COMP 3270 FALL 2020

Programming Project: Autocomplete

Name: Colin Vande Vijvere Date Submitted: 11/11/2020

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, weight)

1. If word = NULL, Throw null pointer exception
2. If weight < 0, Throw Illegal Argument exception
3. NodeX = myRoot
4. For each character in word (1…n)
5. If (weight > NodeX's subtreeMaxWeight)
6. NodeX's subtreeMaxWeight = weight
7. If NodeX.getChild(word's character at n)
8. Create new Node 'Addition' (word.charAt(n), nodeX, weight)
9. nodeX.children.put(word's char at n, Addition)
10. nodeX = Addition
11. Else nodeX.getChild(word.charAt(n))
12. nodeX.setWeight(weight)
13. nodeX.isWord = true
14. nodeX.setWord(word)

- Complexity analysis:

Step #	Complexity stated as O(_)
1	O(1)
2	O(1)
3	O(1)
4	O(N)
5	O(N-1)
6	O(N-1)
7	O(N-1)
8	O(N-1)
9	O(N-1)
10	O(N-1)
11	O(N-1)
12	O(1)
13	O(1)
14	O(1)

Complexity of the algorithm = O(N + 6) which we can simplify into O(N) where N is the length of the word that is being added.

topMatch Pseudocode:

```
topMatch(string prefix)
1.  If Prefix = NULL, throw an exception
2.  nodeX = myRoot
3.  For each character in word (1…n)
4.          If nodeX does not contain character at n in children
5.                  return “”
6.          nodeX = nodeX.getChild(char at n)
7.  While (nodeX is not a word AND nodeX’s weight < nodeX’s
    subtreeMaxWeight)
8.      For each child node of nodeX (1…K)
9.          If node at K’s subtreeMaxWeight = nodeX’s subtreeMaxWeight
10.                 nodeX = node at K
11.                 Break
12. Return nodeX.getWord()
```

- Complexity analysis:

Step #	Complexity stated as O(_)
1	O(1)
2	O(1)
3	O(N)
4	O(N-1)
5	O(N-1)
6	O(N-1)
7	O(N)
8	O(N^2)
9	O(N^2)
10	O(N^2)
11	O(N^2)
12	O(1)

Complexity of the algorithm = O(N^2 + N) which we can simplify to O(N^2)

topMatches Pseudocode:

```
topMatches(string prefix, int k)
1.  If prefix = NULL, throw exception
2.  If k = 0, throw exception
3.  NodeX = myRoot
4.  Create a new PriorityQueue called ‘nodes’
5.  Create a new arrayList called ‘terms’ with term datatypes
6.  For each character in the prefix (1…n)
7.      If nodeX.getChild(char at n) == NULL, Return an empty arrayList with
        string datatype
8.  If nodeX.isWord, Add new Term(nodeX.myWord, nodeX.myWeight) to
    ‘nodes’ queue
9.  For each node ‘X’ in nodeX.children.values()
10.     If X != NULL, offer node to priorityqueue named ‘nodes’
11. While nodes.size() > 0
12.     nodeX = nodes.poll()
13.     If nodeX.isWord, Add new Term(nodeX.myWord, nodeX.myWeight)
        to ‘nodes’ queue
14.     For each node ‘X’ in nodeX.children.values()
        If X != NULL,
15.         Offer node to priorityqueue named ‘nodes’
16. Terms.sort(new Term.ReverseWeightOrder())
17. Create a new ArrayList called ‘output’ with term datatypes
18. If k > terms.size,   add all terms from ‘terms’ to ‘output’
19. Else, add terms from index 1…k in ‘terms’ to ‘output’
20. Create new ArrayList called ‘outputStrings’ with string datatypes
21. For each term in output
22.     outputStrings.add(x.getWord)
23. Return outputStrings arrayList
```

- Complexity analysis:

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

Complexity of the algorithm = O(N^2 + 2N + 12) which simplifies to O(N^2).

