P4 Submission

Nicholas Werle - U00764124

March 14, 2017

1 Preamble

The code and other files, in addition to being included with the pilot submission as required, will be available at `https://github.com/NickWer/CEG3900_P4` on Thursday, March 16th.

2 Task 1

My program works well, with no bugs or crashes that I am aware of. To test, I made the file many times larger (copy and pasted the original 240k log until it was 300MB) and it still ran well. Additionally, using a script called memusg from `https://gist.github.com/netj/526585`[1], I was able to verify that the memory consumption of my implementation did not noticeably grow with the size of the input.

Preconditions: The filenames specified are valid and properly escaped on input. The appropriate output directories are writable.

Postconditions: The provided output file (or the default one) contains a list of invalid logins

I developed this program by creating a package for parsing, which will hopefully be trivial enough to modify for the later parts of the assignment. Using streams and regex it searches each line of the log against the regex string and extracts the IP address if possible. I think the callback that I put in place will be valuable for the later portions, and what code there is will be simple to modify.

[1] This script is included in my repository as a submodule.

3 Task 2

See figures 1-4 for the screenshots of this application running. Note that they may end up in funny places throughout the document, because that's what LaTeX likes to do.

My application works well enough. One thing that could be improved is that, when clicking either processes button, the calculation is run on the UI thread and so the application hangs while it processes the log.

Building this application wasn't too terrible but also certainly came with many minor issues. Passing data between activities is surprisingly difficult if you actually want to do it the right way, and need to send more than a simple integer or string. I decided that this wasn't worth figuring out given my somewhat limited time and the simplicity of the application, so I just used a global static class to hold the Stream object I wanted to pass.

4 Task 3

The source for this program is available in the Task 3 folder. I used the following input to test my program:

```
https://www.gutenberg.org/files/38280/38280-0.txt
http://www.gutenberg.org/cache/epub/20/pg20.txt
http://www.gutenberg.org/cache/epub/8209/pg8209.txt
https://www.gutenberg.org/files/25340/25340-0.txt
https://www.gutenberg.org/files/25340/25340-0.txt
http://www.gutenberg.org/cache/epub/8388/pg8388.txt
http://www.gutenberg.org/cache/epub/22312/pg22312.txt
http://www.gutenberg.org/cache/epub/2939/pg2939.txt
http://www.gutenberg.org/cache/epub/36446/pg36446.txt
http://www.gutenberg.org/cache/epub/5891/pg5891.txt
```

Which generates the following output (n = 10):

```
the - 57874
of - 31890
and - 29439
to - 19994
a - 16415
in - 16365
is - 9199
i - 8946
```

`that - 8150`

Preconditions: n must be ¿= 0, the input files must use the English alphabet.

Postconditions: A list of words and the tally of their frequency across ALL documents in the input file is written out.

My solution works well, as best I can tell. It does not attempt to strip out html or anything similar, and so it generally assumes that input is plaintext. Additionally it will only be compatible with the standard 26/52 character English alphabet. As I understand it this solution works with ASCII as well as other UTF encodings, though I have not tested this.

The experience of developing this was overall not very different from Task 1, though the optimization phase did strain my familiarity with the Java regex/Pattern API, and I've never used the reduction capabilities of streams either. Once again I took the approach of making a general enough solution that I will be able to re-purpose my package for the android application, however this time I took a more straightforward approach by returning the list and allowing the package user to manipulate it as they please.

One thing I found during development is that it is considerably faster to use regex to match the words themselves (my current solution), rather than their delimiters (my initial solution). While this arguable produces less true results (e.g. hyphenated words will not be treated thusly), it simplifies the sanitation process considerably (lowercasing, removing trailing punctuation, etc.) and avoids multiple, costly passes over the dataset as different map and filter operations are applied for sanitization. You could, of course, apply one very good map or filter that does the sanitation in a single step, but by grabbing the words directly, you can avoid even that single extra pass, speeding up the output considerably.

5 Task 4

Screenshots of my application running are shown in figures 5-8. My sample input is the same as before. The output is as shown in figure 8.

This android application went much faster than the previous. IT seems to work correctly though I have not tested it extensively. One issue I did encounter is that, because Task 3 requires network IO, I found myself forced to background the task. This was actually much easier than I anticipated, thankfully. Other than that, the process was relatively quick and painless compared to most of my previous android experiences. Perhaps, just maybe, I'm getting the hang of it.

