COMP 3270 FALL 2018

Programming Project: Autocomplete

Name: Zachary Parker   Date Submitted: 11-26-18

1. **Pseudocode**: Understand the strategy provided for *TrieAutoComplete*. State the algorithm for the functions precisely using numbered steps that follow the pseudocode conventions that we use. Provide an approximate efficiency analysis by filling the table given below, for your algorithm.

Add

- Pseudocode:

```
Add(word: string, weight: double)
1       if word is null
2               throw an exception
3       if weight < 0
4               throw an exception
5       Node curr = root of tree
6       Node nextNode = null
7       char index = '-'
8       for i to word.length
9               index = the character in word at index i
10              nextNode = the child of curr with character index
11              if nextNode is null
12                      instantiate nextNode with character index,
                        parent curr, and given weight
13                      add new node to children map
14              if curr's subtree max weight < weight
15                      curr's subtree max weight = weight
16              point curr to nextNode
17      set curr's word value
19      set curr's weight
20      set that curr is a word
```

- Complexity analysis:

Step #	Complexity stated as O(_)
1	O(1)
2	O(1)
3	O(1)
4	O(1)
5	O(1)
6	O(1)
7	O(1)
8	O(n)
9	O(n)
10	O(n)
11	O(n)
12	O(n)
13	O(n)
14	O(n)
15	O(n)
16	O(n)
17	O(1)
18	O(1)
19	O(1)

Complexity of the algorithm = O(n)

topMatch

- Pseudocode:

```
topMatch(prefix: string)
    if prefix is null
        throw an exception
    nodeQueue = new priority queue of Nodes using a
    reverse weight comparator
    Node curr = root of tree
    char index = '-'
    boolean wordExists = true
    string empty = ""
    String topMatch = ""
    for i to prefix.length
        index = the character in the word at index i
        if a child of curr contains the value of index
            point curr to that child
        else
            wordExists = false
            break
    if (wordExists is false)
        return empty
    while (curr is a word and curr's weight < curr's
    subtree max weight)
        for every node in curr's children
            add that node to nodeQueue
        point curr to the head of queue and remove
        element from queue
    topMatch = curr's word value
    return topMatch
```

- Complexity analysis:

Step #	Complexity stated as O(_)
1	O(1)
2	O(1)
3	O(1)
4	O(1)
5	O(1)
6	O(1)
7	O(1)
8	O(1)
9	O(n)
10	O(n)
11	O(n)
12	O(n)
13	O(n)
14	O(n)
15	O(n)
16	O(1)
17	O(1)
18	O(n)
19	$O(n^2)$
20	$O(n^2)$
21	O(n)
22	O(1)
23	O(1)

Complexity of the algorithm = $O(n^2)$

topMatches

Pseudocode:

```
topMatches(prefix: string, k: integer)
    if prefix is null
        throw an exception
    Node curr = root of tree
    char index = '-'
    boolean matchExists = true
    nodeQueue = new priority queue of Nodes using a
    reverse weight comparator
    emptyList = new ArrayList of strings
    topMatches = new ArrayList of Nodes
    topMatchesString = new ArrayList of strings
    if k = 0
        return emptyList
    for i to prefix.length
        index = the character in the word at index i
        point curr to its child with the value of index if
        possible
        if curr is null
            return emptyList
    if curr is not null
        add curr to nodeQueue
    while nodeQueue is not empty
        point curr to head of nodeQueue and remove that
        node from queue
        for every node in curr's children
            add that node to nodeQueue
        if the size of topMatches >= k
            sort topMatches in descending order of
            value
            break
        if curr points to a word
            add curr to topMatches
    for every node in topMatches
        add that node's word to topMatchesString
    return topMatchesString
```

- Complexity analysis:

Step #	Complexity stated as O(_)
1	O(1)
2	O(1)
3	O(1)
4	O(1)
5	O(1)
6	O(1)
7	O(1)
8	O(1)
9	O(1)
10	O(1)
11	O(n)
12	O(n)
13	O(n)
14	O(n)
15	O(n)
16	O(1)
17	O(1)
18	O(n)
19	O(n)
20	O(n^2)
21	O(n^2)
22	O(n)
23	O(n log(n))
24	O(1)
25	O(n)
26	O(n)
27	O(n)
28	O(n)
29	O(1)

Complexity of the algorithm = O(n^2)

2.**Testing**: Complete your test cases to test the *TrieAutoComplete* functions based upon the criteria mentioned below.

