COMP 3270 FALL 2021

Programming Project: Autocomplete

Name: **Mary Mitchell** Date Submitted: **29 October 2021**

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:
  // word is of the string type and weight is a double type
- ADD(word, weight)

```
1. if weight < 0 then error
2. if word = NIL then error
3. pointer = root of trie
4. index = 1
5. Let W[1…word.length] be a new array of the chars from word
6.  while (index <= W.length)
7.      key = W[index]
8.      if (weight > pointer's subtreeMaxWeight)
9.      //subtreeMaxWeight is an attribute of a node
10.             pointer's subtreeMaxWeight = weight
11.     if (pointer.child(key) = NIL)
12.             create a new child node for pointer with key value
13.     pointer = pointer.child(key)
14.     index = index + 1
15.     if (index = W.length)
16.             set pointer's word
17.             set pointer is a word
18.             set pointer's weight
```

- Complexity analysis:

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

7	O(n)
8	O(n)
9	Ignore comment
10	O(n)
11	O(n)
12	O(n)
13	O(n)
14	O(n)
15	O(n)
16	O(n)
17	O(n)
18	O(n)

Complexity of the algorithm = O(n)

topMatch

- Pseudocode:
  // prefix is a string
- TOPMATCH(prefix)

```
1. if prefix = NIL then error
2. pointer = root of trie
3. index = 1
4. Let W[1…prefix.length] be a new array of the chars from prefix
5. while (index <= W.length)
6.     key = W[index]
7.     if (pointer.child(key) ≠ NIL)
8.         pointer = pointer.child(key)
9.     else
10.        return " "
11.    index = index + 1
12. create a new max-priority queue PQ for Nodes sorted by
    subtreeMaxWeight
13. wordFound = false
14. output = " "
15. while (wordFound)
16.     if (pointer is a word)
17.         if (pointer's weight = pointer's maxSubtreeWeight)
18.             wordFound = true
19.             ouput = pointer's word
20.         else wordFound = false
21.     for all of pointer's children
22.         add them to PQ
23.     pointer = PQ.head
24. return output
```

- Complexity analysis:

Step #	Complexity stated as O( )
1	O(1)
2	O(1)
3	O(1)
4	O(1)
5	O(n)
6	O(n)
7	O(n)
8	O(n)
9	O(n)
10	O(n)
11	O(n)
12	O(1)
13	O(1)
14	O(1)
15	O(n)
16	O(n)
17	O(n)
18	O(n)
19	O(n)
20	O(n)
21	$O(n^2)$
22	$O(n^2)$
23	O(n)
24	O(1)

Complexity of the algorithm = $O(n^2)$

topMatches

- Pseudocode:
  // prefix is of the string type and k is an integer type
- TOPMATCH(prefix, k)

```
1. if prefix = NIL then error
2. Let N be a new array list of strings that is empty
3. if k <= 0 then return N
4. pointer = root of trie
5. index = 1
6. Let W[1…prefix.length] be a new array of chars from prefix
7. while (index <= W.length)
8.        key = W[index]
9.        if (pointer.child(key) ≠ NIL)
10.               pointer = pointer.child(key)
```

```
11.     else
12.             return N
13.     index = index + 1
14. create a new max-priority queue PQ for Nodes sorted by
    subtreeMaxWeight
15. create a new array list TN of nodes that is empty
16. create a new array list OUT of strings that is empty
17. add pointer to PQ
18. while (PQ.size > 0)
19.     if (TN.size = k)
20              sort TN //ascending order
21.             firstNode = TN's first node
22.             secondNode = PQ.head
23.             if (firstNode's weight >  secondNode)
24.                     break
25.     pointer = PQ.head
26.     for all of pointer's children
27.             add child to PQ
28.     if (pointer is a word)
29.             add pointer to TN
30. sort TN in descending order
31. if (TN.size >= k)
32.     for i = 1 to k
33.             add TN's $i^{th}$ node's word to OUT
34. else
35.     for each node in TN
36              add the node's word to OUT
37. return OUT
```

- 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(n)
8	O(n)
9	O(n)
10	O(n)
11	O(n)
12	O(n)
13	O(n)
14	O(1)
15	O(1)

16	O(1)
17	O(1)
18	O(n)
19	O(n)
20	O(n^2)
21	O(n)
22	O(n)
23	O(n)
24	O(n)
25	O(n)
26	O(n^2)
27	O(n^2)
28	O(n)
29	O(n)
30	O(n)
31	O(1)
32	O(n)
33	O(n)
34	O(1)
35	O(n)
36	O(n)
37	O(1)

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?