6 Task 5

Figures 9-12 show my revised Task 4 submission screenshots.

I found this to be extremely easy to do because my parser used streams right from the beginning (as soon as it loads the book list), and in order to make my task 4 submission work, I already had to remove the processing from the main thread onto a secondary one. At that point, it's basically just a matter of throwing is a .parallel() call where it makes sense to do so. Initially I took your instructions of "using as much concurrency as possible", however, not surprisingly, I found that repeatedly parallelizing the stream actually hindered performance. Once I changed it such that it was only parallelized *per book* (as opposed to per line, at the collection phase, etc.), I saw the expected performance improvements.

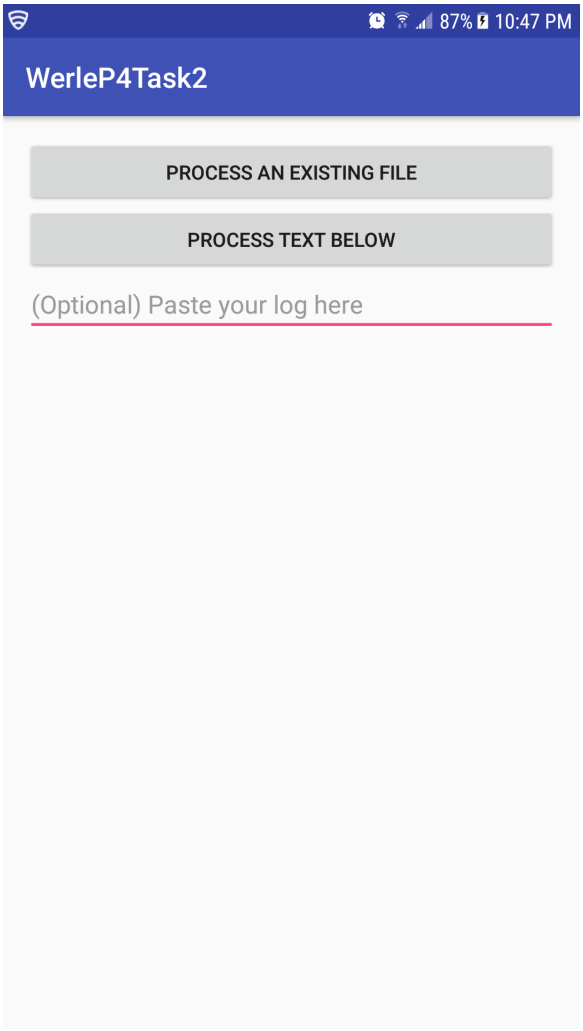

Figure 1: Home screen

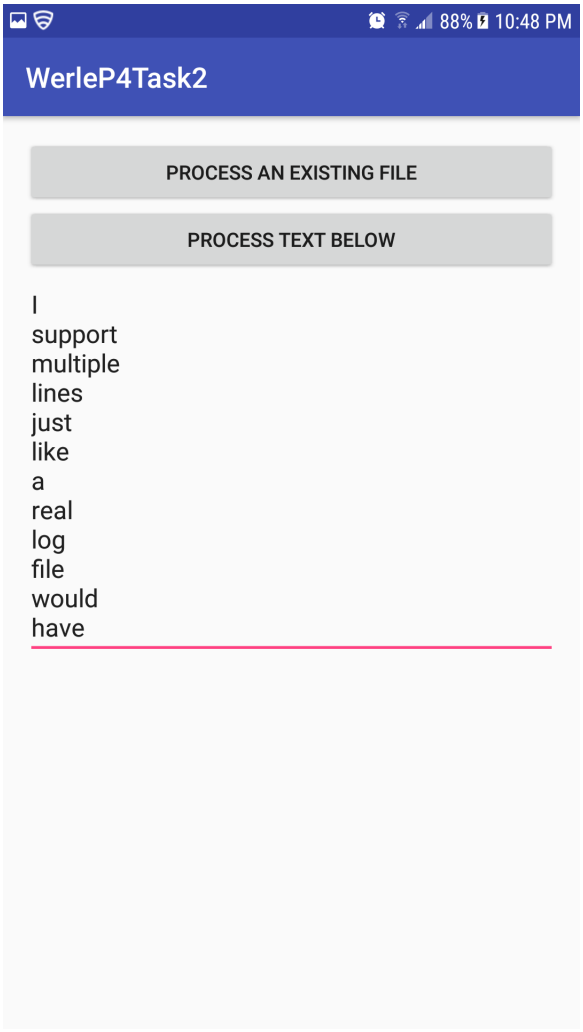

Figure 2: Some text in the log textbox (It is multiline)

