COMP 3270 SPRING 2020

**Programming Project: Autocomplete**

Name: Elizabeth Dayton Date Submitted: 4/5/2020

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 if word = null return “Word cannot be null”

2 if weight < 0 return “Weight cannot be less than zero”

3 currentNode = myRoot

4 for (each letter in word)

5 if weight > currentNode’s subtree max weight

6 currentNode’s subtree max weight = weight

7 child = currentNode.getChild(letter)

8 if child = null

9 child = new Node(letter, currentNode, weight)

10 currentNode.children.put(letter, child)

11 currentNode = child

12 currentNode.setWord(word)

13 currentNode.setWeight(weight)

14 currentNode.isWord = true

* Complexity analysis:

|  |  |
| --- | --- |
| Step # | Complexity stated as O(\_) |
| 1 | O(1) |
| 2 | O(1) |
| 3 | O(1) |
| 4 | O(n) |
| 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) |

Complexity of the algorithm = O(n)

*topMatch*

* Pseudocode:

1 if prefix = null return “Prefix cannot be null”

2 ArrayList nodes = new ArrayList

3 currentNode = myRoot

4 for (each letter in prefix)

5 if(currentNode’s children does not contain letter)

6 return “”

7 currentNode = currentNode.getChild(letter)

8 while (currentNode is not a word and currentNode’s weight does not equal subtree’s max weight)

9 for (each child in currentNode’s children)

10 if (child’s subtree max weight = currentNode’s

subtree max weight)

11 currentNode = child

12 break

13 return currentNode.getWord

* Complexity analysis:

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

Complexity of the algorithm = O(n2 + n)

*topMatches*

* Pseudocode:

1 if prefix = null return “Prefix cannot be null”

2 currentNode = myRoot

3 List matches = new ArrayList

4 PriorityQueue nodes = new PriorityQueue

5 for (each letter in prefix)

6 if currentNode.getChild(letter) = null

7 return empty ArrayList

8 currentNode = currentNode.getChild(letter)

9 if (currentNode is a word)

10 matches.add(new Term(currentNode’s word, currentNode’s weight))

11 for (each child in currentNode’s children)

12 if (child does not equal null) nodes.offer(child)

13 while (nodes.size > 0)

14 currentNode = nodes.poll

15 if(currentNode is a word)

16 matches.add(new Term(currentNode’s word, currentNode’s weight))

17 for (each child in currentNode’s children)

18 if (child does not equal null) nodes.offer(child)

19 matches.sort(terms in reverse order)

20 ArrayList kmatches = new ArrayList

21 if (k > matches.size())

22 kMatches.addAll(matches)

23 else

24 kMatches.addAll(matches.sublist(0, k))

25 ArrayList topKMatches = new ArrayList

26 for (each term in kMatches)

27 topKMatches.add(term’s word)

28 return topKMatches

* Complexity analysis:

|  |  |
| --- | --- |
| Step # | Complexity stated as O(\_) |
| 1 | O(1) |
| 2 | O(1) |
| 3 | O(1) |
| 4 | O(1) |
| 5 | O(n) |
| 6 | O(1) |
| 7 | O(1) |
| 8 | O(1) |
| 9 | O(1) |
| 10 | O(1) |
| 11 | O(n) |
| 12 | O(1) |
| 13 | O(n) |
| 14 | O(1) |
| 15 | O(1) |
| 16 | O(1) |
| 17 | O(n) |
| 18 | O(1) |
| 19 | O(1) |
| 20 | O(1) |
| 21 | O(1) |
| 22 | O(1) |
| 23 | O(1) |
| 24 | O(1) |
| 25 | O(1) |
| 26 | O(n) |
| 27 | O(1) |
| 28 | O(1) |

Complexity of the algorithm = O(n3 + 2n)

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:

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 big-Oh in the worst case for add is n, for topMatch n2, and for topMatches n3.

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’s runtime varies slightly with a varying k. When the prefix is relatively short, the runtime gets faster as k gets larger. When the prefix is longer, the runtime gets only slightly slower as k gets larger. For TrieAutocomplete, the runtime is much slower than even BruteAutocomplete when the prefix is very small, but does get slightly faster as k gets larger. When the prefix is large it is substantially faster, and gets slightly faster as k gets larger.