   For the add operation the order of growth is O(n). This is because at most it will have to look at n characters in a word and add them to the trie.

   For the topMatches operation the order of growth is O(n^2). This is because, for every node looked at in a subtree, each of its children nodes will also have to be looked at. So if you have n nodes to look at with n children each, the cost will be O(n^2)

   The topMatch operation performs like topMatches with a growth of O(n^2). (topMatch and topMatches have very similar algorithms)

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.

   Based on the benchmark data in the graph below (Figure 1) TrieAutoComplete performs slower as k increases. However, this is only when the prefix is a real word. When prefix is not a real word the algorithm detects it on the traversal to the prefix node and never enters the part of the program that looks for the k top words. When prefix is a real word it takes longer to find the k top words when k is larger because the algorithm will generally have to run its loop for a longer time. However, compared to Brutes benchmark data, Trie will still perform better with a higher k value.

   BruteAutoComplete's benchmark data (Figure 2) suggests that it is not as affected by k as TrieAutoComplete is. This is because BruteAutoComplete goes through every term it has no matter what the k value is (it's for each loop does not break until every word is looked at). It's time cost is also much higher than TrieAutoComplete's in every case.

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.)

Based on the results from the benchmark analysis, there were not many distinct time differences between fourletterwords.txt and fourletterwordshalf.txt. As far as increasing the prefix, it shouldn't have too much of an effect on time cost as opposed to the effect the k value would have. This is because the size of the prefix is only important in so far as navigating to the prefix node only requires a growth of O(n) compared to the second part of the algorithm (the part that finds the k words) which has growth of O(n^2). Additionally, increasing the input size usually results in much higher time cost. However, based on the benchmark data there was not much of a difference between the results of the two text files.

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.
The big-Oh for the pseudocode and the actual implementation are the same.

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

Words-333333.txt(Figures 3 and 4): For the most part TrieAutoComplete is comparable in time cost to BruteAutoComplete and BinarySearchAutoComplete. There are several instances where it performs faster than either Brute or Binary (e.g. when anotrealword is passed). However, when whitespace is passed as a parameter for topMatches, TrieAutoComplete performs significantly slower than the other two.
Fourletterwords.txt (Firgure 5): Unlike with words-333333, with four letter words TrieAutoComplete performs better than both BruteAutoComplete and BinaryAutoComplete in terms of time cost. BruteAutoComplete performs the worst out of the three, with Binary and Trie being the most comparable to each other.

GRAPHS

Figure 1:
Trie Benchmark fourletterwords
(also includes topMatch data)

