ufo asssigment 3

April 2021

1 Introduction

2 Letter frequencies

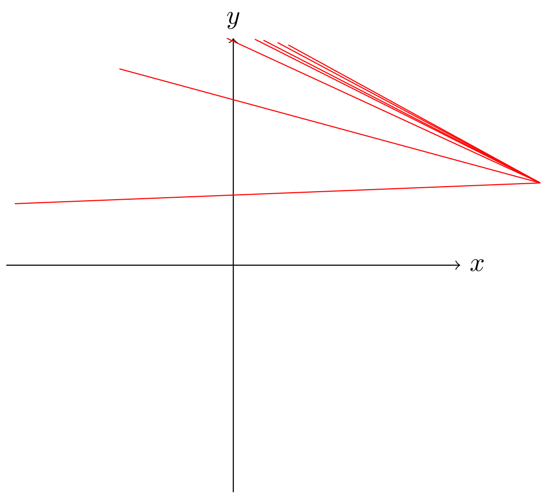
The intention of the program is to read the file FoundationSeries.txt. We read each line using the tallyChars method which fills our map with characters and fills the hashmap with entries for their keys. The part we would want to optimze are the tallyChars method, as well as how we use the file reader and how we track our time properly.

3 Documentation of the current performance

For the specification it might be adviseable to follow examples given in this document: https://datsoftlyngby.github.io/soft2021spring/resources/ee799a67-SestoftMicrobenchmarking.pdf For the Timer class it also will be relevant to view the following page: https://datsoftlyngby.github.io/soft2021spring/resources/98dc3b91-02Slides.pdf

The original performance is hindered for example by the map storing integers instead of characters. Using a readline also slows the performance compared to a buffered reader using readline().

the below graph represent results from running 100 iterations where we only use the original tallychars method. This graph only represent some handpicked



values out of the 100 for this graph.

4 Explanation of bottleneck(s)

One bottleneck might be tallyChars method using a try/catch instead of a plain null check, that should be faster. Another potential bottleneck might be the sorted map consisting of character keys and Long values. It calls many methods on the Hashmap, It starts with calling EntrySet which may traverse the entire hashmap as a set only for then to convert the set into a stream to call a sort method on it. This sorted methodcall has to sort the entire stream of the entire entrySet. The sorted method uses collections.reverse for getting the entry comparing values. It then sorts them back to map again using getkey and getvalue creating a new instance of a hashmap for each? This should be several individual steps perharps. It just seems to be a lot of things occuring at once.

Other bottleneck might be plainly reader.read() where it might be worth to replace with a buffered reader instead.

The major bottleneck in the old code was the tallychar method doing just read() instead of readline() and not using a bufferedreader. Also its not storing the values in memory and hence they perform slower.

5 A hypothesis of what causes the problem

The file is very large to read and because it by default is not split into certain chunks sizes it might load too much at the same time.

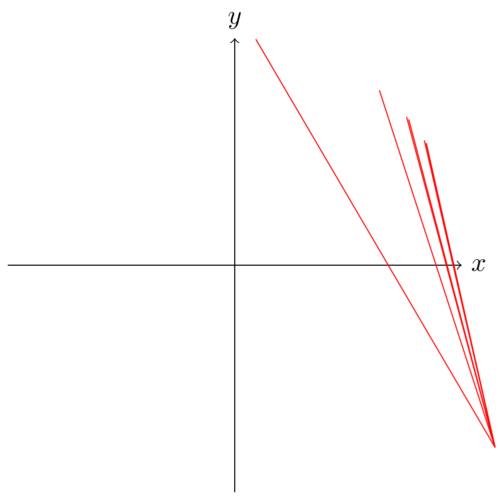
6 A changed program with better performance

Tobias also ran a profiler with his old laptop that claimed the reader.read() method took the longest.

The changes are the following: We think we have to replace the Readfile with BufferedReader for better performance. We also forced it to allocating into memory by calling toCharArray(). we also replaced read() with readline() which is a string that then gets converted to a char array which acts faster by being in memory.

Here are results from running on my windows 10 laptop: Run : 95.93350000000001 ms Run : 49.4078 ms Run : 52.5739 ms Run : 43.783500000000004 ms Run : 36.4041 ms Run : 37.2189 ms Run : 39.2093 ms Run : 34.6847 ms Run : 36.0696 ms Run : 35.662600000000005 ms Run : 34.7796 ms Run : 33.605000000000004 ms Run : 35.174200000000006 ms Run : 35.4606 ms Run : 36.1589 ms Run : 34.5157 ms Run : 39.966 ms Run : 35.9639 ms Run : 35.4863 ms Run : 35.5467 ms Run : 43.255300000000005 ms Run : 37.3739 ms Run : 35.2592 ms Run : 37.2166 ms Run : 35.629 ms Run : 40.5069 ms Run : 35.4315 ms Run : 35.6845 ms Run : 41.5454 ms Run : 34.3655 ms Run : 35.9738 ms Run : 50.3872 ms Run : 63.1061 ms Run : 60.0357 ms Run : 56.588 ms Run : 57.5557 ms Run : 56.934400000000004 ms Run : 53.337700000000005 ms Run : 53.1362 ms Run : 38.9221 ms Run : 35.9251 ms Run : 34.3752 ms Run : 35.488899999999994 ms Run : 42.2014 ms Run : 37.3603 ms Run : 35.366 ms Run : 36.467600000000004 ms Run : 34.5634 ms Run : 35.047200000000004 ms Run : 35.6693 ms Run : 37.2082 ms Run : 37.7107 ms Run : 37.3034 ms Run : 38.217800000000004 ms Run : 37.4557 ms Run : 37.583400000000005 ms Run : 38.7633 ms Run : 39.2085 ms Run : 39.4091 ms Run : 50.7025 ms Run : 47.5415 ms Run : 53.708800000000004 ms Run : 46.8354 ms Run : 50.054 ms Run : 48.194300000000005 ms Run : 47.4686 ms Run : 41.3747 ms Run : 41.3834 ms Run : 46.065000000000005 ms Run : 38.1371 ms Run : 40.0125 ms Run : 39.1668 ms Run : 40.499300000000005 ms Run : 39.4803 ms Run : 39.1611 ms Run : 37.549800000000005 ms Run : 39.2941 ms Run : 36.780100000000004 ms Run : 38.3265 ms Run : 38.919599999999996 ms Run : 40.8446 ms Run : 38.9751 ms Run : 37.502300000000005 ms Run : 40.1366 ms Run : 38.142500000000005 ms Run : 42.475899999999996 ms Run : 40.2777 ms Run : 38.1784 ms Run : 40.0436 ms Run : 39.2136 ms Run : 40.7185 ms Run : 39.1745 ms Run : 42.378699999999995 ms Run : 45.237399999999994 ms Run : 38.8374 ms Run : 40.4676 ms Run : 40.4722 ms Run : 43.1743 ms Run : 46.921200000000006 ms Run : 48.1896 ms Average optimized: 41.471911 ms Average unoptimized: 106.581477 ms

here is a representation of running it once from my laptop with again 100 iterations where some values are picked but with the new main method instead.



7 Documentation of the new performance

The largest performance boost was gained through switching to a buffered reader using readline() instead of just a .read() from a readfile. Another significant boost comes from forcing memory usage instead of normal storing. This can quickly turn memory intensive with large chunks but by reading a single line at a time the memory option also helps at turning it faster. But buffered reader adaption using readline() instead of read() should be most singificant change.

8 Written in LATEX

This was written with overleaf.com online editor.