Statistical and Empirical Analyses of Boggle

You need to report in your Analysis file information about which Lexicon implementation is fastest[1]. You should compare all the lexicons and report on their relative times — you shouldn't simply say "this one is fastest". You should have at least three lexicons to test and four if you do the extra credit. You should explain the methods you used to determine this and report on experiments you run, giving times.

LexiconBenchmark with different Lexicon benchmarks

	Iter time (s)	Word time (s)	Pref time (s)	Words	Size
SimpleLexicon	0.011	0.008	0.025	80612	16466
BinarySearchLexicon	0.008	0.001	0.014	80612	16466
TrieLexicon	0.120	0.001	0.019	80612	16466
CompressedTrieLexicon	0.103	0.000	0.026	80612	16466

The results in the table above were for a 4 x 4 Boggle board with 10 trials. With repeated trials, BinarySearchLexicon is the fastest, likely owing to the more binary search algorithm that it uses. Also note that the list in BinarySearchLexicon is sorted into ascending order per the method's specifications.

In terms of speed, the variations in the structure of each class accounts for differences in timings. Coming in a close second is SimpleLexicon, followed by a distant CompressedTrieLexicon and TrieLexicon. Note some savings achieved by CompressedTrieLexicon's with its compressed nodes.

---

[1] We also provide a benchmarking class **LexiconBenchmark** that facilitates evaluating the efficiency of different implementations as well as correctness. Confidence in an implementation's correctness is increased if it returns the same results as other implementations.

BoggleStats with different Lexicon benchmarks

	Average Time (s)
LexiconFirstAutoPlayer SimpleLexicon	0.537
BoardFirstAutoPlayer SimpleLexicon	0.084
LexiconFirstAutoPlayer BinarySearchLexicon	0.507
BoardFirstAutoPlayer BinarySearchLexicon	0.069
LexiconFirstAutoPlayer TrieLexicon	1.011
BoardFirstAutoPlayer TrieLexicon	0.042
LexiconFirstAutoPlayer CompressedTrieLexicon	0.729
BoardFirstAutoPlayer CompressedTrieLexicon	0.052

You must write code to play lots of auto-games, not games in which humans play, but games in which all words are found on thousands of boards — see BoggleStats for a starting point. You must provide a board that scores the highest of all the 4×4 and 5×5 boards you test in running at least 10,000 auto-games. and preferably 50,000 games. Report on how many seconds it takes your code to run for 1,000 games; for 10,000 games (or predict that if you can't run that many); and predict/justify on how much time it would take your computer and implementation to simulate both 100,000 games and one million games. When doing the experiments be sure to set the random-number generator's seed, currently done already in BoggleStats and described above. If you can't run 10,000 auto-games, indicate the maximum number you can run.

BoggleStats for BoardFirstAutoPlayer with different Lexicons

	Count	Max	Time (s)
SimpleLexicon	1000	612	2.733
BinarySearchLexicon	1000	612	1.843
TrieLexicon	1000	612	1.403
CompressedTrieLexicon	1000	675	1.227
SimpleLexicon	10000	753	21.408
BinarySearchLexicon	10000	753	16.024
TrieLexicon	10000	753	11.202
CompressedTrieLexicon	10000	881	11.403
SimpleLexicon	50000	779	103.873
BinarySearchLexicon	50000	779	87.329
TrieLexicon	50000	779	55.237
CompressedTrieLexicon	50000	937	58.283

We witness a linear relationship between the time taken versus the number of trials conducted. Observe, a 10x increase in count from 1000 to 10,000 results in a corresponding 10x increase in times. A similar 5x increase in count from 10,000 to 50,000 also displays the expected 5x increase in time. Also note the slightly inflated max figures that CompressedTrieLexicon yields in comparison to the other Lexicon methods.

As such, to use 100,000 games as an example, it would take approximately — in seconds — 214, 160, 112 and 114 for SimpleLexicon, BinarySearchLexicon, TrieLexicon and CompressedTrieLexicon.