Test of correctness:

Assuming the trie already contains the terms {”ape, 6”, ”app, 4”, ”ban, 2”, ”bat, 3”, ”bee, 5”, ”car, 7”, ”cat, 1”}, you would expect results based on the following table:

Query	k	Result
””	-	Car
”a”	-	Ape
”ap”	-	Ape
”b”	-	Bee
”ba”	-	Bat
”c”	-	Car
”ca”	-	Car
”cat”	-	Cat
”d”	-	””
” ”	-	””
””	8	{”car”, ”ape”, ”bee”, ”app”, ”bat”, ”ban”, ”cat”}
””	1	{”car”}
””	2	{”car”, ”ape”}
””	3	{”car”, ”ape”, ”bee”}
”a”	1	{”ape”}
”ap”	1	{”ape”}
”b”	2	{”bee”, ”bat”}
”ba”	2	{”bee”, ”bat”}
”d”	100	{}

Test cases have been provided in the file AutocompleteTest.java.

3.**Analysis**: Answer the following questions. Use data wherever possible to justify your answers, and keep explanations brief but accurate:

NOTE: Whenever fourletterwords.txt and fourletterwordshalf.txt are compared in any of the following graphs, note that the letter “a” in fourletterwordshalf is considered synonymous to the letter “n” in fourletterwords.

i. What is the order of growth (big-Oh) of the number of compares (in the worst case) that each of the operations in the *Autocompletor* data type make?

add: O(n)

topMatch: O(n^2)

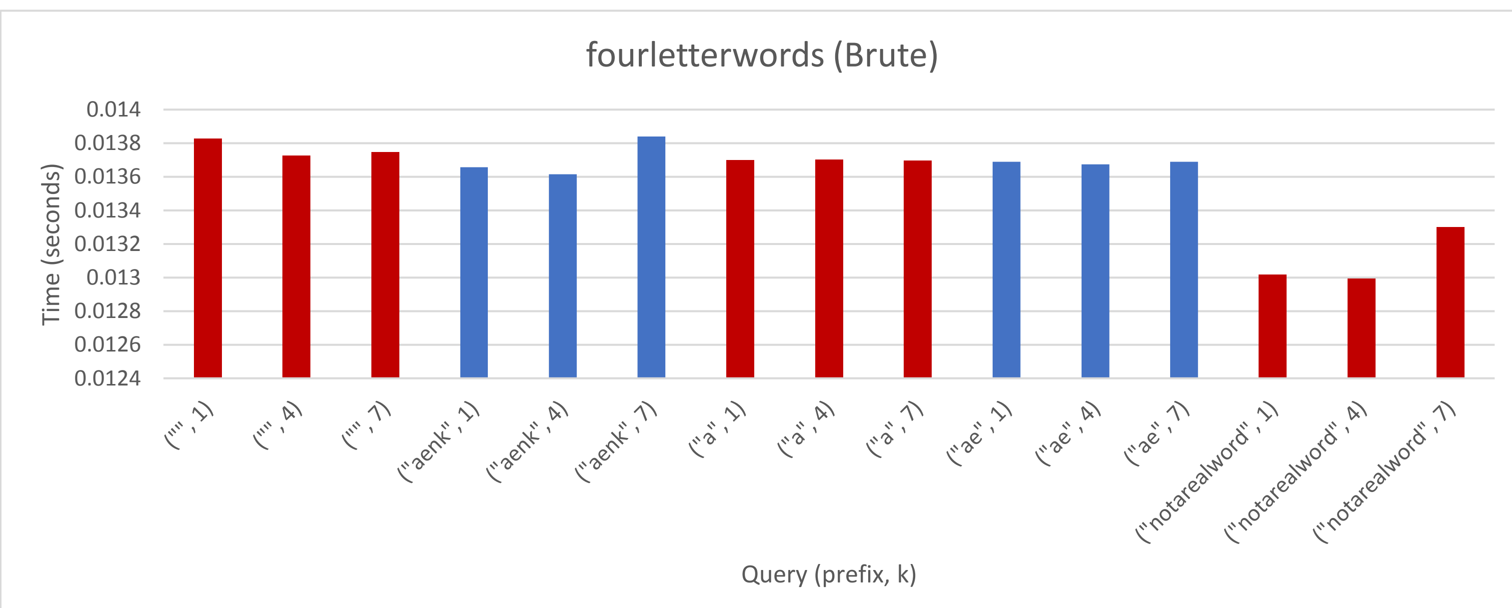
topMatches: O(n^2)