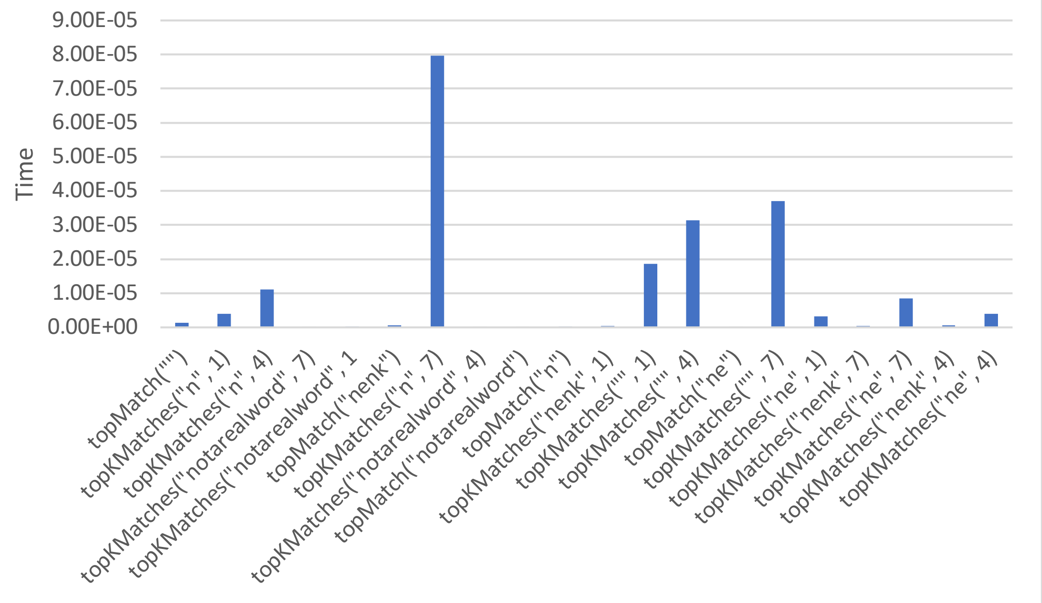


Figure 2:
Brute Benchmark fourletterwords
(also includes topMatch data)

