Autocomplete Analysis

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 Autocomplete data type make, as a function of the number of terms N, the number of matching terms M, and k, the number of matches returned by topKMatches for BinarySearchAutocomplete?

   For topKMatches, the order of growth of the number of compares should be O(log N + M log M).

```
int first = firstIndexOf(myTerms, new Term(prefix, 0), new
Term.PrefixOrder(prefix.length()));
        if(first == -1)
                return new String[0];
                int last = lastIndexOf(myTerms, new Term(prefix, 0), new

Term.PrefixOrder(prefix.length()));

Term[] match = Arrays.copyOfRange(myTerms, first, last+1);
Arrays.sort(match, new Term.ReverseWeightOrder());

int numResults = Math.min(k, last-first+1);
```

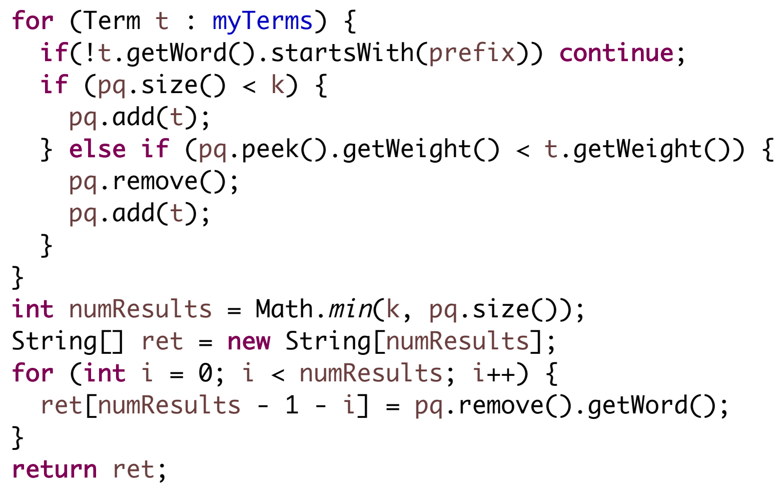
   In topKMatches, firstIndexOf and lastIndexOf are called. For each of them, binary search takes O(log N), because it halves the input size on every iteration. Then, the M matching terms are sorted, which takes O(M log M). Then, it take O(1) to compare k and M. So in total, it takes O(log N + M log M) compares.

   For topMatch, the order of growth of the number of compares should be O(log N + M).

```
int first = firstIndexOf(myTerms, new Term(prefix, 0), new
Term.PrefixOrder(prefix.length()));
if(first == -1)
        return "";
int last = lastIndexOf(myTerms, new Term(prefix, 0), new
Term.PrefixOrder(prefix.length()));
int max = first;
        for(int i = first+1; i <= last; i++){
                if(myTerms[i].getWeight() > myTerms[max].getWeight())
                        max = i;
        }
```

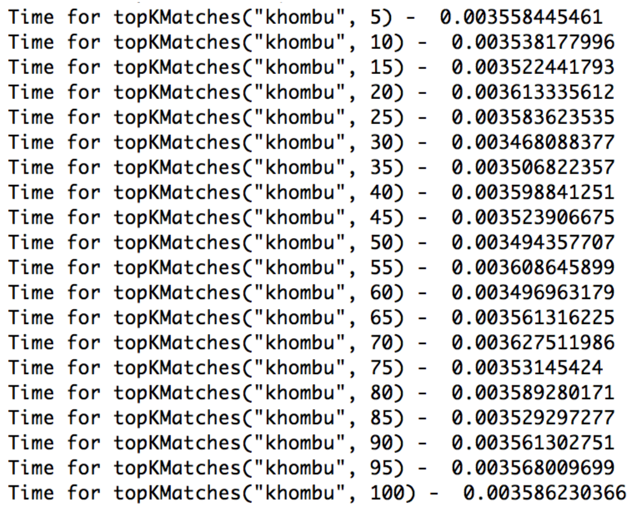
   In topMatch, firstIndexOf and lastIndexOf are called. For each of them, binary search takes O(log N), because it halves the input size on every iteration. Then, from the M matching terms, we want to find the term with the largest weight. So it will iterate from the first term to the last term and compares weight with the temporary largest weight, which takes O(M). So in total, it takes O(log N + M) compares.

