Miguel Ricardo Anderson (netId: mra21)

CompSci 201 – Assignment Analysis for Autocomplete

04/08/2016

Part 1

What is the order of growth (big-Oh) of the number of compares (in the worst case) that each of the operations in the Autocomplete data type make, as a function of the number of terms N, the number of matching terms M, and k, the number of matches returned by topKMatches for BinarySearchAutocomplete?

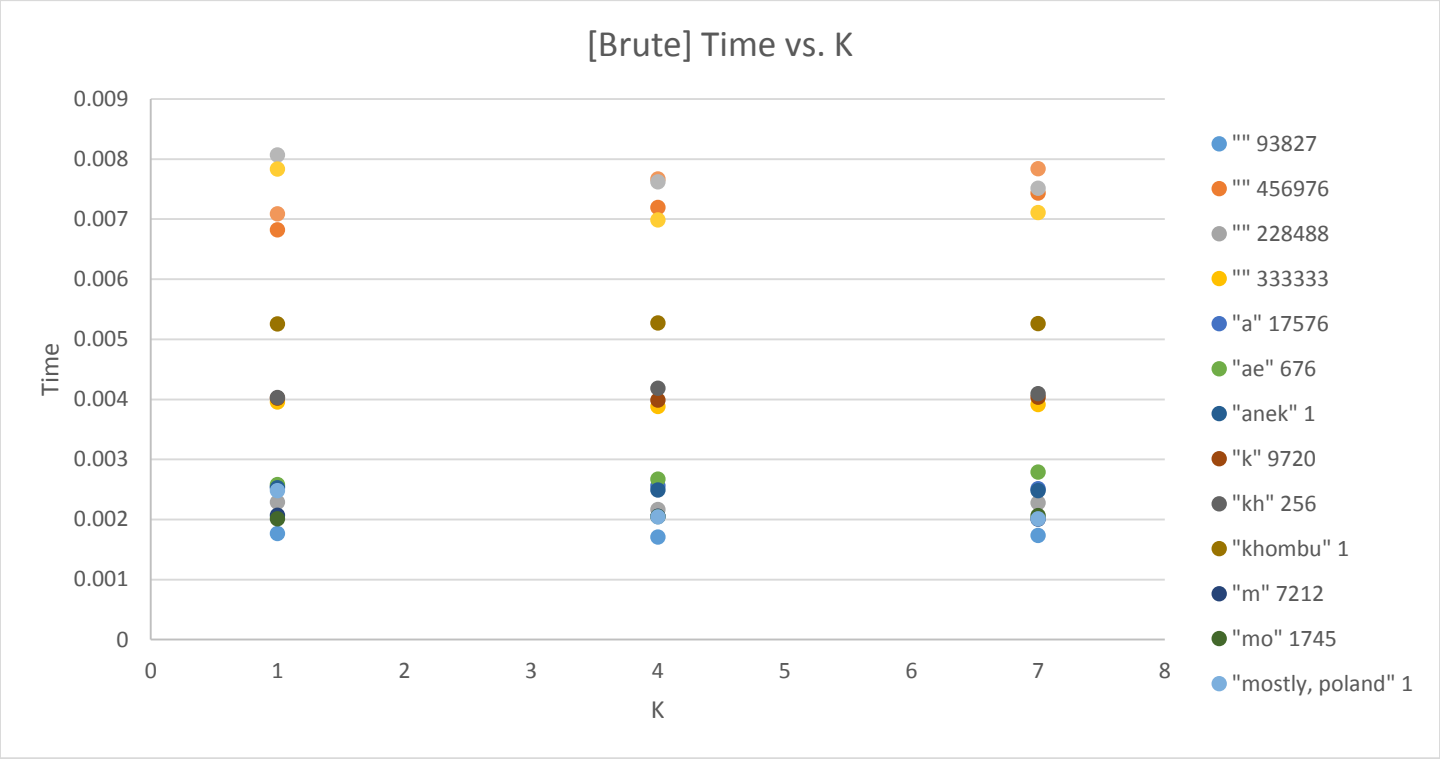
Call	Big-Oh time	Compare calls
firstIndexOf	1+log(N)	1+log(N)
lastIndexOf	1+log(N)	1+log(N)
Arrays.copyOfRange	M	0
Arrays.sort	M*log(M)	M*log(M)
Fill final array	K	0

