COMP 3270 SPRING 2020

**Programming Project: Autocomplete**

Name: Seth Kinsaul Date Submitted: April 8, 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:

1. for i = 1 to wordArray.length
2. if (node.mySubtreeMaxWeight < weight)
3. node.mySubtreeMaxWeight = weight
4. if not(node.children.contains(i))
5. node.children.put(i, child)
6. else
7. node = node.getChild(i)
8. node.setWeight(weight)
9. node.setWord(word)
10. node.isWord = true

* Complexity analysis:

|  |  |
| --- | --- |
| Step # | Complexity stated as O(\_) |
| 1 | O(n) |
| 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) |

Complexity of the algorithm = O(n)

*topMatch*

* Pseudocode:

1. for i = 1 to prefixArray.length
2. if (node.children.contains(i))
3. node = node.getChild(i)
4. else
5. return ““
6. if (node.myWeight == largestWW)
7. return node.myWord
8. while (node.myWeight != largestWW)
9. for j = 1 to node.children.values
10. if (child.mySubtreeMaxWeight == largestWW)
11. node = child
12. return node.myWord

* Complexity analysis:

|  |  |
| --- | --- |
| Step # | Complexity stated as O(\_) |
| 1 | O(n) |
| 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(1) |
| 11 | O(1) |
| 12 | O(1) |

Complexity of the algorithm = O(n2)

*topMatches*

* Pseudocode:

1. PriorityQueue nodeList = new PriorityQueue
2. ArrayList<String> wordsList = new ArrayList
3. for i = 1 to prefixArray.length
4. if (node.children.contains(i))
5. node = node.getChild(i)
6. else
7. return wordsList
8. nodeList.add(node)
9. while (nodeList.size() > 0)
10. node = nodeList.poll()
11. if (node.isWord)
12. wordsList.add(node.myWord)
13. if (wordsList.size() >= k)
14. break
15. nodeList.addAll(node.children)
16. return wordsList

* Complexity analysis:

|  |  |
| --- | --- |
| Step # | Complexity stated as O(\_) |
| 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(n2) |
| 10 | O(1) |
| 11 | O(1) |
| 12 | O(1) |
| 13 | O(1) |
| 14 | O(1) |
| 15 | O(1) |
| 16 | O(1) |

Complexity of the algorithm = O(n2)

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

Using my AutocompleteTest.java I have completed and tested all the cases below. 18 passed test cases.

**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:

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?

The Autocompletor data structure contains the method weightOf with having a big-Oh time complexity of O(n). This is due to the weightOf method looking through n words to find the weight of the specified word being searched. TopMatch and TopMatches share the same time complexity which is O(n2). This is due to both methods having the same nature of looking at ArrayLists and nodes at least twice as much as the weightOf function. This displays that TopMatch and TopMatches must have a big-Oh time complexity of O(n2). With our three methods, the order of growth starts with weightOf then goes to TopMatch and TopMatches.

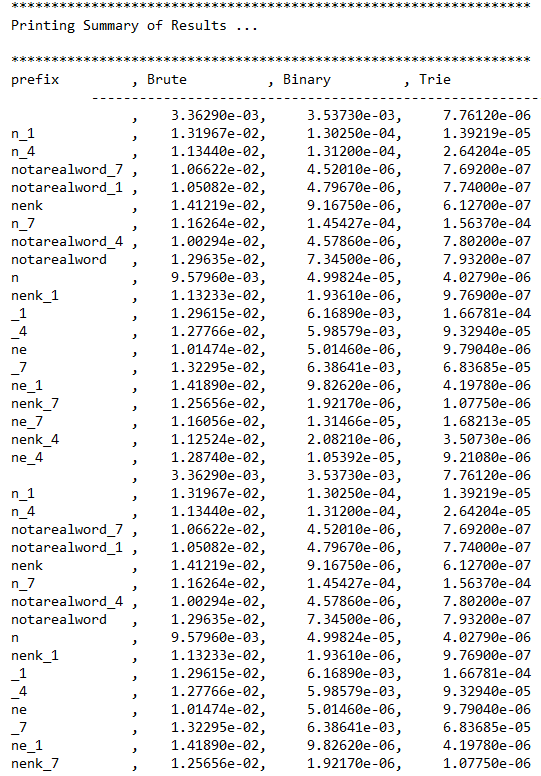
1. How does the runtime of *topMatches()* vary with k, assuming a fixed prefix and set of terms? Provide answers for *BruteAutocomplete* and *TrieAutocomplete*. Justify your answer, with both data and algorithmic analysis.