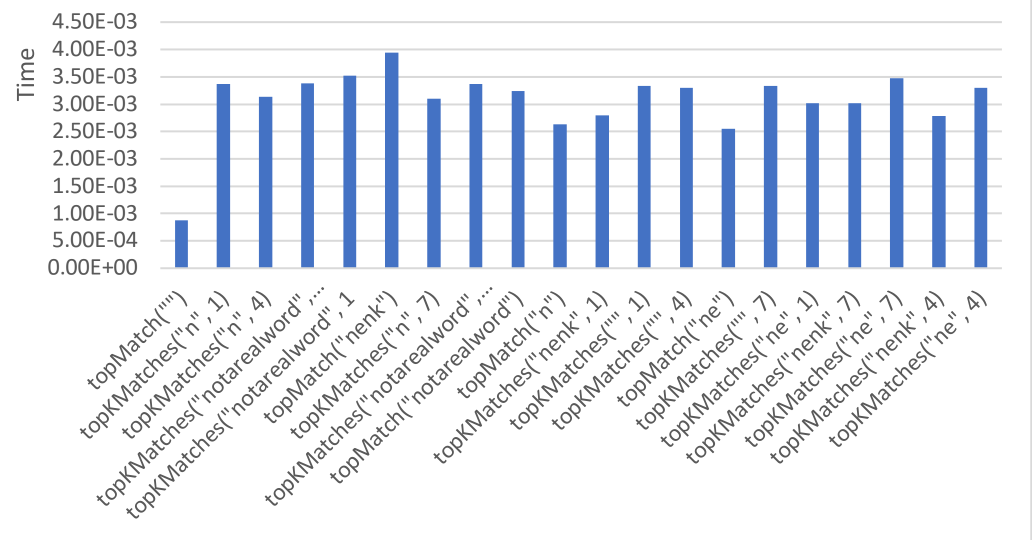


Figure 3:
Benchmark for Brute, Binary, and Trie with words-333333

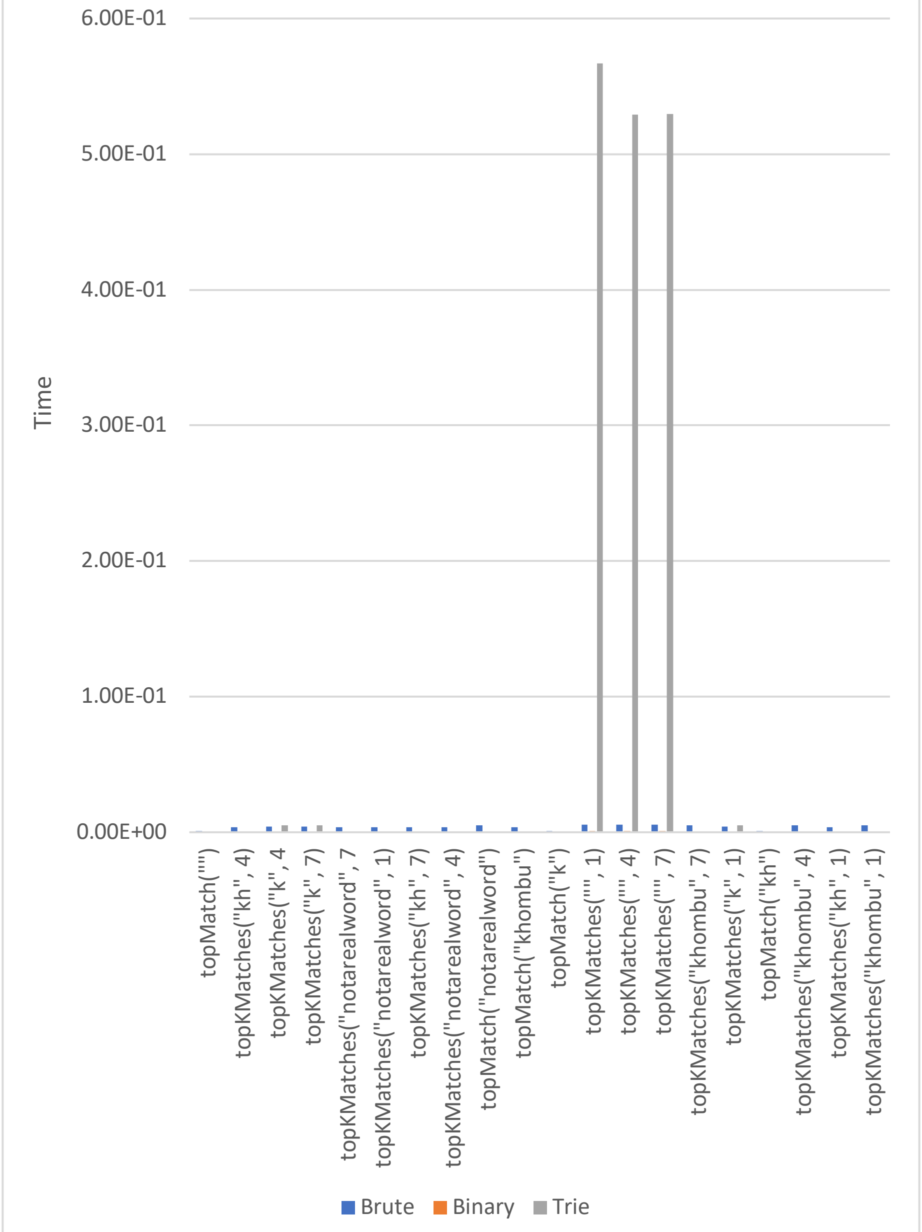


Figure 4:
Without whitespace parameters
words-333333

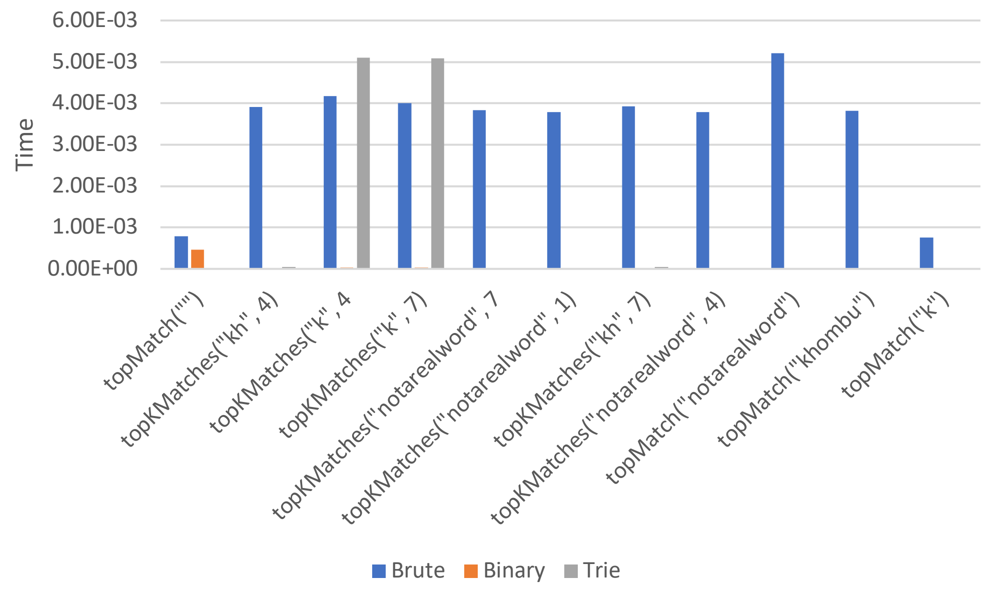
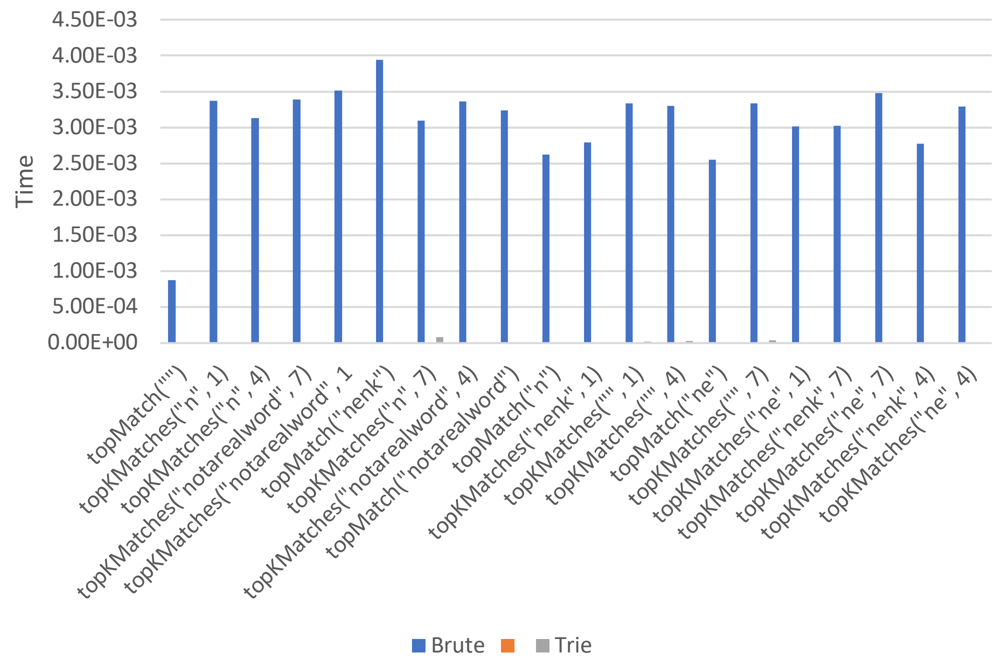


Figure 5:
Benchmark for Brute, Binary, and Trie
(fourletterwords)


Time
4.50E-03
4.00E-03
3.50E-03
3.00E-03
2.50E-03
2.00E-03
1.50E-03
1.00E-03
5.00E-04
0.00E+00
topMatch("")
topKMatches("n", 1)
topKMatches("n", 4)
topKMatches("notarealword", 7)
topKMatches("notarealword", 1
topMatch("nenk")
topKMatches("n", 7)
topKMatches("notarealword", 4)
topMatch("notarealword")
topMatch("n")
topKMatches("nenk", 1)
topKMatches("", 1)
topKMatches("", 4)
topMatch("ne")
topKMatches("", 7)
topKMatches("ne", 1)
topKMatches("nenk", 7)
topKMatches("ne", 7)
topKMatches("nenk", 4)
topKMatches("ne", 4)
Brute
Trie