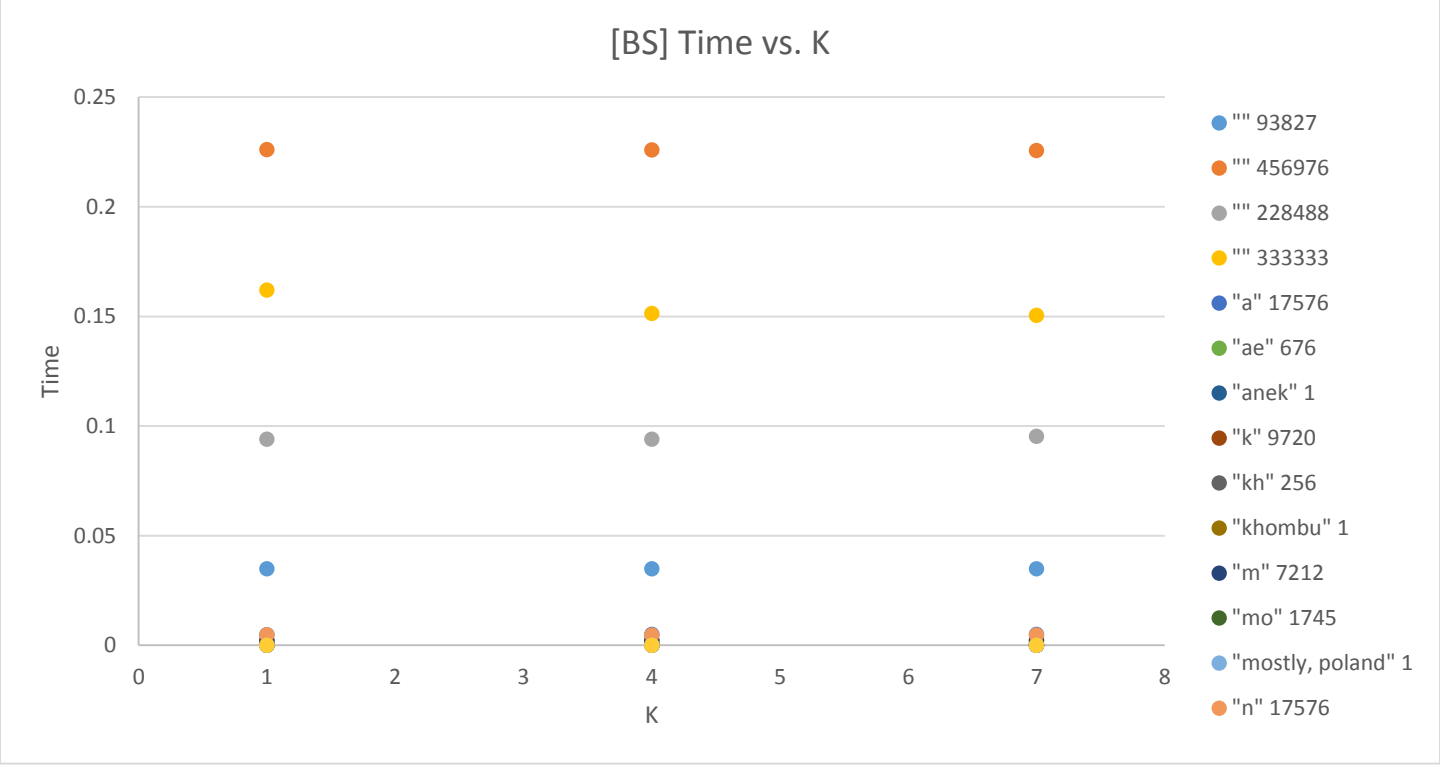
This table leads to these results. The number of compare calls should therefore be **(2+2*log(N)+M*log(M))** in the worst case