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

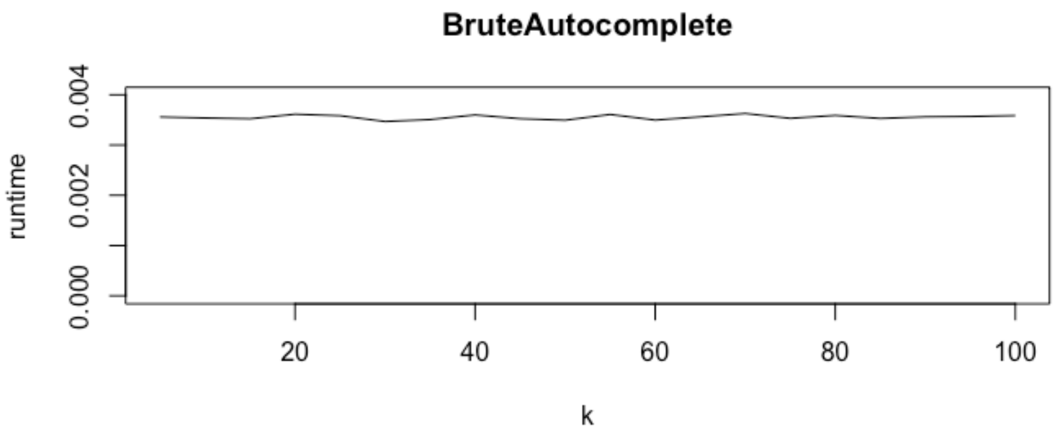


```
for (Term t : myTerms) {
  if(!t.getWord().startsWith(prefix)) continue;
  if (pq.size() < k) {
    pq.add(t);
  } else if (pq.peek().getWeight() < t.getWeight()) {
    pq.remove();
    pq.add(t);
  }
}
int numResults = Math.min(k, pq.size());
String[] ret = new String[numResults];
for (int i = 0; i < numResults; i++) {
  ret[numResults - 1 - i] = pq.remove().getWord();
}
return ret;
```

<u>**For BruteAutocomplete**</u>, in the procedure of adding terms with input prefix to the priority queue, when M>k, for the first k terms with input prefix, it only calls pq.add(), so it takes k times. For the rest M-k matching terms, first it calls pq.remove() to remove the least-weight term in pq, and then calls pq.add() to add the new term to pq, which takes 2(M-k) times. For returning the top k matches, it takes k times. So in total, the runtime of topKMatches() approximately does not have any relationship with k. When M<k, the adding process takes M times, and the returning process takes M times. The whole process has nothing to do with k.



```
Time for topKMatches("khombu", 5) -  0.003558445461
Time for topKMatches("khombu", 10) -  0.003538177996
Time for topKMatches("khombu", 15) -  0.003522441793
Time for topKMatches("khombu", 20) -  0.003613335612
Time for topKMatches("khombu", 25) -  0.003583623535
Time for topKMatches("khombu", 30) -  0.003468088377
Time for topKMatches("khombu", 35) -  0.003506822357
Time for topKMatches("khombu", 40) -  0.003598841251
Time for topKMatches("khombu", 45) -  0.003523906675
Time for topKMatches("khombu", 50) -  0.003494357707
Time for topKMatches("khombu", 55) -  0.003608645899
Time for topKMatches("khombu", 60) -  0.003496963179
Time for topKMatches("khombu", 65) -  0.003561316225
Time for topKMatches("khombu", 70) -  0.003627511986
Time for topKMatches("khombu", 75) -  0.00353145424
Time for topKMatches("khombu", 80) -  0.003589280171
Time for topKMatches("khombu", 85) -  0.003529297277
Time for topKMatches("khombu", 90) -  0.003561302751
Time for topKMatches("khombu", 95) -  0.003568009699
Time for topKMatches("khombu", 100) -  0.003586230366
```



Coefficients:

