COMP 3270 FALL 2017

**Programming Project: Autocomplete**

Name: Jackson Pettus Date Submitted: 11-17-2017

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:

1. if word is null then
2. throw nullpointerexception
3. if weight < 0 then
4. throw illegalargumentexception
5. if myRoot.mySubtreeMaxWeight < weight then
6. myRoot.mySubtreeMaxWeight < weight
7. set current = myRoot
8. for i = 1 to word.length
9. if current.mySubtreeMaxWeight < weight then
10. current.mySubtreeMaxWeight = weight
11. if !current.children.containsKey(current.charAt(i)) then
12. add child node with key = charAt(i), parent = currentnode,
13. and weight = weight
14. set current to current.children with charAt(i)
15. set current.isWord = true, current.myWord = word
16. if current.isWord and current.myWeight > weight
17. set current.myWeight
18. while current isn’t null
19. maxWeight = -1
20. for each char in curr.children.keySet
21. if curr.getChild(char).mySubtreeMaxWeight > maxWeight then
22. set maxWeight to mySubtreeMaxWeight
23. curr.mySubtreeMaxWeight = maxWeight
24. curr = curr.parent
25. else set curr.isWord=true, curr.setWeight(weight), curr.setWord(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(w) where w = word.length |
| 9 | O(1) |
| 10 | O(1) |
| 11 | O(1) |
| 12 | O(1) |
| 13 | O(1) |
| 14 | O(1) |
| 15 | O(1) |
| 16 | O(1) |
| 17 | O(1) |
| 18 | O(1) |
| 19 | O(1) |
| 20 | O(1) – amount of characters |
| 21 | O(1) |
| 22 | O(1) |
| 23 | O(1) |
| 24 | O(1) |
| 25 | O(1) |

Complexity of the algorithm = O(w)

*topMatch*

* Pseudocode:

1. If prefix is null
2. Throw null pointer exception
3. set curr = myRoot
4. for int i to prefix.length
5. if curr.children containsKey(prefix.charAt(i))
6. set curr = curr.children.get(prefix.charAt(i))
7. else return empty string
8. if curr.mySubtreeMaxWeight == curr.getWeight and curr is a word then return curr.getWord()
9. while !curr.mySubtreeMaxWeight == curr.getWeight and !curr.isWord
10. for each char in curr.children.keySet()
11. if curr.children at char.mySubtreeMaxWeight == curr.mySubtreeMaxWeight then
12. set curr = curr.children.get(char)
13. break
14. return curr.myWord

* Complexity analysis:

|  |  |
| --- | --- |
| Step # | Complexity stated as O(\_) |
| 1 | O(1) |
| 2 | O(1) |
| 3 | O(1) |
| 4 | O(p) where p = prefix length |
| 5 | O(1) |
| 6 | O(1) |
| 7 | O(1) |
| 8 | O(1) |
| 9 | O(1) |
| 10 | O(26) |
| 11 | O(1) |
| 12 | O(1) |
| 13 | O(1) |
| 14 | O(1) |

Complexity of the algorithm = O(p)

*topMatches*

Pseudocode:

1. If prefix is null
2. Throw null pointer exception
3. set curr = myRoot
4. for int i to prefix.length
5. if curr.children containsKey(prefix.charAt(i))
6. set curr = curr.children.get(prefix.charAt(i))
7. else return empty iterable string
8. pq = PriorityQueue<Node>(k, new Node.ReverseMaxWeightComparator()
9. min = PriorityQueue<Node>(k)
10. add curr to pq
11. while pq !isEmpty
12. curr = pq.remove
13. if min.size == k and min.peek.myWeight< curr.myWeight and curr.isWord
14. min.poll()
15. min.add(curr)
16. else if curr isWord
17. min.add(curr)
18. for each char in curr.children.keySet()
19. pq.add(curr.children.get(char)
20. create new iterable string ArrayList answers
21. create new Node array nodes with size = min.size()
22. for int i = 1 to nodes.length and k
23. int index = nodes.length - 1- i
24. add nodes[index].getWord() to answers
25. return answers
26. Complexity analysis:

|  |  |
| --- | --- |
| Step # | Complexity stated as O(\_) |
| 1 | O(1) |
| 2 | O(1) |
| 3 | O(1) |
| 4 | O(p) |
| 5 | O(1) |
| 6 | O(1) |
| 7 | O(1) |
| 8 | O(1) |
| 9 | O(1) |
| 10 | O(1) |
| 11 | O(1) |
| 12 | O(1) |
| 13 | O(1) |
| 14 | O(1) |
| 15 | O(1) |
| 16 | O(1) |
| 17 | O(1) |
| 18 | O(26) |
| 19 | O(1) |
| 20 | O(1) |
| 21 | O(1) |
| 22 | O(k) |
| 23 | O(1) |
| 24 | O(1) |

Complexity of the algorithm = O(k)

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 | {} |

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

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 *Autocompletor* data type make?