This information is presented above in Problem 1.

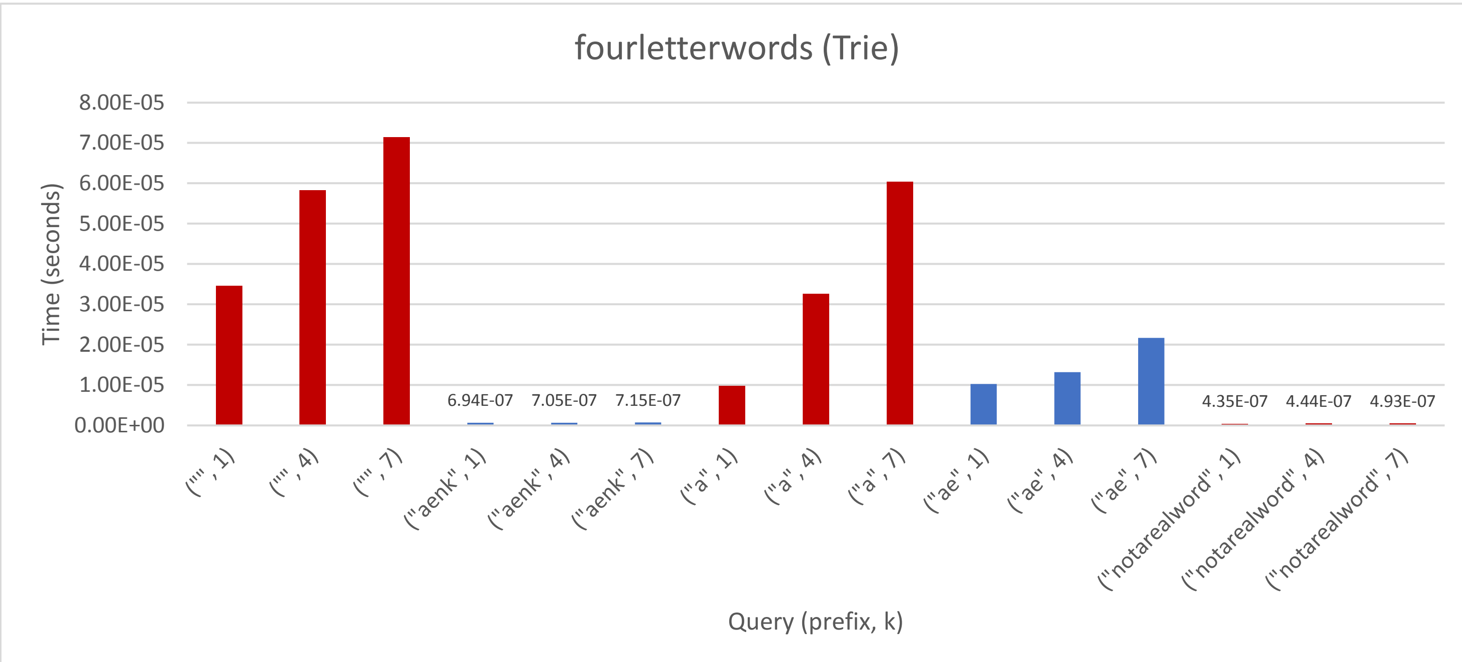
ii. How does the runtime of *topMatches()* vary with k, assuming a fixed prefix and set of terms? Provide answers for *BruteAutocomplete* and *TrieAutocomplete*. Justify your answer, with both data and algorithmic analysis.