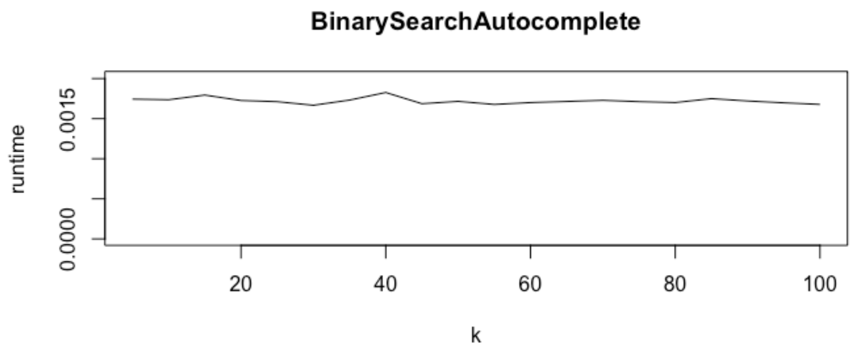
```
             Estimate Std. Error t value Pr(>|t|)
(Intercept) 3.539e-03  2.072e-05 170.817   <2e-16 ***
k           2.780e-07  3.459e-07   0.804    0.432
---
Signif. codes:  0 ‘***’ 0.001 ‘**’ 0.01 ‘*’ 0.05 ‘.’ 0.1 ‘ ’ 1

Residual standard error: 4.46e-05 on 18 degrees of freedom
Multiple R-squared:  0.03464,	Adjusted R-squared:  -0.01899
F-statistic: 0.646 on 1 and 18 DF,  p-value: 0.432
```

From the data, we can tell that the runtime doesn’t vary much when k changes from 1 to 7. And the p-value for T-test is 0.432, so it fails the significance test. We can conclude that the runtime of topKMatches() approximately does not have any relationship with k.

```java
int numResults = Math.min(k, last-first+1);
String[] ret = new String[numResults];
for (int i = 0; i < numResults; i++) {
     ret[i] = match[i].getWord();
}
return ret;
```

For BinarySearchAutocomplete, if the number of matching terms M is less than k, it will return M matching terms, so the runtime for that operation should be O(M). If the number of matching terms M is no less than k, it will return k matching terms, so the runtime for that operation should be O(k). So the runtime of topKMatches() has linear relationship with k only when M is no less than k.

```
Time for topKMatches("k", 5) -  0.00174334816
Time for topKMatches("k", 10) -  0.001736013235
Time for topKMatches("k", 15) -  0.001794133586
Time for topKMatches("k", 20) -  0.001726397706
Time for topKMatches("k", 25) -  0.001711732313
Time for topKMatches("k", 30) -  0.001667038386
Time for topKMatches("k", 35) -  0.001731738504
Time for topKMatches("k", 40) -  0.001826132168
Time for topKMatches("k", 45) -  0.001686414506
Time for topKMatches("k", 50) -  0.001715682705
Time for topKMatches("k", 55) -  0.001677278102
Time for topKMatches("k", 60) -  0.001700733716
Time for topKMatches("k", 65) -  0.001714383872
Time for topKMatches("k", 70) -  0.001728518986
Time for topKMatches("k", 75) -  0.001712105607
Time for topKMatches("k", 80) -  0.001700953786
Time for topKMatches("k", 85) -  0.001749584841
Time for topKMatches("k", 90) -  0.001720420334
Time for topKMatches("k", 95) -  0.001697530781
Time for topKMatches("k", 100) -  0.001677575236
```