Add: O(w) where w is the length of the word being added

topMatches: O(p) where p is the length of the prefix

topMatch: O(p) where p is the length of the prefix

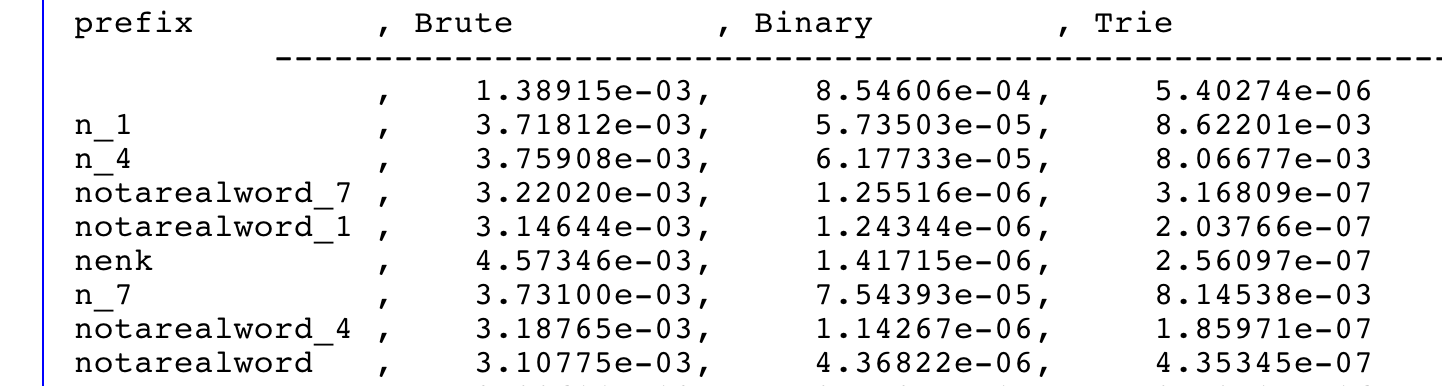
1. How does the runtime of *topMatches()* vary with k, assuming a fixed prefix and set of terms? Provide answers for *BruteAutocomplete* and *TrieAutocomplete*. Justify your answer, with both data and algorithmic analysis.

TrieAutoComplete will have to navigate k paths of the trie, so the time complexity in terms of big Oh will not change, but the runtime will increase by a miniscule amount. BruteAutoComplete will have to search the entire data set k times, so its run time will increase exponentially as k increases.

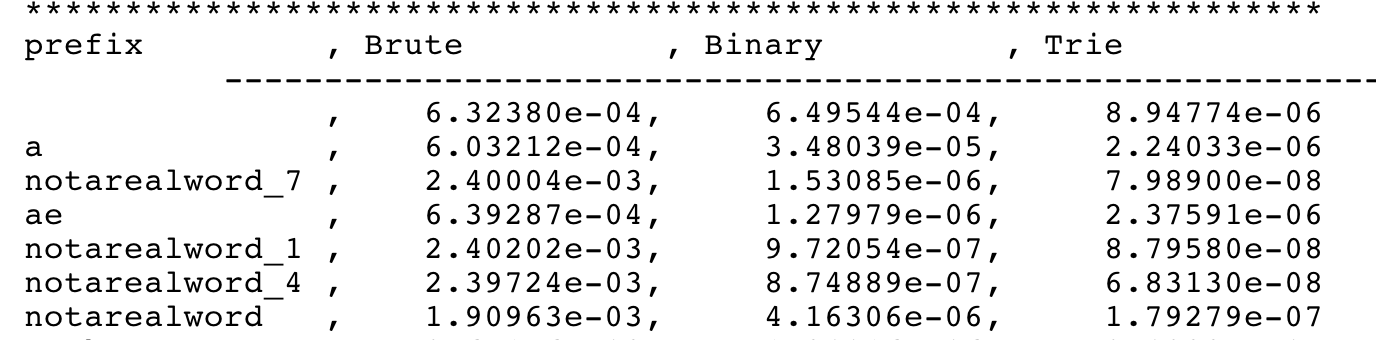
1. How does increasing the size of the source and increasing the size of the prefix argument affect the runtime of *topMatch* and *topMatches*? (Tip: Benchmark each implementation using fourletterwords.txt, which has all four-letter combinations from aaaa to zzzz, and fourletterwordshalf.txt, which has 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.)

Since TrieAutocomplete topMatches is O(k), its time complexity will not change in big Oh terms but will increase in run time a small amount. Brute autocomplete’s time increases much faster in terms of big Oh because it does not utilize a data structure and has to search through the entire set instead of just one path of the trie.

Fourletterwords.txt:



fourletterwordshalf.txt:



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

1. The big-Oh for *TrieAutoComplete* after analyzing the pseudocode and big-Oh for *TrieAutoComplete* after the implementation.
2. Compare the *TrieAutoComplete* with *BruteAutoComplete*.