COMP 3270 FALL 2020
Programming Project: Autocomplete

Name: Sean O'Connor Date Submitted: 11-8-20

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: a String, weight: a Double)
  1. input = word
  2. wordWeight = weight
  3. current = myRoot
  4. if (wordWeight > current.mySubtreeMaxWeight) then current.mySubtreeMaxWeight = wordWeight
  5. len = 0
  6. while (len < input.length)
  7. temp = new Node(input.charAt(len), current, wordWeight)
  8. if (!current.children.containsKey(input.charAt(len))) then current.children.put(input.charAt(len), temp)
  9. current = temp
  10. if (current.mySubtreeMaxWeight < wordWeight) then current.mySubtreeMaxWeight = wordWeight
  11. len++
  12. current.isWord = true
  13. current.setWord(input)
  14. if (current.mySubtreeMaxWeight == current.getWeight())
  15. current.setWeight(wordWeight)
  16. tempWeight = wordWeight
  17. len = 0
  18. while (len <= input.length)
  19. for (temp : current.children.values())
  20. if (tempWeight < temp.mySubtreeMaxWeight) then tempWeight = temp.mySubtreeMaxWeight
  21. if (tempWeight < current.getWeight()) then tempWeight = current.getWeight()
  22. current.mySubtreeMaxWeight = tempWeight
  23. current = current.parent
  24. len++
  25. else current.setWeight(wordWeight)

- Complexity analysis:
  - **Everything not listed is assumed O(1)

Step #	Complexity stated as O(_)
6	O(n)
18	O(n^2)
19	O(n^2)

Complexity of the algorithm = O(n^2)

topMatch

- Pseudocode:

topMatch(A String: prefix)

1. look = prefix
2. weight = 0
3. current = myRoot
4. for i = 0 to look.length()
5. if current.children.get(look.charAt(i)) == null then return null
6. current = current.children.get(look.charAt(i))
7. weight = current.mySubtreeMaxWeight
8. while weight != current.getWeight()
9. for each Node temp in current.children.values()
10. If temp.mySubtreeMaxWeight == weight then current = temp
11. return current.getWord()

- Complexity analysis:

** Everything not listed assume O(1)

Step #	Complexity stated as O(_)
4	O(n)
8	O(n)
9	O(n^2)

Complexity of the algorithm = O(n^2)

topMatches

- Pseudocode:

topMatches(A String: prefix, A int: k)

1. Current = myRoot
2. empty = new ArrayList()
3. finalWord = new ArrayList()
4. terms = new ArrayList()
5. priority = new PriorityQueue(ReverseSubtreeMaxWeightComparator)

```
6.  i = 0
7.  while i < prefix.length()
8.          if current.children.get(prefix.charAt(i)) then return empty
9.          Current = current.children.get(prefix.charAt(i))
10.         i++
11. priority.add(current)
12. while priority.size > 0
13.         current = priority.remove()
14.         If terms.size() >= k
15.                 Collections.sort(terms, new Term.ReverseWeightOrder())
16.                 If terms.get(terms.size()-1).getWeight() > current.mySubtreeMaxWeight then break
17.         For each Node temp in current.children.values()
18.                 Priority.add(temp)
19.         If current.isWord
20.                 temp = new Term(current.getWord(), current.getWeight())
21.                 terms.add(temp)
22. If terms.size() < k then k = terms.size()
23. Collections.sort(terms, new Term.ReverseWeightOrder())
24. For i = 0 to i < k
25.         finalWord.add(terms.get(i).getWord())
26. return finalWord
```

- Complexity analysis:

  ** Everything not listed assume O(1)

Step #	Complexity stated as O(_)
7	O(n)
12	O(n)
17	O(n^2)
24	O(n)

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
""	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?

   a. Add()
      i. The worst case big-Oh of the add function is O(n^2) because of a special case. This special case is when the word already exists in the tree. When this happens, the tree is traversed completely and back up to ensure the weight of the word has been manipulated to the proper value.
   b. TopMatch()
      i. Once again, the worst case big-Oh of the TopMatch function is O(n^2). This is the case when the heaviest weighted word is found as the last node. The inside loop will run to the end of the child nodes each time and do it until the outer loop is complete. This causes the method to have an O(n^2) complexity in the worst case.
   c. topMatches()
      i. Finally, the worst case big-Oh for the topMatches method is also O(n^2). This occurs when the outer loop is iterating k amount of times while the inner loop is iterating until each child is traversed of the parent node. This event in itself causes the method to have a worst case big-Oh complexity of O(n^2).