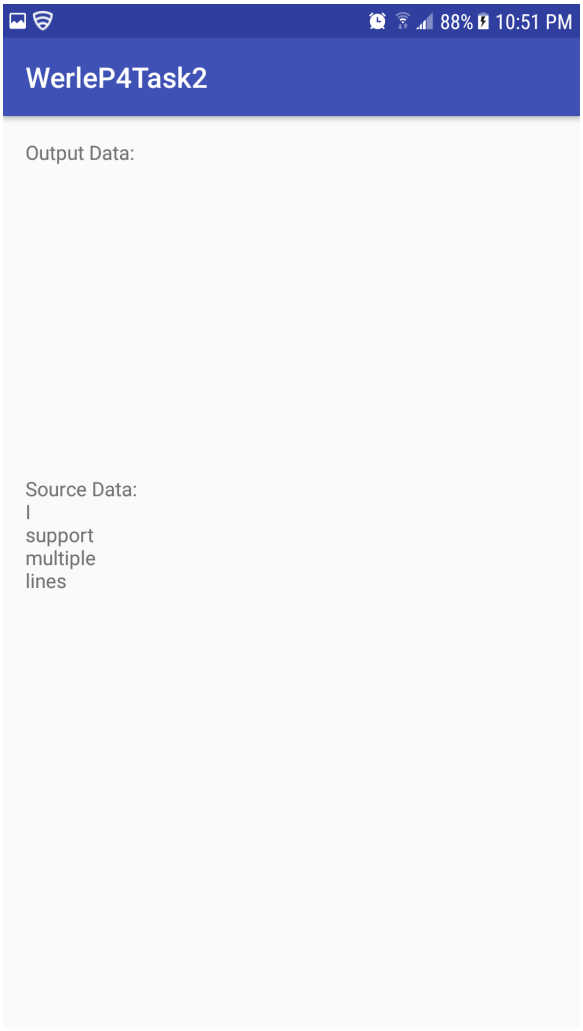

Figure 3: Example output from a nonsense input

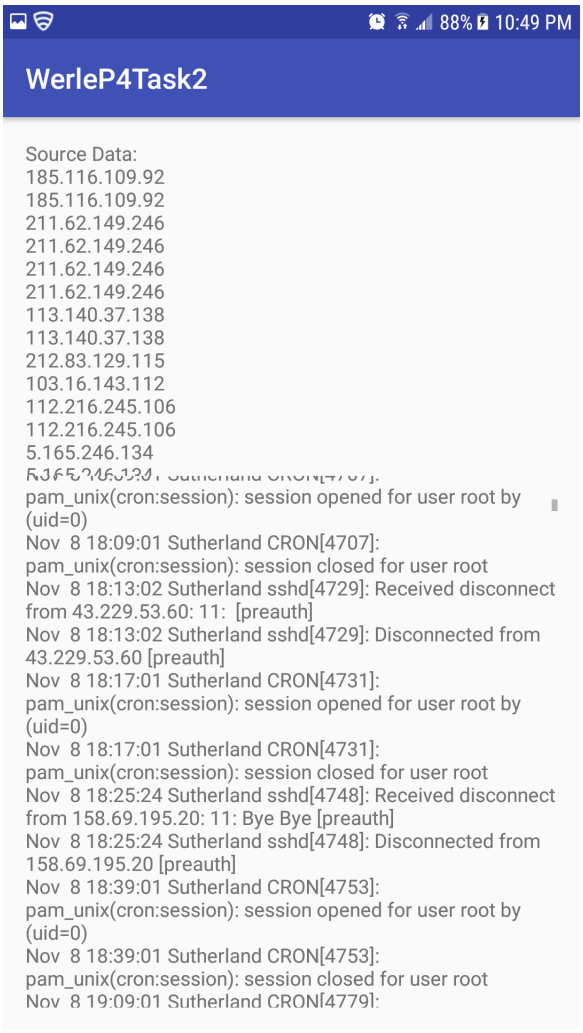

Figure 4: Example output from the provided input. You can kind of see that the lower portion of text is scrollable (the scrollbar is visible), but it's hard to capture that in a screenshot.

