COMP 3270 FALL 2020

Programming Project: Autocomplete

Name: Carson Barnett Date Submitted: 11/11/20

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. curr = myRoot
2. for char in word
3.         if char not in myRoot.children
           //build child if it does not exist
4.                 curr.children.put(char, new Node(char, curr, weight)
5.         curr = curr.getChild(c)     //traverse to child
6. curr.setWord(word)
7. curr.setWeight(weight)
8. curr.isWord = true
```

- Complexity analysis:

Step #	Complexity stated as O(_)
1.	O(1)
2.	O(n)
3.	O(1)
4.	O(1)
5.	O(1)
6.	O(1)
7.	O(1)
8.	O(1)

Complexity of the algorithm = O(n) where n is the length of the word

topMatch

- Pseudocode:

```
topMatch(prefix)
1.  Node curr = myRoot
    //traverse to proper prefix
2.  for char in prefix
3.          if char not in curr.children
4.                  return “”
5.          curr = curr.children.get(c)
6.  maxWeight = curr.mySubtreeMaxWeight
7.  while curr.mySubtreeMaxWeight != maxWeight
8.          for child in curr.children.values
9.                  if child .mySubtreeMaxWeight == maxWeight
10.                         curr = child
11.                         break
12. return curr.myWord
```

- Complexity analysis:

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

Complexity of the algorithm = O(n^2)

topMatches

- Pseudocode:

topMatches(prefix, k)

1. output = new ArrayList<String>()
   //traverse to proper node
2. Node curr = myRoot
3. for char in prefix
4. if char not in curr.children
5. return output //return empty itterable
6. curr = curr.children.get(c)
7. pq = PriorityQueue(ReverseSubtreeMaxWeightComparator()) //this pq orders nodes based on their subtreeMaxWeight so we navigate to larger weight nodes asap
8. kBest = PriorityQueue<Node> //this pq will store k nodes bases on the best weights visited so far
9. pq.add(curr)
10. while pq is not empty
11. Node next = pq.poll()
12. if kBest.size() >= k
13. if kBest.peek().getWeight()> next.mySubtreeMaxWeight
14. break //we not have our k best
15. if next.isWord()
16. kBest.add(next)
17. if (kBest.size() > k)
18. kBest.poll() //remove smallest node so we only keep k nodes in our k best
19. for child in next.children.values()
20. pq.add(child)
21. while kbest is not empty
    //after we have kbest pop nodes and add to our output
22. output.addToFront(kBest.poll().getWord())
23. return output

- Complexity analysis:

Step #	Complexity stated as O(n)
1.	O(1)
2.	O(1)
3.	O(n)
4.	O(1)

5.	O(1)
6.	O(1)
7.	O(1)
8	O(1)
9	O(log(n))
10	O(n)
11	O(log(n))
12	O(1)
13.	O(1)
14.	O(1)
15	O(1)
16	O(log(n))
17.	O(1)
18	O(log(n))
19	O(n)
20	O(n)
21	O(n)
22	O(log(n))
23.	O(1)

Complexity of the algorithm = O(n^2 * log(n))

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
””	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”} //thinks this is incorrect
”d”	100	{}

My code passed all test cases, minus the one noted above

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?

add: O(n) where n is the length of the word

topMatch: O(n^2)

topMatches: O(n^2log(n))

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.

data below is found using output from the .3333 words txt file

BruteAutocomplete:

Time for topKMatches("k", 1) - 0.003625774076

Time for topKMatches("k", 4) - 0.003620483657

Time for topKMatches("k", 7) - 0.003618026339

The run time for topKMathches does not change with the varying value of K for Brute Force AutoComplete. The brute force algorithm compares every word in the wordlist.

TreeAutocomplete:

Time for topKMatches("kh", 1) - 2.82365E-6
Time for topKMatches("kh", 4) - 3.543304E-6
Time for topKMatches("kh", 7) - 5.831168E-6

The runtime for topKMatches for TrieAutocomplete increases as k increases. This is because more items will need to be found to add to kBest, and as a result more of the trie must be traversed.

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.)