The following observations were made using the given AutocompletorBenchmark class and fourletterwords text file.

BruteAutocomplete: The runtime of topMatches() is nearly the exact same for a fixed prefix and set of terms. This suggests that the runtime of topMatches() in this given implementation is not actually affected the value of k.

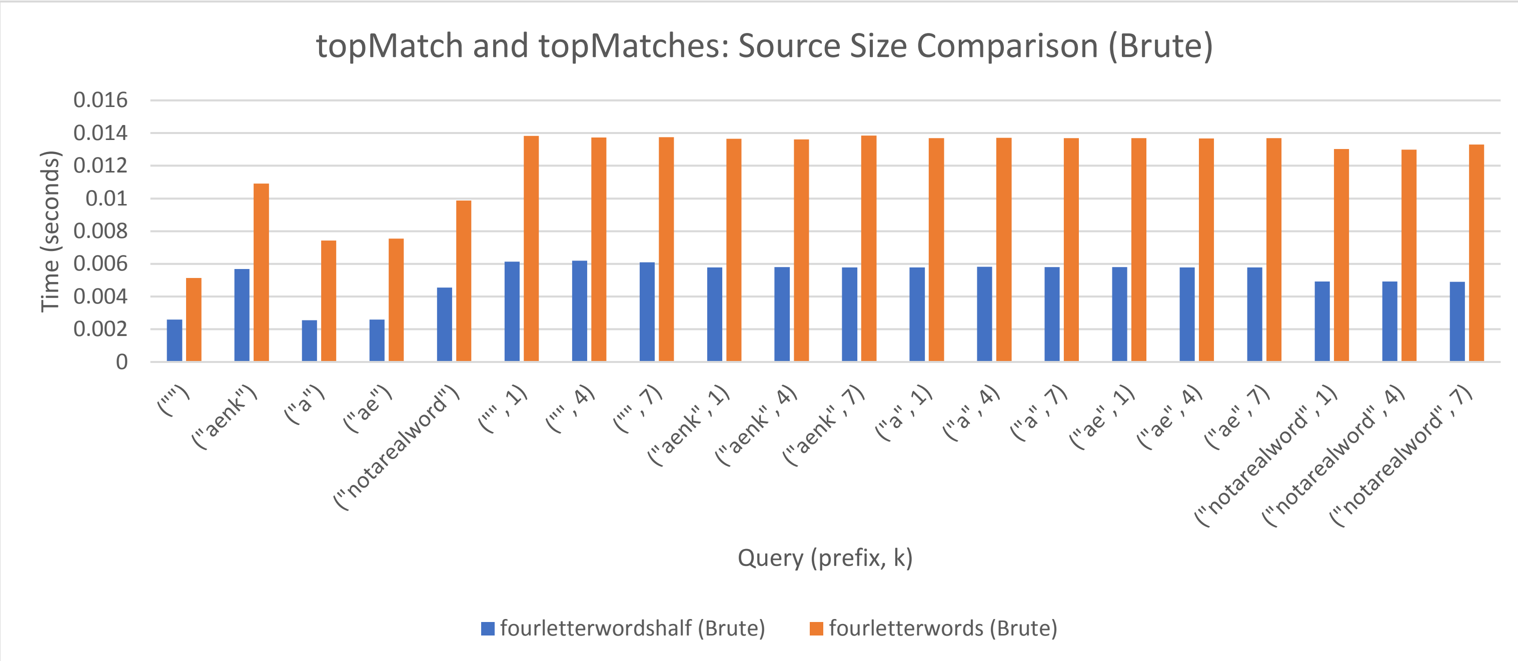


TrieAutocomplete: The runtime of topMatches() consistently increases as the value of k increases. This suggests that increasing values of k are correlated to a slower runtime. This may at least partly be attributed to the fact that TrieAutocomplete has the opportunity to break out of its loop, but BruteAutocomplete cannot do so.