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

Test case is written and added to the deliverables in the folder under AutocompleteTest.java

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
””	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	{}

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

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?

Answer: there are Three functions in the autocompletor:

topMatch(), and topMatches(), and weightOf()

For topMatch(), The worst case is O(N^2).

For topMatches(), the worst case is O(n^2).

For WeightOf(), the time complexity is O(N) where N is the length of the word since it only traverses through your trie and returns the weight of the word if that word exists it works similar to the add() method when we talk about traversing the trie.

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.

Answer:

For BruteAutocomplete, a smaller k makes no difference at all since the program runs through an array containing the entire wordlist regardless of what it is looking for.

For TrieAutocomplete, when k increases, the runtime increases as well since it has to do more navigation in the trie. Note that it will still be faster than BruteForce in most cases

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.)

Fourletterwords.txt

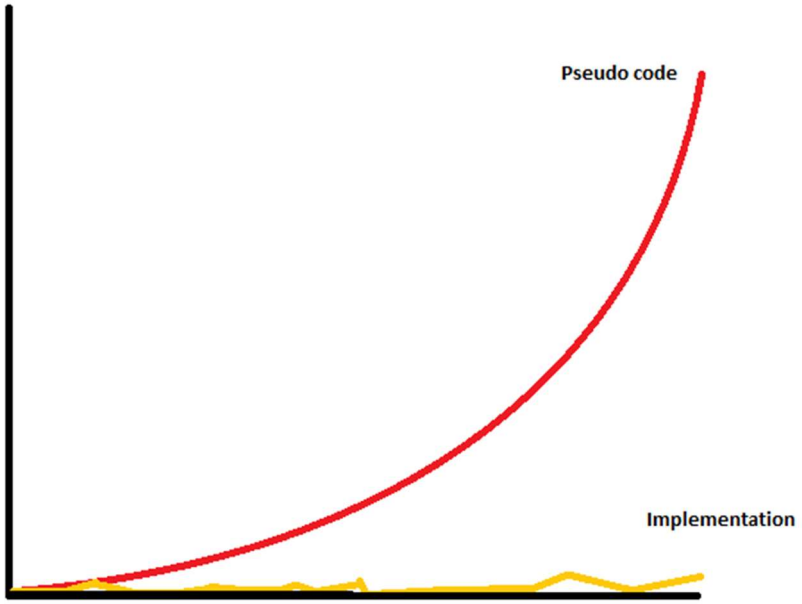
prefix	Brute	Binary	Trie
	1.60445e-03	1.64175e-03	9.42530e-06
n_1	6.21453e-03	1.05680e-04	1.06381e-02
n_4	6.41103e-03	1.09638e-04	1.07196e-02
notarealword_7	4.96547e-03	2.94370e-06	2.54100e-07
notarealword_1	4.96632e-03	3.09070e-06	5.23900e-07
nenk	4.91003e-03	3.83710e-06	5.90100e-07
n_7	5.66663e-03	9.15773e-05	9.32178e-03
notarealword_4	4.68374e-03	2.55420e-06	9.30800e-07
notarealword	5.34879e-03	6.90420e-06	6.99400e-07
n	3.64383e-03	2.92193e-05	4.36460e-06
nenk_1	6.32958e-03	9.07700e-07	4.86270e-06
_1	6.25633e-03	2.41769e-03	9.21240e-01
_4	6.78368e-03	2.33773e-03	8.99091e-01
ne	3.73213e-03	2.86180e-06	2.55650e-06
_7	7.11046e-03	2.81260e-03	7.92033e-01
ne_1	5.12219e-03	9.60380e-06	6.22915e-04
nenk_7	6.94221e-03	2.02390e-06	1.08480e-06
ne_7	5.31767e-03	8.34100e-06	2.45530e-04
nenk_4	5.31777e-03	1.00430e-06	1.35920e-06
ne_4	5.13409e-03	7.07610e-06	4.21244e-04