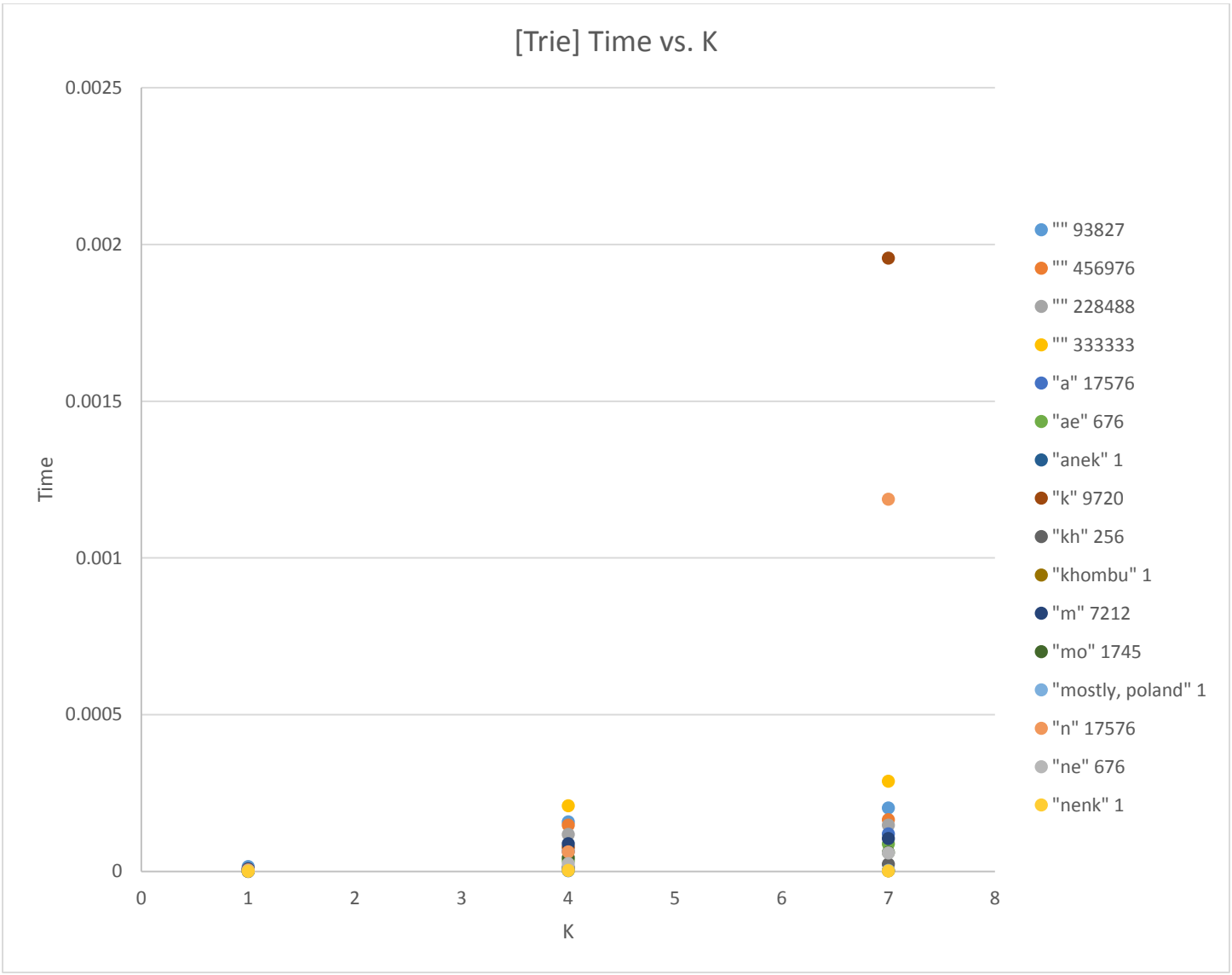
Part 2

How does the runtime of topKMatches() vary with k, assuming a fixed prefix and set of terms?

The only code that explicitly depends on k that has significant time considerations is in TrieAutocomplete as k will limit greatly the extent to which the while loop continues to find enough children to fill k largely weighted spots. The other two methods should show no strong relationships. Graphs are provided below. The legends refer to the prefix and then the number of occurrences of that prefix in the text file.






[Trie] Time vs. K
0.0025
0.002
0.0015
0.001
0.0005
0
Time
0
1
2
3
4
5
6
7
8
K
"" 93827
"" 456976
"" 228488
"" 333333
"a" 17576
"ae" 676
"anek" 1
"k" 9720
"kh" 256
"khombu" 1
"m" 7212
"mo" 1745
"mostly, poland" 1
"n" 17576
"ne" 676
"nenk" 1

Part 3

Look at the methods topMatch and topKMatches in BruteAutocomplete and BinarySearchAutocomplete and compare both their theoretical and empirical runtimes. Is BinarySearchAutocomplete always guaranteed to perform better than BruteAutocomplete?