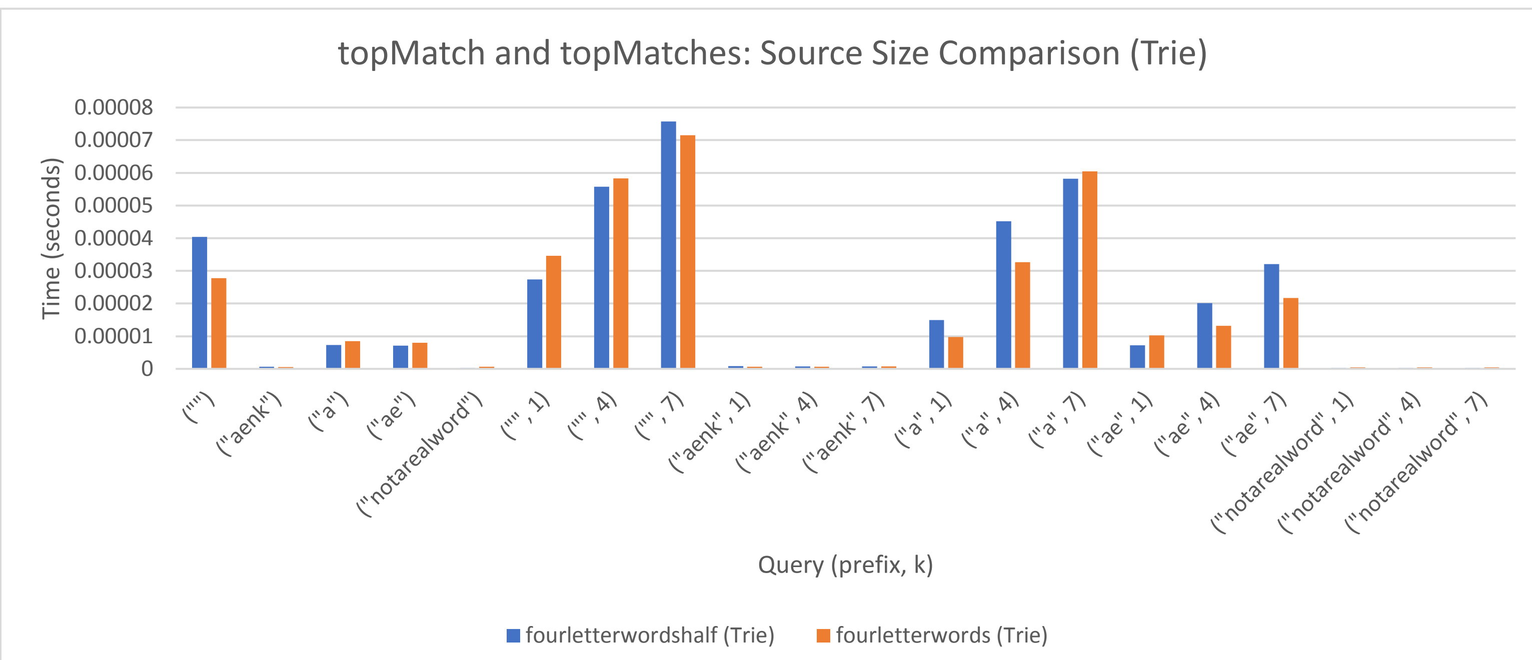


iii. How does increasing the size of the source and increasing the size of the prefix argument affect the runtime of *topMatch* and *topMatches*? (Tip: Benchmark each implementation using fourletterwords.txt, which has all four-letter combinations from aaaa to zzzz, and fourletterwordshalf.txt, which has all four-letter word combinations from aaaa to mzzz. These datasets provide a very clean distribution of words and an exact 1-to-2 ratio of words in source files.)

For BruteAutocomplete, increasing the size increases the runtime of topMatch and topMatches in a linear fashion. This is to say that doubling the input will, on average, double the runtime of these functions.



For TrieAutocomplete, increasing the size of the source does not discernibly alter the runtime of topMatch and topMatches. This means that this implementation is far more scalable in comparison to BruteAutocomplete.



4. Graphical Analysis: Provide a graphical analysis by comparing the following:

i. The big-Oh for *TrieAutoComplete* after analyzing the pseudocode and big-Oh for *TrieAutoComplete* after the implementation.