Fourletterwordshalf.txt

prefix	Brute	Binary	Trie
	8.73133e-04	9.15543e-04	9.00280e-06
a	9.14569e-04	2.49136e-05	4.30840e-06
notarealword_7	2.80001e-03	5.15520e-06	1.15400e-07
ae	9.94759e-04	1.96500e-06	4.85300e-06
notarealword_1	2.88071e-03	3.99270e-06	9.78000e-08
notarealword_4	3.13584e-03	5.96630e-06	9.13000e-08
notarealword	2.70094e-03	3.89430e-06	3.12000e-07
aenk	3.90363e-03	2.16450e-06	3.91620e-06
aenk_7	3.13605e-03	1.47120e-06	1.42930e-06
ae_7	2.89350e-03	1.67309e-05	1.08900e-04
_1	3.77748e-03	1.37355e-03	3.87304e-01
a_1	3.39618e-03	9.02278e-05	9.11233e-03
ae_4	2.80896e-03	9.23000e-06	1.22117e-04
_4	3.45702e-03	1.17547e-03	3.04071e-01
a_4	2.87980e-03	1.77859e-04	7.97783e-03
_7	3.28073e-03	1.45055e-03	3.00680e-01
a_7	2.87356e-03	2.42527e-04	8.88601e-03
aenk_1	3.33423e-03	1.56955e-05	1.90970e-06
ae_1	2.88730e-03	2.87612e-05	1.20709e-04
aenk_4	3.28341e-03	2.11130e-06	2.16380e-06

	Increasing the size of prefix:	Increasing the source:
BruteAutoComp.	Almost no change	Runtime increases
BinarySearchAutoComp.	Runtime decreases	Runtime increases
TrieAutoComp	Runtime decreases	Runtime decreases

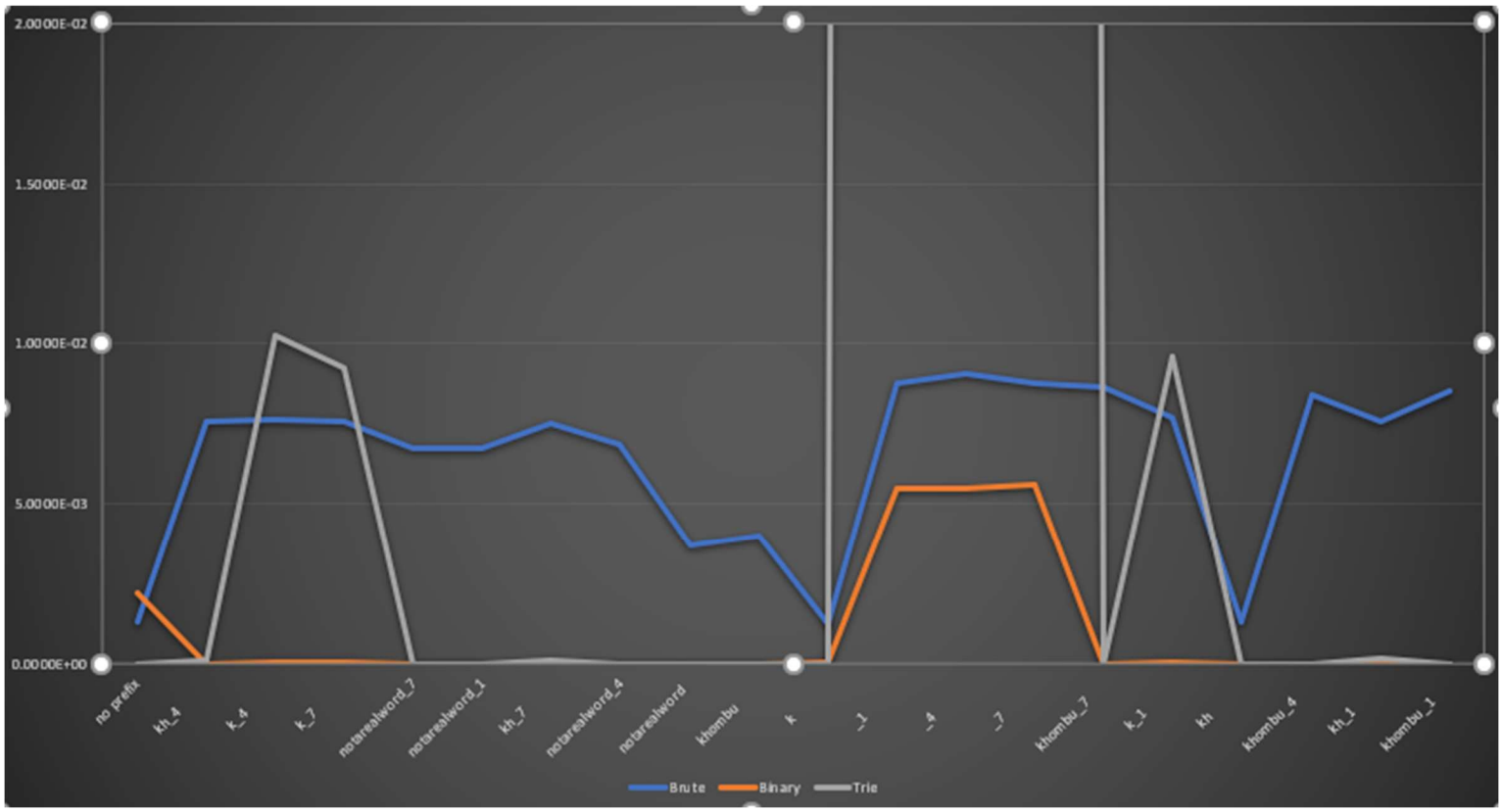
4. Graphical Analysis: Provide a graphical analysis by comparing the following:
    i. The big-Oh for *TrieAutoComplete* after analyzing the pseudocode and big-Oh for *TrieAutoComplete* after the implementation.