size of Prexix argument:

topMatch():

- Time for topMatch("") - 1.90499E-6
- Time for topMatch("k") - 1.21901E-6
- Time for topMatch("kh") - 4.10326E-7
- Time for topMatch("khombu") - 1.83826E-7

The size of the prefix seems to have little effect on topMatch. In fact it seems that longer prefixes are returned slightly faster, but this may just be a coincidence.

topMatches():

- Time for topKMatches("", 4) - 2.3387321E-5
- Time for topKMatches("k", 4) - 8.274493E-6
- Time for topKMatches("kh", 4) - 3.543304E-6
- Time for topKMatches("khombu", 4) - 6.44088E-7

For topMatches(), again prefix size does not seem to have much effect on the runtime. Longer prefixes again seem to do better

size of source:

using the 1 : 2 ratio as described above

topMatch()

fourletterwordshalf.txt

- Time for topMatch("") - 2.124334E-6
- Time for topMatch("aenk") - 1.55761E-7
- Time for topMatch("a") - 8.33814E-7
- Time for topMatch("ae") - 7.54751E-7
- Time for topMatch("notarealword") - 5.8428E-

fourletterwords.txt

- Time for topMatch("") - 2.037756E-6
- Time for topMatch("nenk") - 1.44993E-7
- Time for topMatch("n") - 8.94592E-7
- Time for topMatch("ne") - 6.63976E-7

- Time for topMatch("notarealword") - 1.44386E-7

When the source size is doubled, the run time for top match does not greatly change, which means the algorithm is pretty scalable.

topMatches():

fourletterwordshalf.txt

- Time for topKMatches("aenk", 1) - 3.17615E-7
- Time for topKMatches("aenk", 4) - 2.90529E-7
- Time for topKMatches("aenk", 7) - 2.8576E-7

fourletterwords.txt

- Time for topKMatches("nenk", 1) - 3.30297E-7
- Time for topKMatches("nenk", 4) - 2.99439E-7
- Time for topKMatches("nenk", 7) - 3.02464E-7