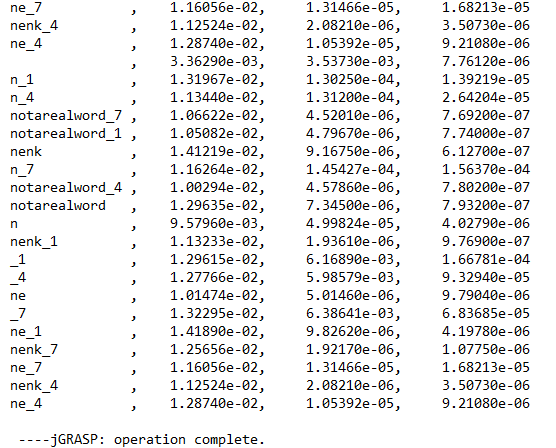
BruteAutocomplete: k does not effect the runtime because it runs through the entirety of the word list. Brute takes a very big array and looks through the whole array iteratively. When it is done doing this (array is full), it compares the element to the first element in the array because the first element is already sorted. This repeats until it runs through the entire word list.

TrieAutocomplete: k has a direct effect on the runtime of Trie. The greater k is, the greater the runtime will be. Trie uses k to determine how many extra elements it will look for under execution. This will make the runtime longer and make the iteration of the for loop longer because of the extra elements.

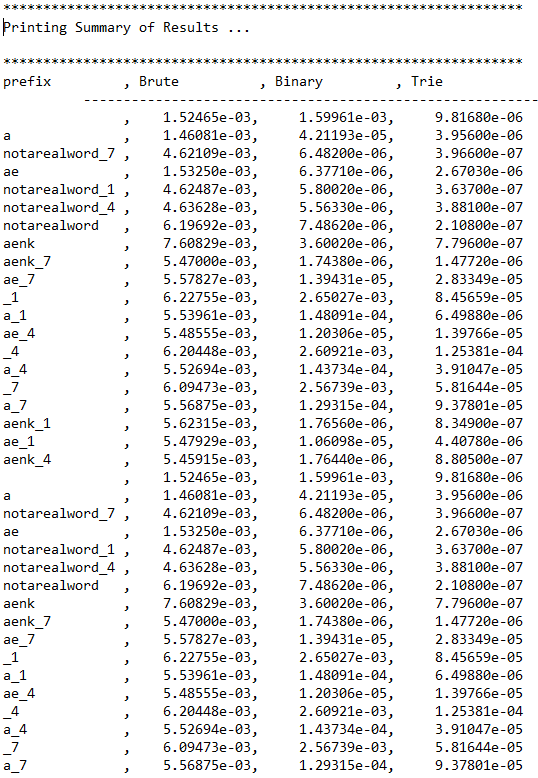
1. How does increasing the size of the source and increasing the size of the prefix argument affect the runtime of *topMatch* and *topMatches*? (Tip: Benchmark each implementation using fourletterwords.txt, which has all four-letter combinations from aaaa to zzzz, and fourletterwordshalf.txt, which has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

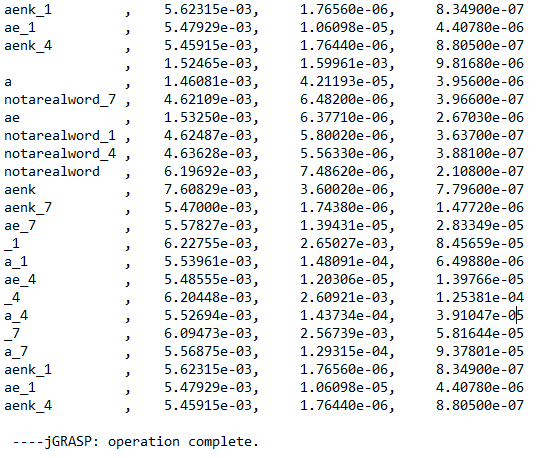
fourletterwords.txt output-





fourletterwordshalf.txt output





Answer to iii:

Based on the data above, fourletterwords.txt and fourletterwordshalf.txt show that by increasing the prefix and increasing the source size helps TrieAutocomplete to make the word matches faster. The bigger the size of the prefix the less amount of time it takes to find the match. The data above proves this conclusion and the differences in the runtimes are negligible.

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.

After analyzing the pseudocode for TrieAutoComplete, I found that the big-Oh time complexity is O(n2). The graph below shows differences in runtime for TrieAutoComplete. These differences are within milliseconds of each other and their differences are negligible. After implementing the TrieAutoComplete, it is shown by the graph that it has more of a O(1) big-Oh time complexity (can be seen clearer by the graph comparing brute, binary and trie). This is the key difference between my pseudo code analysis and after the implementation.

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

TrieAutoComplete is the best and most efficient with BinarySearchAutoComplete being the second most efficient consistently. BruteAutoComplete is the least efficient with the worst average runtime.