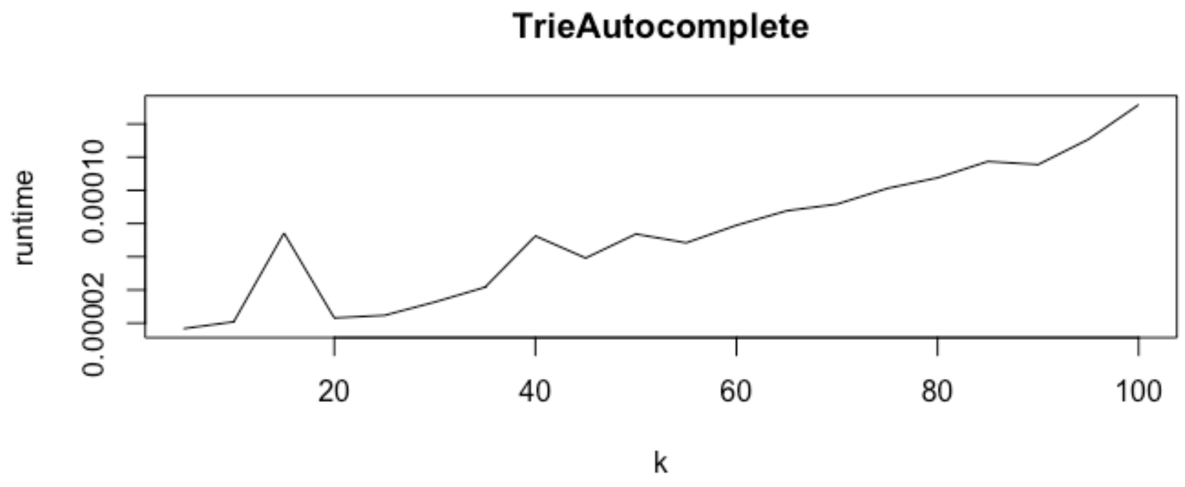


```
Coefficients:
              Estimate Std. Error t value Pr(>|t|)
(Intercept)  1.745e-03  1.706e-05 102.272   <2e-16 ***
k           -4.508e-07  2.848e-07  -1.583    0.131
---
Signif. codes:  0 ‘***’ 0.001 ‘**’ 0.01 ‘*’ 0.05 ‘.’ 0.1 ‘ ’ 1

Residual standard error: 3.672e-05 on 18 degrees of freedom
Multiple R-squared:  0.1222,  Adjusted R-squared:  0.07342
F-statistic: 2.506 on 1 and 18 DF,  p-value: 0.1309
```

From the data, we can tell that the runtime doesn’t vary much when k changes from 5 to 100. And the p-value for T-test is 0.1309, so it fails the significance test. We can conclude that the runtime of topKMatches() approximately does not have any relationship with k.

```
        //navigate current to prefix node
        while(!pq.isEmpty()){
                if(words.size()==k && words.peek().getWeight() >
pq.peek().mySubtreeMaxWeight)
                        break;
                current = pq.poll();
                if(current.isWord)
                        if(words.size()<k)
                                words.add(current);
                        else if(current.getWeight() > words.peek().getWeight()){
                                        words.remove();
                                        words.add(current);
                                }
                for(Node child : current.children.values()){
                        pq.add(child);
                }
        }
        int num = words.size();
        String[] result = new String[num];
        for(int i=0; i<num; i++)
                result[num-i-1] = words.poll().getWord();
        return result;
        }
```

<u>For TrieAutocomplete</u>, it will add k Nodes to the priority queue words, and then replace the Node with the lowest weight with the highest-weight Node in the priority queue pq. The adding process takes O(log K).



lm(formula = log(runtime) ~ k)

```
Residuals:
     Min       1Q   Median       3Q      Max
-0.37109 -0.15500 -0.01896  0.08425  0.93278

Coefficients:
             Estimate Std. Error t value Pr(>|t|)
(Intercept) -10.744616   0.139523 -77.010  < 2e-16 ***
k             0.020241   0.002329   8.689 7.41e-08 ***
---
Signif. codes:  0 ‘***’ 0.001 ‘**’ 0.01 ‘*’ 0.05 ‘.’ 0.1 ‘ ’ 1

Residual standard error: 0.3004 on 18 degrees of freedom
Multiple R-squared:  0.8075,	Adjusted R-squared:  0.7968
F-statistic:  75.5 on 1 and 18 DF,  p-value: 7.41e-08
```

It seems that the runtime is propotional to log k.

3. Look at the methods topMatch and topKMatches in BruteAutocomplete and BinarySearchAutocomplete and compare both their theoretical and empirical runtimes. Is BinarySearchAutocomplete always guaranteed to perform better than BruteAutocomplete? Justify your answer.

topMatch

BruteAutocomplete

```
String maxTerm = "";
double maxWeight = -1;
for (Term t : myTerms) {
  if (t.getWeight() > maxWeight && t.getWord().startsWith(prefix)) {
    maxTerm = t.getWord();
    maxWeight = t.getWeight();
  }
}
return maxTerm;
     }
```

In the outer loop, t iterates through myTerms, which is O(N). In the loop, the if statement has two conditions. The first one takes O(1) and the second one takes O(prefix.length). In total, the runtime is O(N * (1+prefix.length)).