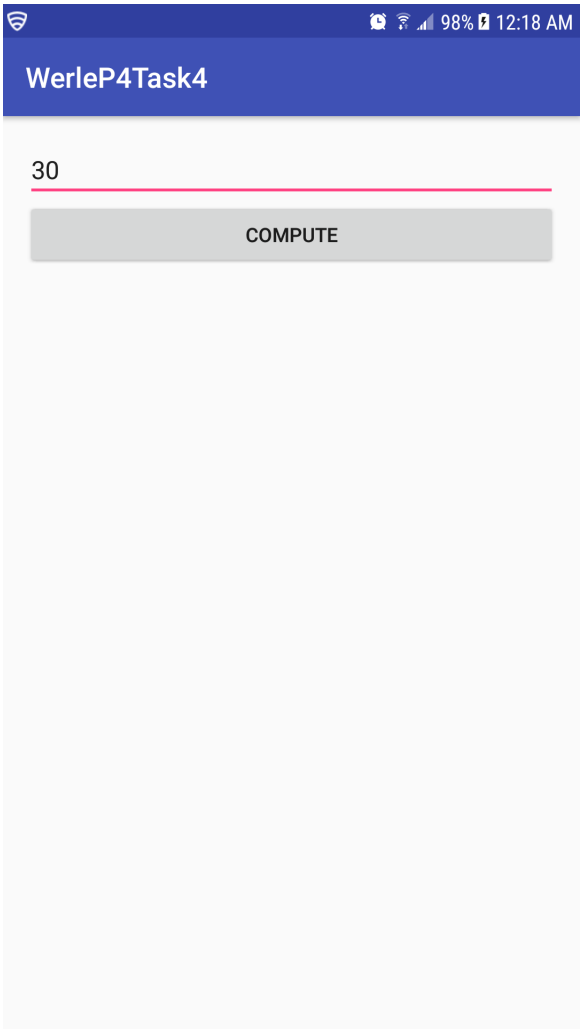

Figure 5: Home screen

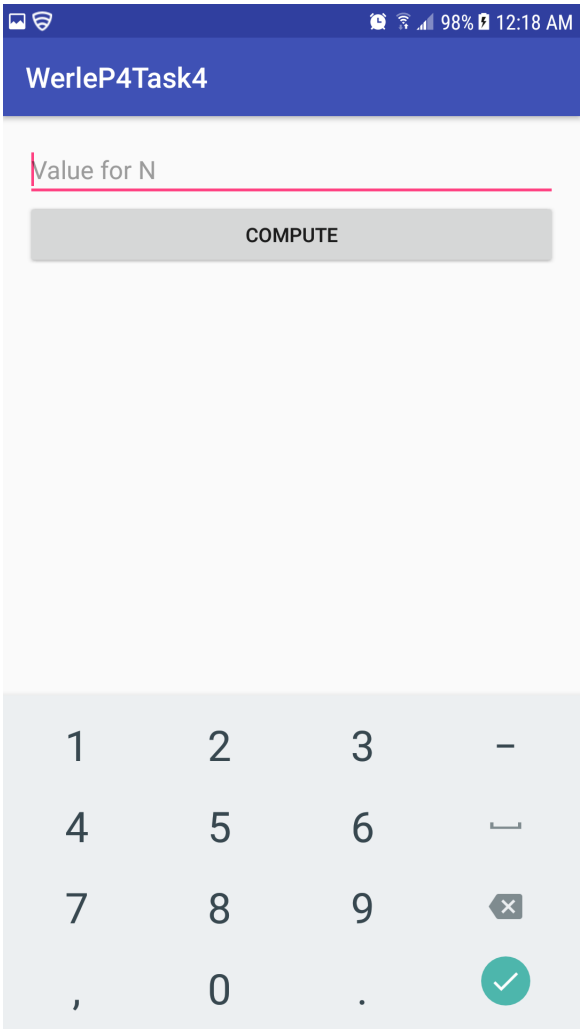

Figure 6: Showing the placeholder and number input

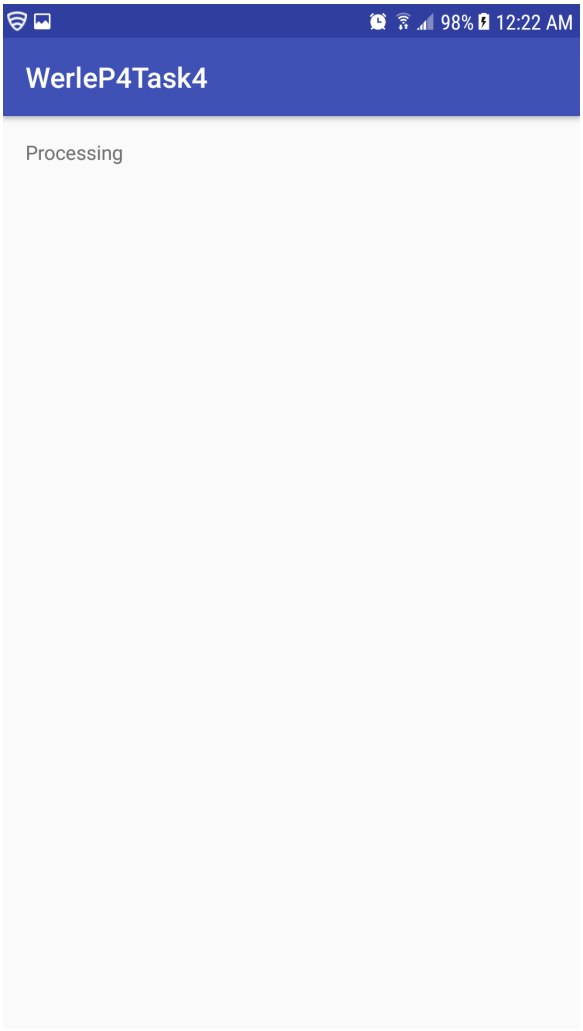

Figure 7: Simple information displayed while the user waits

Figure 8: The result, which on my phone took 20+ seconds to appear.

