CS 1632 – DELIVERABLE 4: Performance Testing

Matt Stropkey

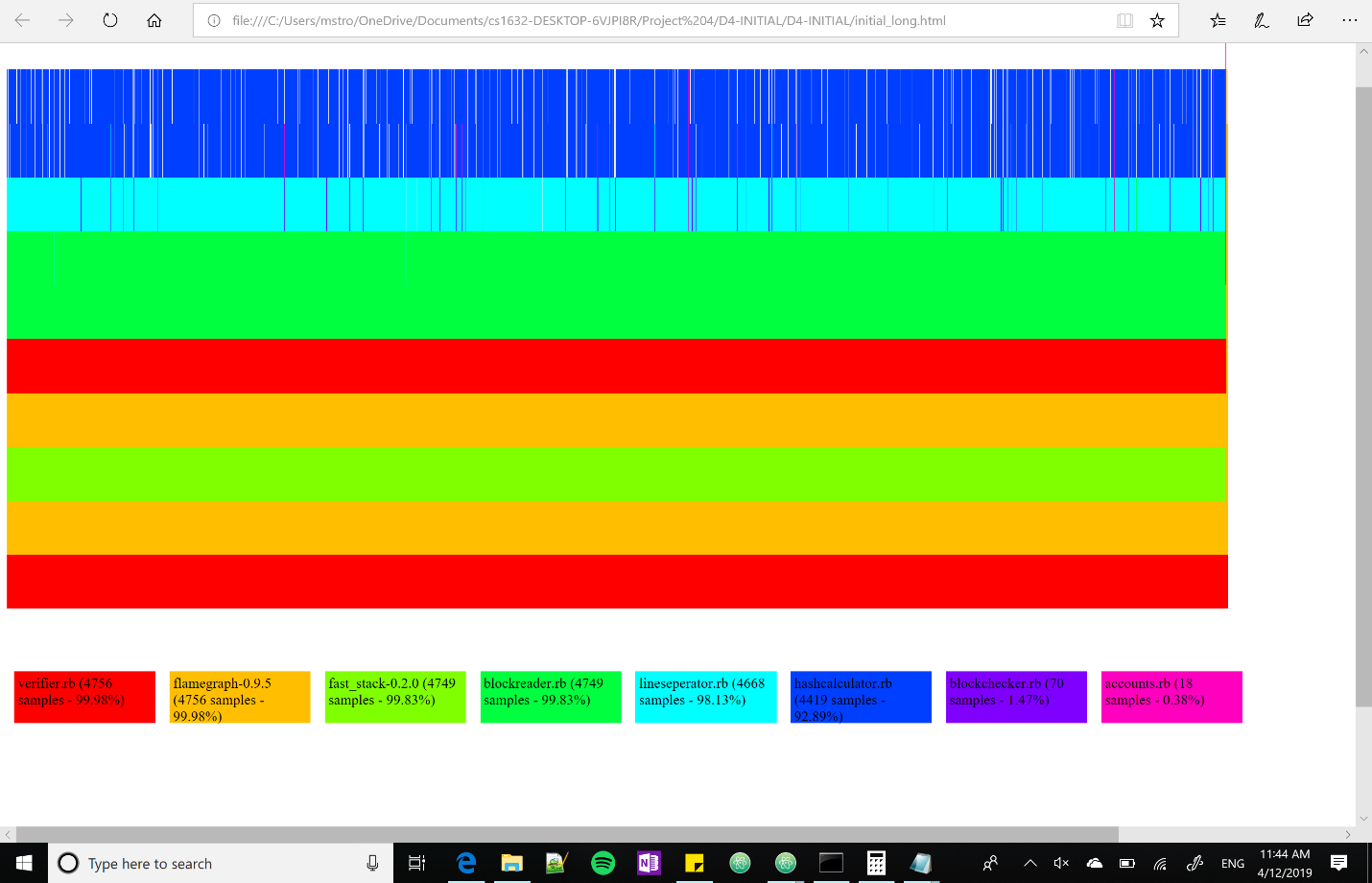
<https://github.com/mas682/D4>

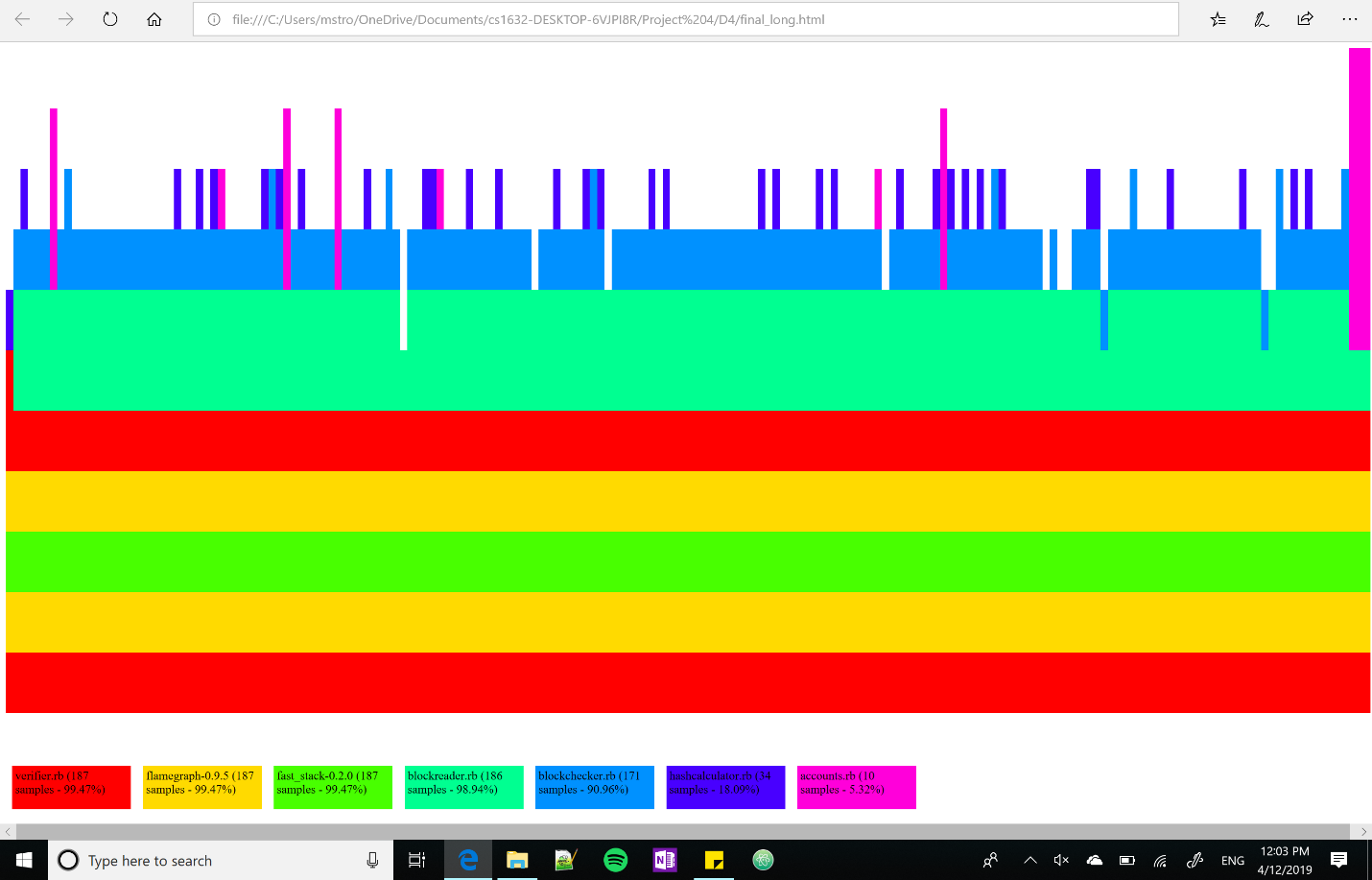
The most challenging part of this deliverable for me was thinking of the various corner cases that may be cause issues as well as writing the test cases. During this deliverable, I made sure to try to write my methods as pure as possible so that they would be easy to test, but it was challenging at times because I had a few methods that had various outputs depending on the issues within the block. This made it kind of frustrating to test just because there were different outputs that were needed for methods at different times. For example, I have a method that handles a blocks transaction and if something is wrong within those transactions, there are various outputs. I also faced some challenges with rubocop because I was using too many classes and passing them as arguments. I ended up having to condense the classes some on the final commit as a result.

Edge cases were particularly challenging just because there were so many of them. I got my program working nearly at what I thought covered every test case for the initial commit, but then I later saw there were a few cases that I did not handle. One of the corner cases that I almost missed was if a transaction has the coins in the format (1)a). Under this circumstance, my program was not causing an error because it found 1 and just ignored the a. It later noticed the hash may have been wrong, but I eventually made it handle this. There is also the case where there is fewer or more | characters in a block then expected, that was checked by parsing the line and seeing how many blocks were returned in the array. Another corner case that I almost missed was having an extra period at the end of the time stamp. Under this case, I parsed the time stamp with the ‘.’ being the key character, but it would still just return an array of size 2 with the second string not having the period.