For topMatches, increasing the source by a factor of 2, also does not greatly change the runtime. This again signifies our algorithm is scalable.

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.

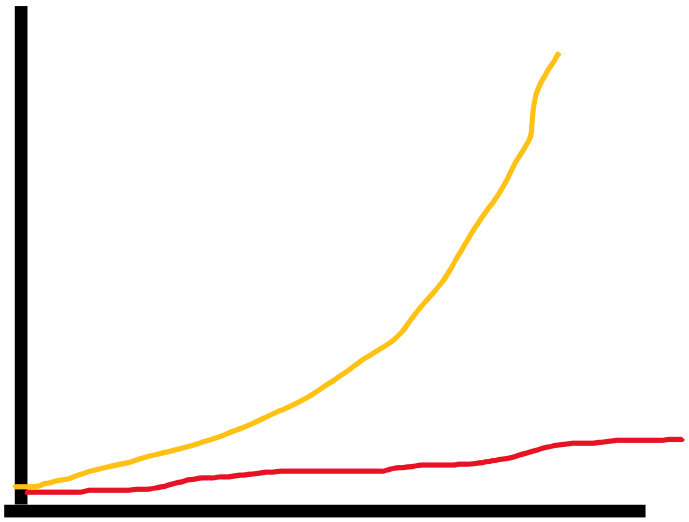
pseudocode:

add: O(n)
topMatch(): O(n^2)
topMatches(): O(n ^2*log(n))

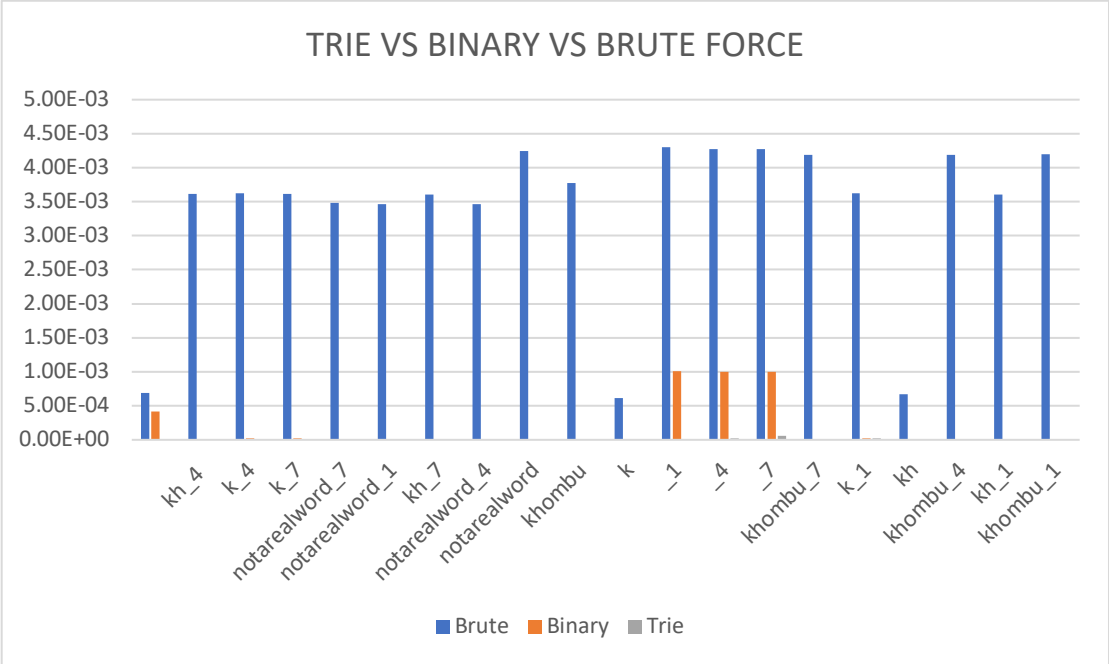
implementation: using data above
add: O(n)
topMatch(): seems to do way better than expected, in some cases approaches O(1)

topMatches(): seems to again to way better than I would have expected, in some cases approaching O(1)

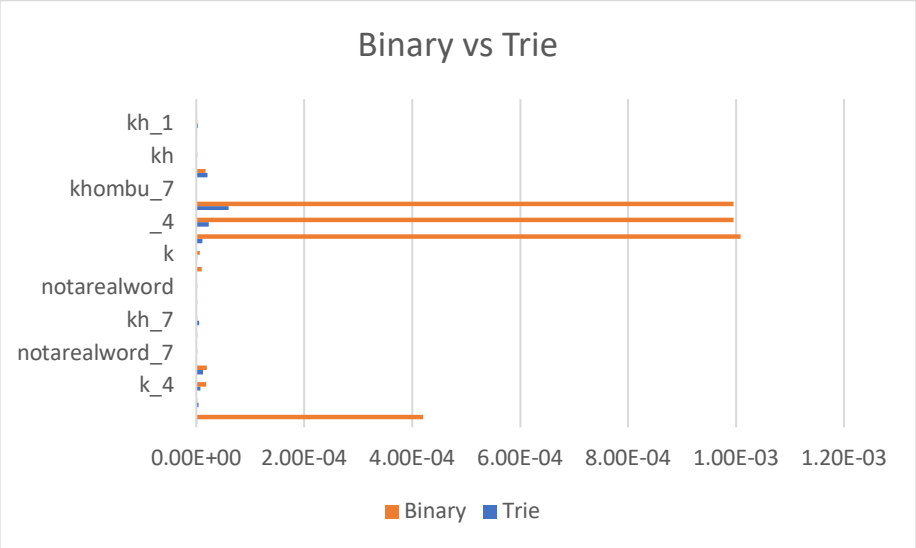
Pseudocode (n^2* log(n))
implementation (constant)



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



It is evident in the graph above that brute is magnitudes worse in runtime than both the trie and binary implementations. Because this difference in magnitude is so drastic, it is hard to fully analyze the binary and trie implentations. Another graph below is provided only comparing binary vs trie.



It is evident in the above graph, that though both implementations are efficient, the trie implementation is way more scalable than the binary implementation.

Also my full benchmarking for the 333333 words is attached below.

Opening - /Users/carsonbarnett/Desktop/School-Stuff/COMP3270/Programming Project 3270 FA 20/autocomplete-start/data/words-333333.txt.
Found 333333 words
Time to initialize BruteAutoComplete - 0.060395073
Time to initialize Autocomplete.BinarySearchAutocomplete - 0.443657166
Time to initialize - Autocomplete.TrieAutocomplete - 0.365406046
Benchmarking Autocomplete$BruteAutocomplete...