Pseudocode:

add: O(n)

topMatch: O(n^2)

topMatches: O(n^2)

Implementation:

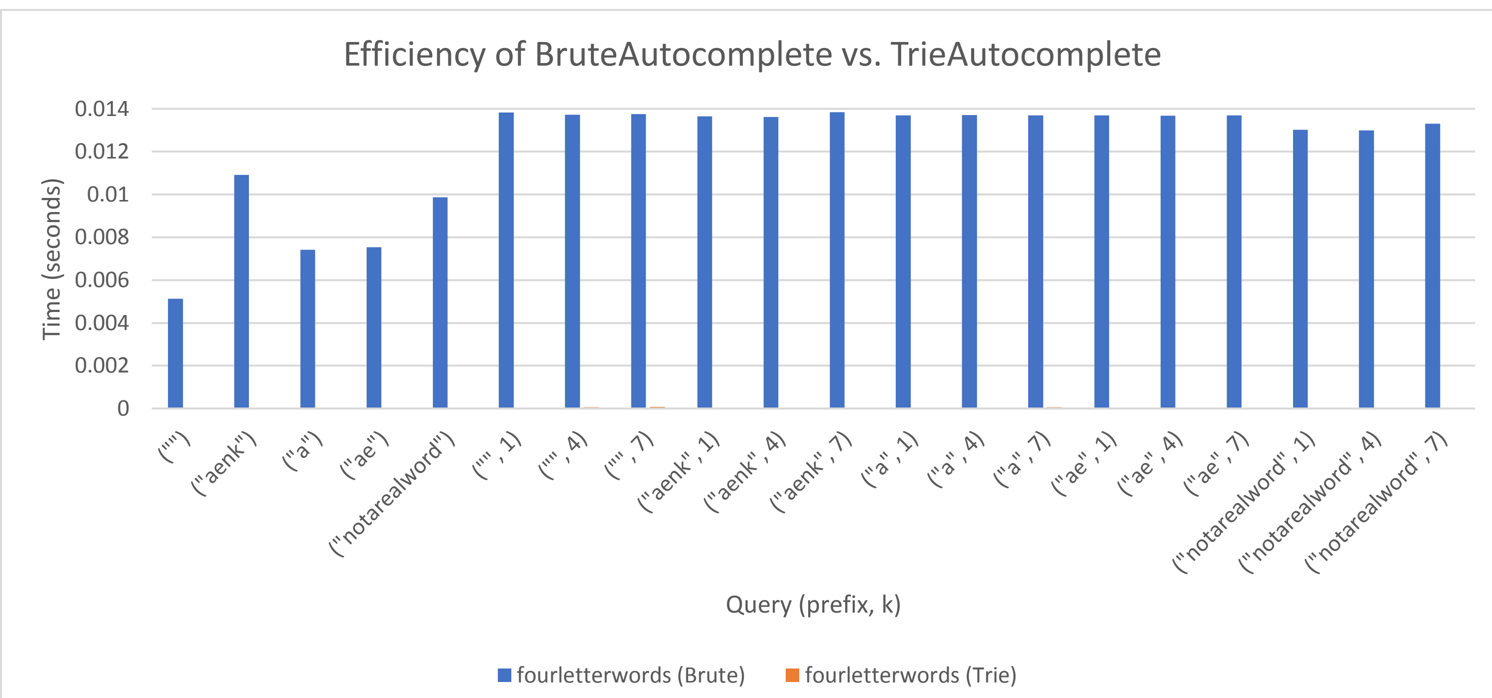
add: O(n)

topMatch: O(n^2)

topMatches: O(n^2)

ii. Compare the *TrieAutoComplete* with *BruteAutoComplete*.

In all given benchmark cases, TrieAutocomplete is orders of magnitude more efficient than BruteAutocomplete. Note that, when compared to the brute force implementation, the allotted time of the trie strategy is hardly visible in the graph.



The next chart represents the same data but more clearly represents TrieAutocomplete's allotted time when compared to BruteAutocomplete.