BinarySearchAutocomplete

```
public String topMatch(String prefix) {
		if(prefix == null)
			throw new NullPointerException("prefix is null");
		int first = firstIndexOf(myTerms, new Term(prefix, 0), new Term.PrefixOrder(prefix.length()));
		if(first == -1)
			return "";
```

```
        int last = lastIndexOf(myTerms, new Term(prefix, 0), new Term.PrefixOrder(prefix.length()));
        int max = first;
        for(int i = first+1; i <= last; i++){
                if(myTerms[i].getWeight() > myTerms[max].getWeight())
                        max = i;
        }
        return myTerms[max].getWord();
        }
```

It takes O(prefix.length * log N + M), since finding the matching terms takes O(prefix.length * log N) and finding the matching term with the largest weight takes O(M).

When the length of the prefix is very short, for example, 0, we can expect a very large M=N. In this case, BruteAutocomplete might be more efficient. Otherwise, BinarySearchAutocomplete is more efficient.

Empirical data:

Prefix	BruteAutocomplete	BinarySearchAutocomplete
""	6.53461256E-4	0.003641022265
"khombu"	0.002577289103	7.819324E-6
"k"	0.001167429167	3.1245164E-5
"kh"	0.001101694534	1.962739E-6
"notarealword"	0.00409118909	6.82366E-7

From the data, we can tell that BruteAutocomplete takes less time than BinarySearchAutocomplete only when the length of the prefix is 0, which matches my expectation.

<u>topKMatches</u>

BruteAutocomplete

```
public String[] topKMatches(String prefix, int k) {
  // maintain pq of size k
  PriorityQueue<Term> pq = new PriorityQueue<Term>(k, new Term.WeightOrder());
  for (Term t : myTerms) {
    if(!t.getWord().startsWith(prefix)) continue;
    if (pq.size() < k) {
      pq.add(t);
    } else if (pq.peek().getWeight() < t.getWeight()) {
      pq.remove();
      pq.add(t);
    }
  }
  int numResults = Math.min(k, pq.size());
  String[] ret = new String[numResults];
  for (int i = 0; i < numResults; i++) {
        ret[numResults - 1 - i] = pq.remove().getWord();
      }
      return ret;
         }
```

In the outer loop, t iterates through myTerms, which is O(N). In the loop, the first if statement takes O(prefix.length). The second if statement takes O(1). For the rest part of the code, there is no difference between BruteAutocomplete and BinarySearchAutocomplete. In total, it takes O(N(1+prefix.length)).

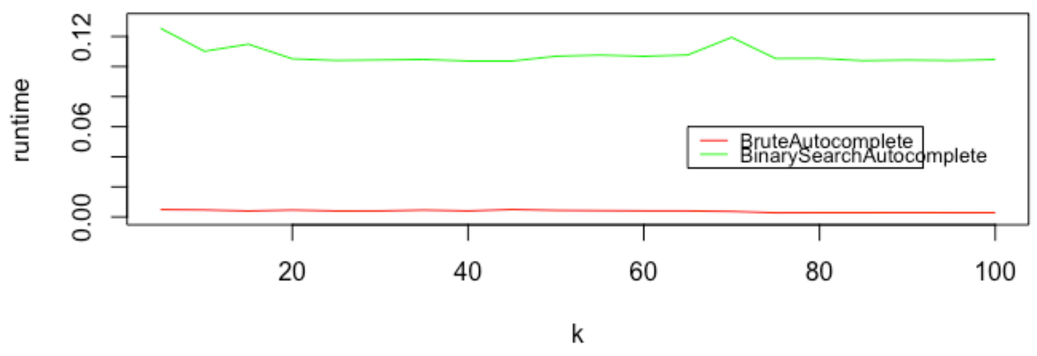
BinarySearchAutocomplete

```
public String[] topKMatches(String prefix, int k) {
        if(prefix == null)
              throw new NullPointerException("prefix is null");
        int first = firstIndexOf(myTerms, new Term(prefix, 0), new
Term.PrefixOrder(prefix.length()));
        if(first == -1)
              return new String[0];
        int last = lastIndexOf(myTerms, new Term(prefix, 0), new
Term.PrefixOrder(prefix.length()));
        Term[] match = Arrays.copyOfRange(myTerms, first, last+1);
        Arrays.sort(match, new Term.ReverseWeightOrder());
      int numResults = Math.min(k, last-first+1);
      String[] ret = new String[numResults];
      for (int i = 0; i < numResults; i++) {
        ret[i] = match[i].getWord();
      }
      return ret;
       }
```