BinarySearchAutocomplete is not always guaranteed to do better. BruteAutocomplete is guaranteed to visit every value in Terms which leads to slow runtimes. BinarySearchAutocomplete is meant to bypass this exhaustive search but does so by sorting twice and then running a search algorithm that has 1+log(N) compares. With initialization, it is possible for BruteAutocomplete to do better yet usually this is not the case.

Surprisingly my code does not reflect this.

Row Labels	Average of Time
BinarySearchAutocomplete	**0.028008547**
topKMatches()	0.026233293
topMatch()	0.033334309
BruteAutocomplete	**0.00362217**
topKMatches()	0.00396032
topMatch()	0.002607717

BinarySearchAutocomplete is slower. I suspect the functions firstIndexOf and lastIndexOf to be the culprits but I am not sure.

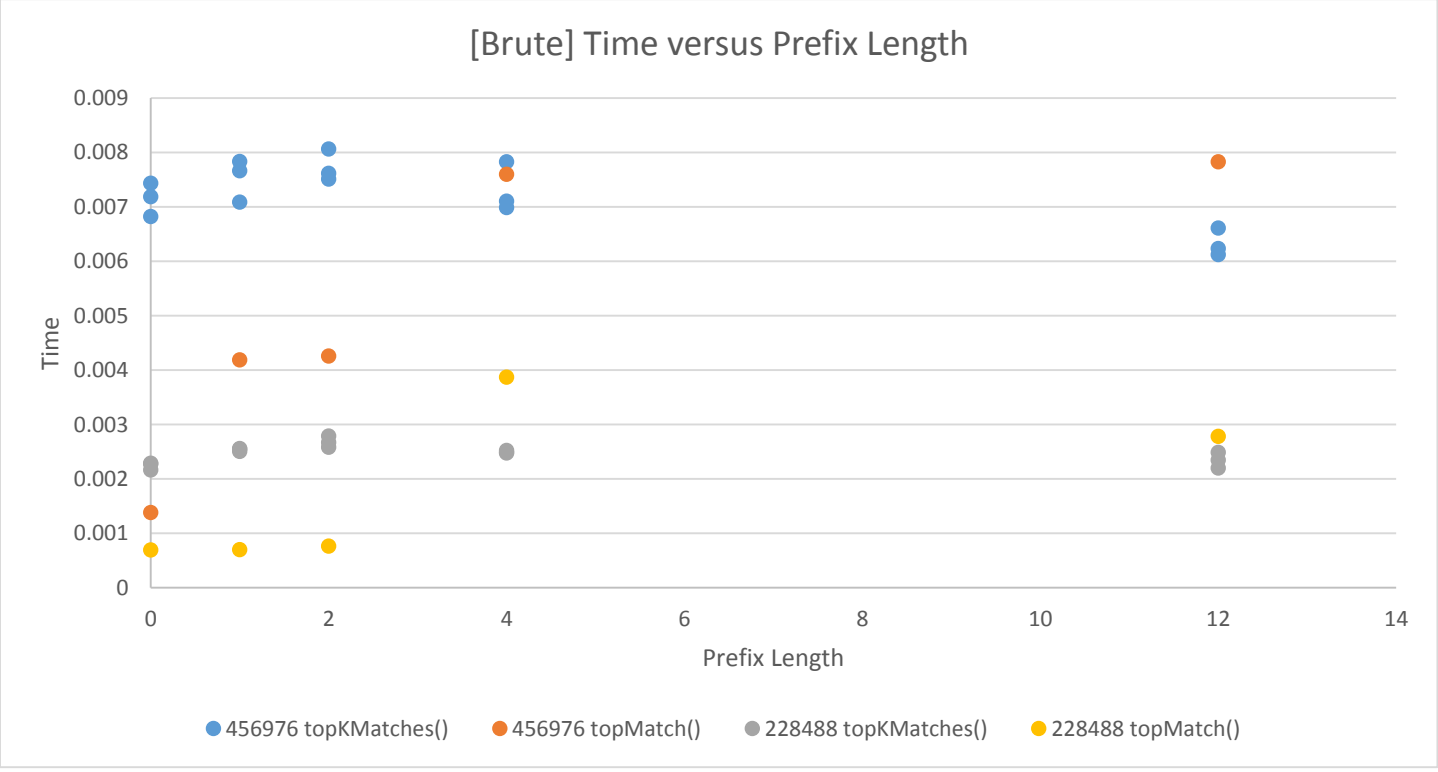
Part 4

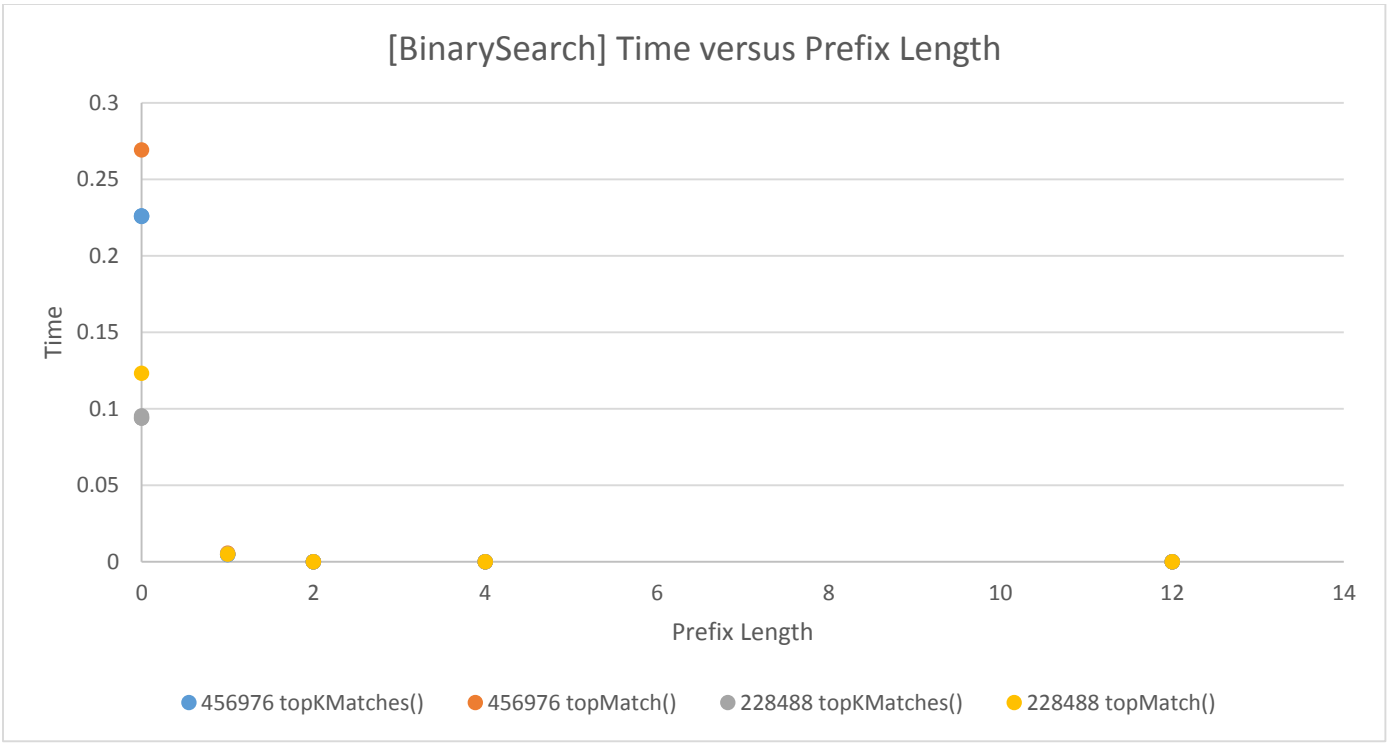
For all three of the Autocompletor implementations, how does increasing the size of the source and increasing the size of the prefix argument affect the runtime of topMatch and topKMatches?