After using rubocop, I could see that my hash methods were taking up the most CPU time, which was not too surprising. Originally, I was having the value for each character calculated every time it was encountered, which involved a lot of multiplications. I then remembered what I believe is called dynamic programming from 1501. I decided to calculate a characters hash value, without taking the mod of it, and then storing it in a table on the first time it was encountered, so that on future references to a character with the same UTF-8 value, the method would not have to do the multiplications. I then noticed I could take this even farther by loading the values of the various characters hashes into a file and then preloading them into an array so the program did not have to do any multiplication at all. This resulted in the program being significantly faster. Lastly, I noticed I could take make this even faster if instead of just looking at single characters, I looked at several in a row. There are often repeated strings in the blocks, specifically for account numbers. I then decided to store whole hash values for 6 digit account numbers when they were encountered so that the program could essentially skip over the transaction list much faster, which will contain most of the characters in the block. This also saved about an additional half second in run time.

I also just want to note here, that I changed the outputs for the final version and the initial version. The initial version only tells you there was a issue in a certain block, while the final version specifies what the issue is. I also added a flamegraph for the initial upload using 1000.txt just in case you have trouble opening the initial\_long flame graph. You also need the INIT.txt file from my github to run the program, as well as the test\_file.txt to test run the test cases. These just need to be located in the folder that the program is being ran from.

**INITIAL:** 

**FINAL:** 

|  |  |
| --- | --- |
| **INITIAL TIMES** |  |
| **Test 1** | 48.7366672 |
| **Test 2** | 48.6939346 |
| **Test 3** | 48.639965 |
| **Average** | 48.6901889 |
| **Median** | 48.6939346 |

|  |  |
| --- | --- |
| **OPTIMIZED TIMES** |  |
| **Test 1** | 1.9580336 |
| **Test 2** | 2.0109095 |
| **Test 3** | 1.9756242 |
| **Average** | 1.9815224 |
| **Median** | 1.9756242 |