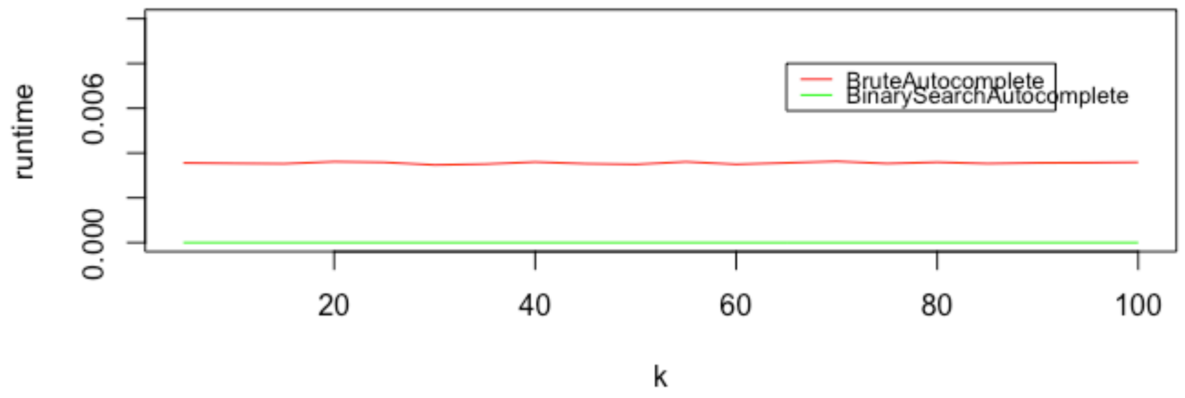
It takes O(prefix.length * log N + M log M), since finding the matching terms takes O(prefix.length * log N) and sorting the matching terms takes O(M log M). Similarly, when the length of the prefix is very short, for example, 0, we can expect a very large M=N. In this case, BruteAutocomplete might be more efficient. Otherwise, BinarySearchAutocomplete is more efficient.

Empirical data:

Prefix = “”



Prefix = “khombu”



From the data, when prefix = “”, BruteAutocomplete is more efficient. When prefix = “khombu”, BinarySearchAutocomplete is more efficient. The results match my expectation.

4. For all three of the Autocompletor implementations, how does increasing the size of the source and increasing the size of the prefix argument affect the runtime of topMatch and topKMatches? (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.)

1) <u>increasing the size of the source</u>

<u>topMatch</u>

prefix = “`notarealword`”

	BruteAutocomplete	BinarySearchAutocomplete	TrieAutocomplete

fourletterwordshalf.txt	0.002414785571	1.543414E-6	1.2E-7
fourletterwords.txt	0.00264156244	2.092428E-6	5.25505E-7

The runtime for topMatch increases when increasing the size of the source.

topKMatches

prefix = “`notarealword`”, k=4

	BruteAutocomplete	BinarySearchAutocomplete	TrieAutocomplete
fourletterwordshalf.txt	0.002877801171	4.51067E-7	1.20333E-7
fourletterwords.txt	0.003582513818	7.43779E-7	3.0926E-7

The runtime for topKMatches increases when increasing the size of the source.

2) **increasing the size of the prefix**

topMatch

	BruteAutocomplete	BinarySearchAutocomplete	TrieAutocomplete
Prefix.length=1	0.003389468409	4.059709E-5	2.038235E-6
Prefix.length=4	0.004960148401	3.829967E-6	9.96352E-7

For BruteAutocomplete, runtime for topMatch increases when increasing the size of the prefix. For BinaryAutocomplete and TrieAutocomplete, runtime for topMatch decreases when increasing the size of the prefix.

topKMatches

k=4

	BruteAutocomplete	BinarySearchAutocomplete	TrieAutocomplete

Prefix.length=1	0.004480471068	0.003319430559	1.1782226E-5
Prefix.length=4	0.003881280883	8.87057E-7	7.35111E-7

For all of them, the runtime for topKMatches decreases when increasing the size of the prefix.