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.

a. BruteAutocomplete –

The topMatches method runtime does not seem to vary much when the prefix is held constant and k is manipulated. This assumption is created based of the following data:

Time for topKMatches("", 1) - 0.008096081428802589
Time for topKMatches("", 4) - 0.006005563578631453
Time for topKMatches("", 7) - 0.004642863378
Time for topKMatches("nenk", 1) - 0.005108827320735444
Time for topKMatches("nenk", 4) - 0.00402811699
Time for topKMatches("nenk", 7) - 0.003845924182
Time for topKMatches("n", 1) - 0.003665685866
Time for topKMatches("n", 4) - 0.00365310743
Time for topKMatches("n", 7) - 0.003642438558
Time for topKMatches("ne", 1) - 0.003656492915
Time for topKMatches("ne", 4) - 0.003613951834
Time for topKMatches("ne", 7) - 0.004263570793

This data accurately shows that the runtime does not change drastically.

b. TrieAutocomplete –

This time, the topMatches method does change when the k value is varied and the prefix is held constant. The same data is tested here as was for BruteAutocomplete.

Time for topKMatches("", 1) - 4.0143096E-5
Time for topKMatches("", 4) - 5.6286729E-5
Time for topKMatches("", 7) - 3.4261738E-5
Time for topKMatches("nenk", 1) - 6.15063E-7
Time for topKMatches("nenk", 4) - 7.57588E-7
Time for topKMatches("nenk", 7) - 4.93535E-7
Time for topKMatches("n", 1) - 3.642341E-6
Time for topKMatches("n", 4) - 1.2937999E-5
Time for topKMatches("n", 7) - 2.5697395E-5
Time for topKMatches("ne", 1) - 2.500346E-6
Time for topKMatches("ne", 4) - 5.26641E-6
Time for topKMatches("ne", 7) - 9.015897E-6

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.)

a. BruteAutocomplete –

fourletter:
Time for topMatch("") - 8.52575496E-4
Time for topMatch("notarealword") - 0.00377833172
Time for topKMatches("", 1) - 0.004231578273
Time for topKMatches("notarealword", 1) - 0.003341263271

fourletterhalf:
Time for topMatch("") - 4.48966641E-4
Time for topMatch("notarealword") - 0.001323776952
Time for topKMatches("", 1) - 0.002233902963
Time for topKMatches("notarealword", 1) - 0.001613238019

For this class, the runtime almost doubles as the data file doubles. This could be a source of issue when dealing with larger files.

b. BinaryAutocomplete –

fourletter:
Time for topMatch("") - 7.40827534E-4
Time for topMatch("notarealword") - 2.235098E-6
Time for topKMatches("", 1) - 0.001769715531
Time for topKMatches("notarealword", 1) - 9.79521E-7

fourletterhalf:
Time for topMatch("") - 3.98011059E-4
Time for topMatch("notarealword") - 2.828661E-6
Time for topKMatches("", 1) - 9.7806588E-4
Time for topKMatches("notarealword", 1) - 8.59651E-7

For this class, the runtime is better than brute; however, as the file size grows the runtime grows slowly which means it could handle bigger files reatively easier than brute.

c. TrieAutocomplete –

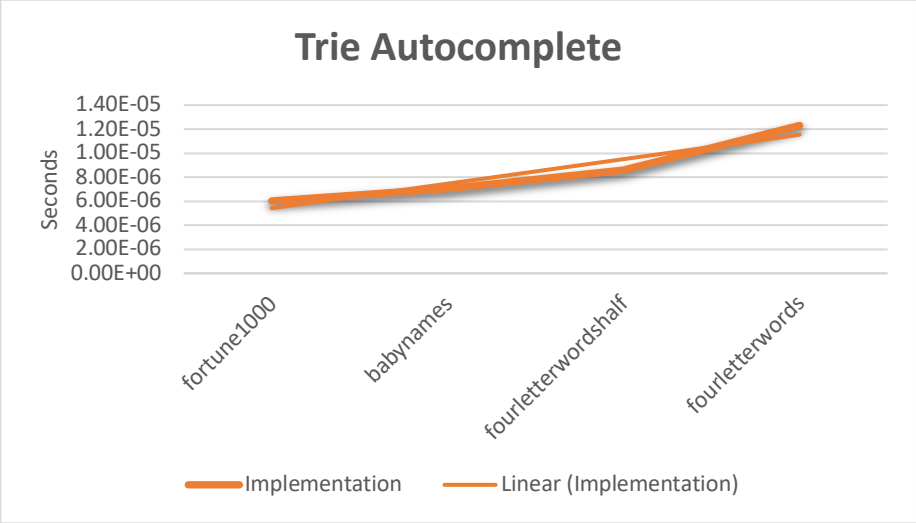
fourletter:
Time for topMatch("") - 8.429851E-6
Time for topMatch("notarealword") - 5.19185E-7
Time for topKMatches("", 1) - 8.1546883E-5
Time for topKMatches("notarealword", 1) - 6.24636E-7

fourletterhalf:
Time for topMatch("") - 2.4577049E-5
Time for topMatch("notarealword") - 1.76157E-7
Time for topKMatches("", 1) - 5.8949008E-5
Time for topKMatches("notarealword", 1) - 1.78957E-7

For this class, the runtime growth related to the file size is similar to binary except it is a tad more efficient.

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.



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