Time for topMatch("") - 6.93609532E-4
Time for topMatch("khombu") - 0.003776081219
Time for topMatch("k") - 6.15100082E-4

```
Time for topMatch("kh") - 6.72093136E-4
Time for topMatch("notarealword") - 0.004242528538
Time for topKMatches("", 1) -  0.004303419974
Time for topKMatches("", 4) -  0.004273343661
Time for topKMatches("", 7) -  0.004270007594
Time for topKMatches("khombu", 1) -  0.004193496175
Time for topKMatches("khombu", 4) -  0.004186569208
Time for topKMatches("khombu", 7) -  0.004190980032
Time for topKMatches("k", 1) -  0.003625774076
Time for topKMatches("k", 4) -  0.003620483657
Time for topKMatches("k", 7) -  0.003618026339
Time for topKMatches("kh", 1) -  0.003602107044
Time for topKMatches("kh", 4) -  0.003616190504
Time for topKMatches("kh", 7) -  0.003601185208
Time for topKMatches("notarealword", 1) -  0.003468668183
Time for topKMatches("notarealword", 4) -  0.003460696081
Time for topKMatches("notarealword", 7) -  0.00348003001
Benchmarking Autocomplete$BinarySearchAutocomplete...
Time for topMatch("") - 4.19729757E-4
Time for topMatch("khombu") - 9.750043E-6
Time for topMatch("k") - 6.707333E-6
Time for topMatch("kh") - 1.52323E-6
Time for topMatch("notarealword") - 1.538973E-6
Time for topKMatches("", 1) -  0.001008499408
Time for topKMatches("", 4) -  9.95349519E-4
Time for topKMatches("", 7) -  9.95110342E-4
Time for topKMatches("khombu", 1) -  1.122534E-6
Time for topKMatches("khombu", 4) -  1.111113E-6
Time for topKMatches("khombu", 7) -  1.103493E-6
Time for topKMatches("k", 1) -  1.7757391E-5
Time for topKMatches("k", 4) -  1.865427E-5
Time for topKMatches("k", 7) -  2.0079506E-5
Time for topKMatches("kh", 1) -  1.653772E-6
Time for topKMatches("kh", 4) -  1.90263E-6
Time for topKMatches("kh", 7) -  2.153093E-6
Time for topKMatches("notarealword", 1) -  1.905598E-6
Time for topKMatches("notarealword", 4) -  1.291562E-6
Time for topKMatches("notarealword", 7) -  1.263439E-6
Benchmarking Autocomplete$TrieAutocomplete...
Created 805917 nodes
Time for topMatch("") - 1.90499E-6
Time for topMatch("khombu") - 1.83826E-7
Time for topMatch("k") - 1.21901E-6
```

Time for topMatch("kh") - 4.10326E-7
Time for topMatch("notarealword") - 1.79474E-7
Time for topKMatches("", 1) - 1.1192139E-5
Time for topKMatches("", 4) - 2.3387321E-5
Time for topKMatches("", 7) - 6.01503E-5
Time for topKMatches("khombu", 1) - 7.00937E-7
Time for topKMatches("khombu", 4) - 6.44088E-7
Time for topKMatches("khombu", 7) - 6.41968E-7
Time for topKMatches("k", 1) - 2.1015713E-5
Time for topKMatches("k", 4) - 8.274493E-6
Time for topKMatches("k", 7) - 1.3006596E-5
Time for topKMatches("kh", 1) - 2.82365E-6
Time for topKMatches("kh", 4) - 3.543304E-6
Time for topKMatches("kh", 7) - 5.831168E-6
Time for topKMatches("notarealword", 1) - 1.84072E-7
Time for topKMatches("notarealword", 4) - 2.28225E-7
Time for topKMatches("notarealword", 7) - 1.73009E-7

****

Printing Summary of Results ...