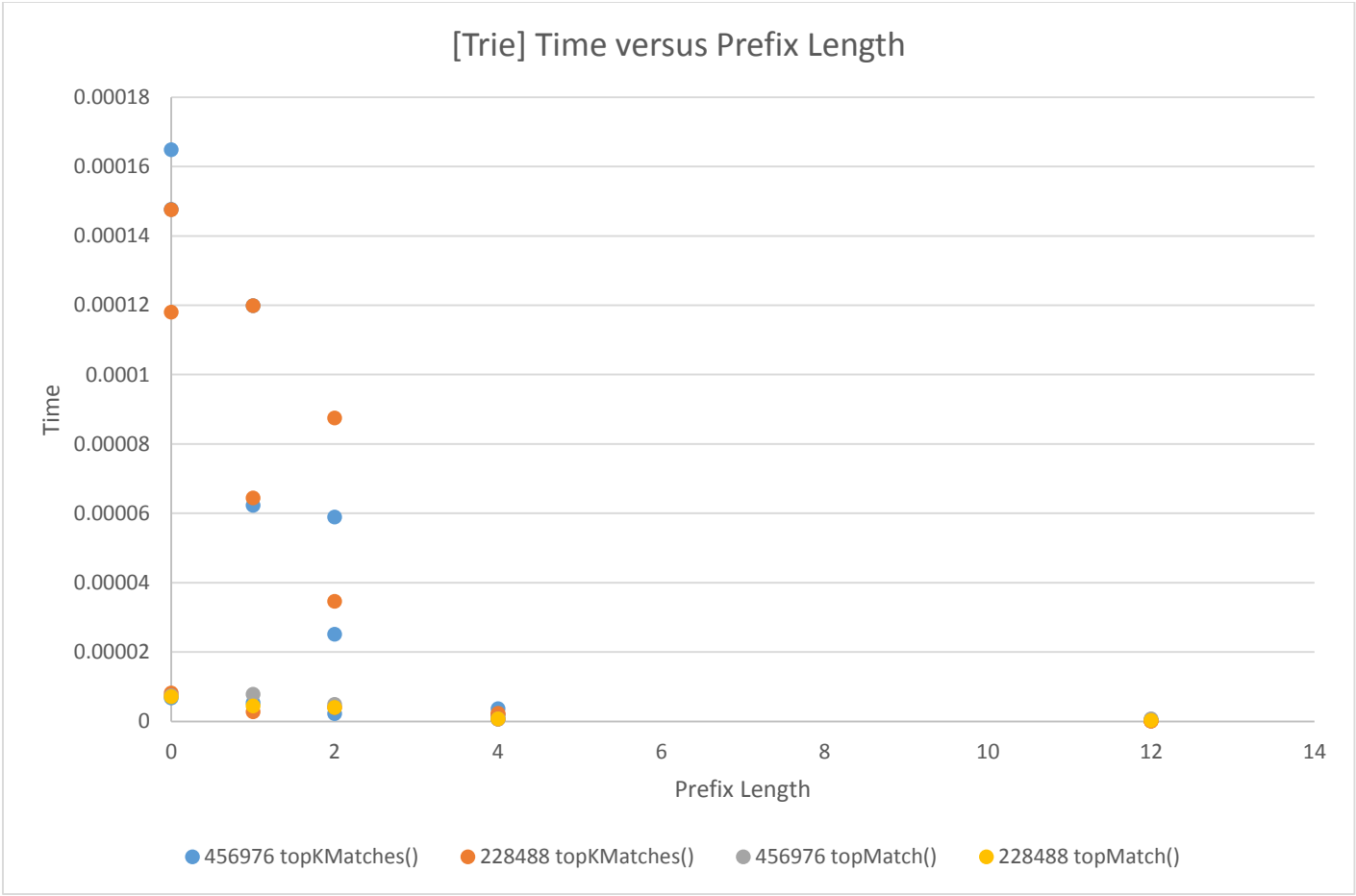
Because of the inverse relationship between prefixLength and the number of occurrences in a text file the answer will be the inverse of the answer for Part 1 of this analysis (now for all methods).

Method	Dependence on M (therefore 1/prefixLength)	Why?
BruteAutocomplete	0	Cycles through all terms
BinarySearchAutocomplete	M*log(M)	Arrays.sort
TrieAutocomplete	M (more complex than this but good approximation)	The more children the longer this code will run to find large enough weighted terms or simply exhaust all children

The graphs below show these results. The legend highlights the length of the input text file (smaller for fourletterwordshalf) and the relevant function.




[BinarySearch] Time versus Prefix Length
Time
0.3
0.25
0.2
0.15
0.1
0.05
0
0
2
4
6
8
10
12
14
Prefix Length
456976 topKMatches()
456976 topMatch()
228488 topKMatches()
228488 topMatch()


[Trie] Time versus Prefix Length
Time
0.00018
0.00016
0.00014
0.00012
0.0001
0.00008
0.00006
0.00004
0.00002
0
0
2
4
6
8
10
12
14
Prefix Length
456976 topKMatches()
228488 topKMatches()
456976 topMatch()
228488 topMatch()