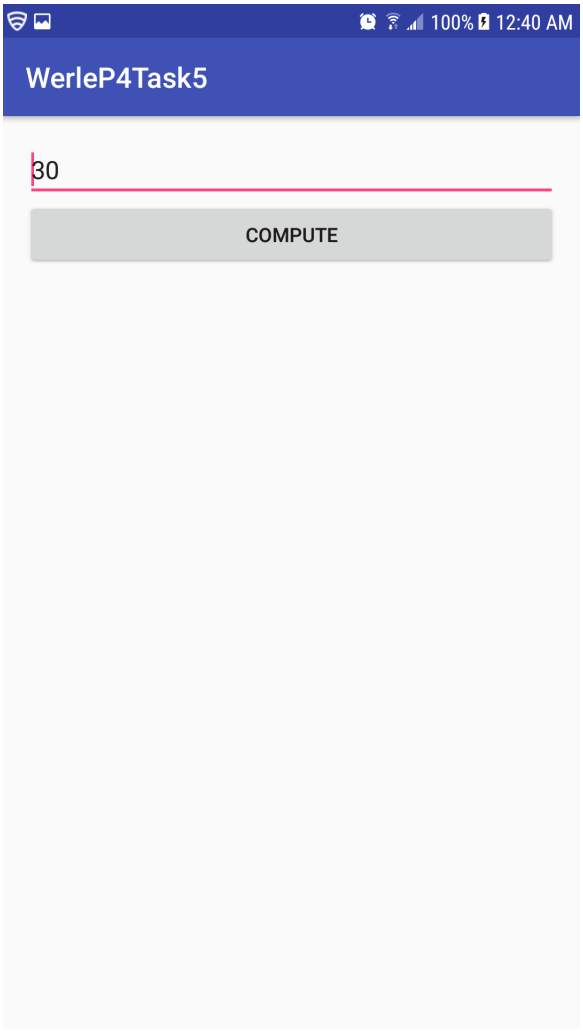

Figure 9: Task 5: Home screen

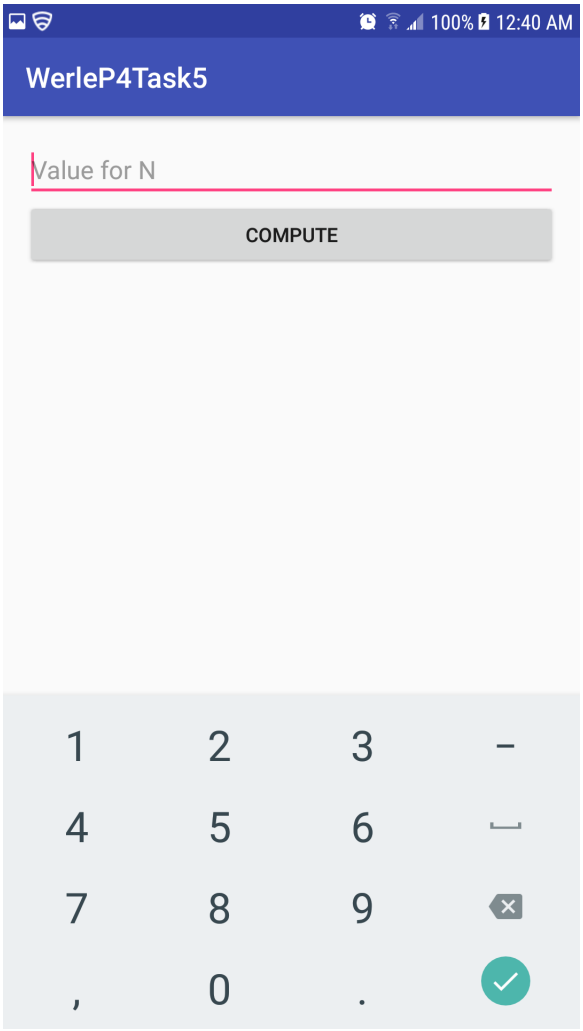

Figure 10: Task 5: Showing the placeholder and number input

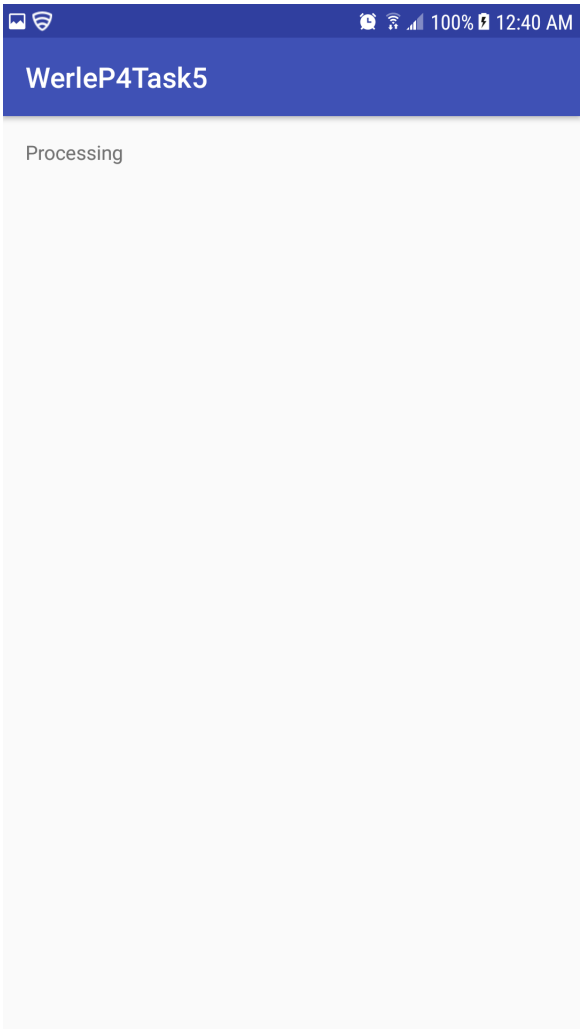

Figure 11: Task 5: Simple information displayed while the user waits

Figure 12: Task 5: The result, which on my phone took about 10-15 seconds to appear (5-10 second improvement!).