****

prefix , Brute , Binary , Trie

, 6.93610e-04, 4.19730e-04, 1.90499e-06
kh_4 , 3.61619e-03, 1.90263e-06, 3.54330e-06
k_4 , 3.62048e-03, 1.86543e-05, 8.27449e-06
k_7 , 3.61803e-03, 2.00795e-05, 1.30066e-05
notarealword_7 , 3.48003e-03, 1.26344e-06, 1.73009e-07
notarealword_1 , 3.46867e-03, 1.90560e-06, 1.84072e-07
kh_7 , 3.60119e-03, 2.15309e-06, 5.83117e-06
notarealword_4 , 3.46070e-03, 1.29156e-06, 2.28225e-07
notarealword , 4.24253e-03, 1.53897e-06, 1.79474e-07
khombu , 3.77608e-03, 9.75004e-06, 1.83826e-07
k , 6.15100e-04, 6.70733e-06, 1.21901e-06
_1 , 4.30342e-03, 1.00850e-03, 1.11921e-05
_4 , 4.27334e-03, 9.95350e-04, 2.33873e-05
_7 , 4.27001e-03, 9.95110e-04, 6.01503e-05
khombu_7 , 4.19098e-03, 1.10349e-06, 6.41968e-07
k_1 , 3.62577e-03, 1.77574e-05, 2.10157e-05

kh , 6.72093e-04, 1.52323e-06, 4.10326e-07
khombu_4 , 4.18657e-03, 1.11111e-06, 6.44088e-07
kh_1 , 3.60211e-03, 1.65377e-06, 2.82365e-06
khombu_1 , 4.19350e-03, 1.12253e-06, 7.00937e-07
, 6.93610e-04, 4.19730e-04, 1.90499e-06
kh_4 , 3.61619e-03, 1.90263e-06, 3.54330e-06
k_4 , 3.62048e-03, 1.86543e-05, 8.27449e-06
k_7 , 3.61803e-03, 2.00795e-05, 1.30066e-05
notarealword_7 , 3.48003e-03, 1.26344e-06, 1.73009e-07
notarealword_1 , 3.46867e-03, 1.90560e-06, 1.84072e-07
kh_7 , 3.60119e-03, 2.15309e-06, 5.83117e-06
notarealword_4 , 3.46070e-03, 1.29156e-06, 2.28225e-07
notarealword , 4.24253e-03, 1.53897e-06, 1.79474e-07
khombu , 3.77608e-03, 9.75004e-06, 1.83826e-07
k , 6.15100e-04, 6.70733e-06, 1.21901e-06
_1 , 4.30342e-03, 1.00850e-03, 1.11921e-05
_4 , 4.27334e-03, 9.95350e-04, 2.33873e-05
_7 , 4.27001e-03, 9.95110e-04, 6.01503e-05
khombu_7 , 4.19098e-03, 1.10349e-06, 6.41968e-07
k_1 , 3.62577e-03, 1.77574e-05, 2.10157e-05
kh , 6.72093e-04, 1.52323e-06, 4.10326e-07
khombu_4 , 4.18657e-03, 1.11111e-06, 6.44088e-07
kh_1 , 3.60211e-03, 1.65377e-06, 2.82365e-06
khombu_1 , 4.19350e-03, 1.12253e-06, 7.00937e-07
, 6.93610e-04, 4.19730e-04, 1.90499e-06
kh_4 , 3.61619e-03, 1.90263e-06, 3.54330e-06
k_4 , 3.62048e-03, 1.86543e-05, 8.27449e-06
k_7 , 3.61803e-03, 2.00795e-05, 1.30066e-05
notarealword_7 , 3.48003e-03, 1.26344e-06, 1.73009e-07
notarealword_1 , 3.46867e-03, 1.90560e-06, 1.84072e-07
kh_7 , 3.60119e-03, 2.15309e-06, 5.83117e-06
notarealword_4 , 3.46070e-03, 1.29156e-06, 2.28225e-07
notarealword , 4.24253e-03, 1.53897e-06, 1.79474e-07
khombu , 3.77608e-03, 9.75004e-06, 1.83826e-07
k , 6.15100e-04, 6.70733e-06, 1.21901e-06
_1 , 4.30342e-03, 1.00850e-03, 1.11921e-05
_4 , 4.27334e-03, 9.95350e-04, 2.33873e-05

_7 , 4.27001e-03, 9.95110e-04, 6.01503e-05
khombu_7 , 4.19098e-03, 1.10349e-06, 6.41968e-07
k_1 , 3.62577e-03, 1.77574e-05, 2.10157e-05
kh , 6.72093e-04, 1.52323e-06, 4.10326e-07
khombu_4 , 4.18657e-03, 1.11111e-06, 6.44088e-07
kh_1 , 3.60211e-03, 1.65377e-06, 2.82365e-06
khombu_1 , 4.19350e-03, 1.12253e-06, 7.00937e-07