The Big-Oh after analyzing the pseudo code is is O(N^2). The actual implementation on the other hand seems to be running a lot faster. It appears to be O(1). I do however believe that this is more of a graphing issue and that it won't be possible to graph a proper comparison. From my own research I have found that the actual timecomplexity for our trie is O(M*N) where M is the length of the longest suggestion. Since our graph does not take into account M, it fails to portray it properly.The following article explain it rather nicely.

(https://medium.com/related-works-inc/autosuggest-retrieval-data-structures-algorithms-3a902c74ffc8)

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



2		Brute	Binary	Trie
3	no prefix	1.2663E-03	2.1897E-03	4.9696E-06
4	kh_4	7.5926E-03	3.8429E-06	1.1383E-04
5	k_4	7.5991E-03	4.9698E-05	1.0230E-02
6	k_7	7.5721E-03	4.7109E-05	9.2318E-03
7	notarealword_7	6.7422E-03	3.9965E-06	9.6440E-07
8	notarealword_1	6.7262E-03	4.5957E-06	1.9090E-07
9	kh_7	7.5055E-03	6.0158E-06	1.2559E-04
10	notarealword_4	6.8435E-03	3.3838E-06	2.9160E-07
11	notarealword	3.7017E-03	4.0805E-06	1.4550E-06
12	khombu	3.9554E-03	6.4894E-06	1.8363E-06
13	k	1.2434E-03	4.2494E-05	3.2860E-07
14	_1	8.7440E-03	5.4836E-03	5.8459E-01
15	_4	9.0673E-03	5.4920E-03	5.8810E-01
16	_7	8.7383E-03	5.5918E-03	5.7303E-01
17	khombu_7	8.6717E-03	2.6835E-06	9.6740E-07
18	k_1	7.6598E-03	5.4182E-05	9.5991E-03
19	kh	1.3197E-03	4.5696E-06	3.9050E-07
20	khombu_4	8.4077E-03	2.8075E-06	1.7608E-06
21	kh_1	7.5721E-03	3.5081E-06	1.3521E-04
22	khombu_1	8.5063E-03	2.4440E-06	4.2766E-06