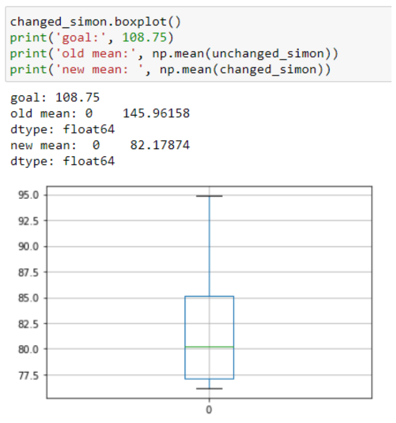
```
private static void tallyChars(BufferedReader reader, Map<Integer, Long> freq) throws IOException {
```

After this one change we saw significant change to the performance, and decided to make samples and check if we hit our goals.

4 Documentation of performance after optimization

Figure 8: Boxplot of optimized test results

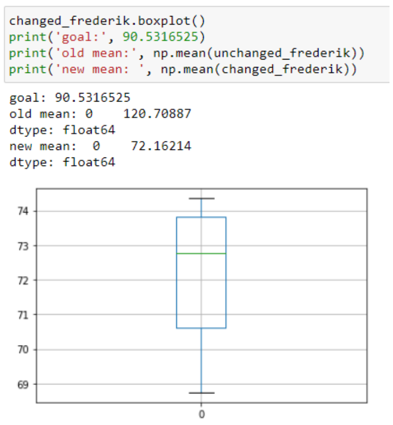
(a) Simon's tests.



```
changed_simon.boxplot()
print('goal:', 108.75)
print('old mean:', np.mean(unchanged_simon))
print('new mean: ', np.mean(changed_simon))
```

```
goal: 108.75
old mean: 0    145.96158
dtype: float64
new mean:  0    82.17874
dtype: float64
```

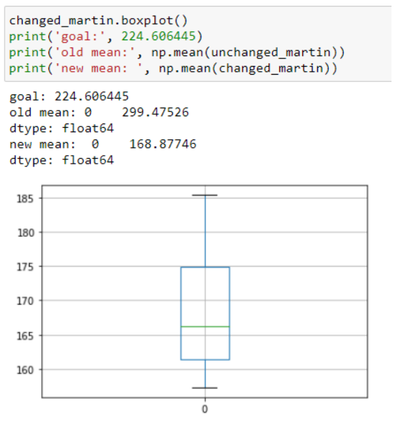
(b) Frederik's tests.



```
changed_frederik.boxplot()
print('goal:', 90.5316525)
print('old mean:', np.mean(unchanged_frederik))
print('new mean: ', np.mean(changed_frederik))
```

```
goal: 90.5316525
old mean: 0    120.70887
dtype: float64
new mean:  0    72.16214
dtype: float64
```

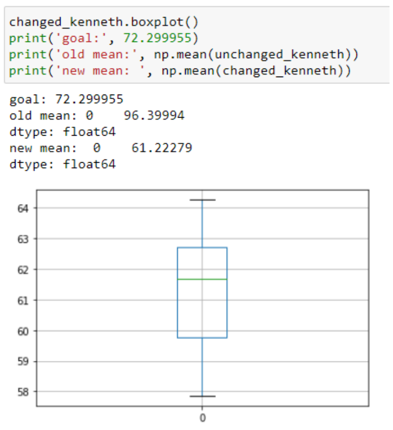
(c) Martin's tests.



```
changed_martin.boxplot()
print('goal:', 224.606445)
print('old mean:', np.mean(unchanged_martin))
print('new mean: ', np.mean(changed_martin))
```

```
goal: 224.606445
old mean: 0    299.47526
dtype: float64
new mean:  0    168.87746
dtype: float64
```

(d) Kenneth's tests.



```
changed_kenneth.boxplot()
print('goal:', 72.299955)
print('old mean:', np.mean(unchanged_kenneth))
print('new mean: ', np.mean(changed_kenneth))
```

```
goal: 72.299955
old mean: 0    96.39994
dtype: float64
new mean:  0    61.22279
dtype: float64
```

Above each of the boxplots in figure 8 we can see the old mean of the unoptimized tests and the new means for the optimized tests along with the goal means.

5 Reflection

We were told by our reviewer, that our measurement tool, the timer, was insufficient. The reviewer suggested that we use a better benchmarking method as laid out in the Sestoft Microbenchmarking report. Furthermore, we could have made a program that runs the code many times and then records the mean result instead of running the program 10 times manually. To improve upon our work, we could research an even better alternative to the BufferedReader instead of simply stopping when the goal was met.

6 Conclusion

We calculated our goal means in the section Documentation of the current performance which were set to have a 50% reduction in mean spent time for each run of the program. By using the Java Flight Recorder we could conclude that the tallyChars method was the main bottleneck of the program, meaning that this is where we could make the most impactful change. For each of our test means we more than accomplished our goal means and were even very close to the 100% improvement stated in the assignment as possible. We therefore concluded the project here.