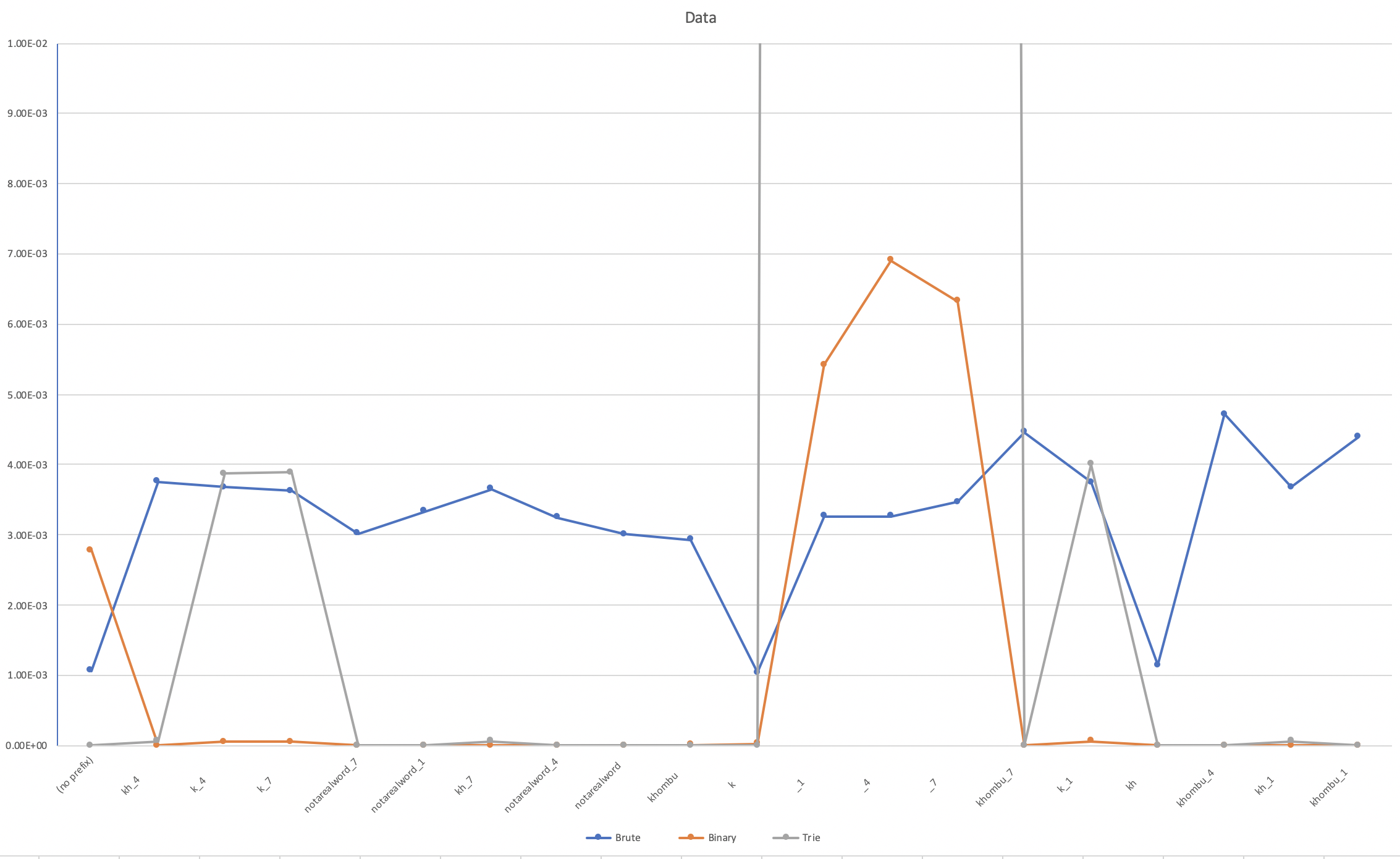
1. How does increasing the size of the source and increasing the size of the prefix argument affect the runtime of *topMatch* and *topMatches*? (Tip: Benchmark each implementation using fourletterwords.txt, which has all four-letter combinations from aaaa to zzzz, and fourletterwordshalf.txt, which has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

For TrieAutocomplete, increasing the size of the source makes the majority of the runtimes slower. When the source is half, everything is much faster except for when the prefix is very small or is just an empty string. For BinaryAutocomplete, it is only very slightly slower for the increased size source and when the prefix is very small. For BruteAutocomplete, it almost stays the same, except for when prefix is small. When prefix is small it’s actually slightly faster for the larger source.

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.

The big-Oh after analyzing the pseudocode was O(n3). After implementation is still appears to be O(n3).

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

Trie is incredibly slow when the prefix entered is small or empty. This is probably because my implementation adds every word in the tree under the prefix given to a list and then sorts it after collecting them all.