AUTOCOMPLETE ANALYSIS:

1.
BINARYSEARCH TOP-MATCH AUTOCOMPLETE:
The operations that seems to take most time in this method (and hence the leading term) is Term[] newTerms = Arrays.*copyOfRange*(myTerms, first, last + 1); which uses System.arrayCopy, which is an O(n) operation, where n represents the size of matching terms. This particular method thus takes O(n) time, where n is the number of items in Terms that start with prefix (matching terms). The other operations are not leading terms hence we cannot say that they determine the run time complexity of this method.

TOPKMATCHES:
The same can be said for my TopKmatches. However, I use Arrays.*sort*(newterms, **new** Term.ReverseWeightOrder());, which is a O(nlogn) operation. Note however that n is not the length of Terms, but the number of terms in Terms that start with the prefix. This is what makes it faster than topKmatches in BruteAutocomplete.

2.

BruteAutoComplete:

Algorithmic Analysis:
The runtime of topKMatches is not significantly affected by the value of k. The runtime of the method is determined by the leading term, which in most cases happens to be O(n), where n is the size of the Terms array.

Empirical Analysis:
My hypothesis is backed up by the following data:

```
Time for topKMatches("mo", 1) -  0.001498336352
Time for topKMatches("mo", 4) -  0.001503375693
Time for topKMatches("mo", 7) -  0.001756798361
```

BINARYSEARCHAUTOCOMPLETE:

Algorithmic Analysis:

The runtime of topKMatches is also not significantly affected by the value of k, as the runtime of the method is determined by the number of terms that start with the prefix, m, thus O(m), which happens to be the leading term in the method most of the times.
Empirical Analysis:
My hypothesis is backed up by the following data:

```
Time for topKMatches("mosty, poland", 1) -  2.36302E-6
```

```
Time for topKMatches("mosty, poland", 4) -  2.128584E-6
Time for topKMatches("mosty, poland", 7) -  2.229259E-6
```

TrieAutoComplete:

Algorithmic Analysis:

The same can be said for topKMatches. The value of k doesn't seem to affect runtime. Runtime is affected mostly by the number of matching terms, and the length of those matching terms, because that determines how far down we go the Trie.

EMPIRICAL ANALYSIS:

```
Time for topKMatches("notarealword", 1) -  1.275257E-6
Time for topKMatches("notarealword", 4) -  1.073263E-6
Time for topKMatches("notarealword", 7) -  1.086167E-6
```

3.Theoretical:
BruteAutocomplete:

-For topKMatches, the method appears to be running in O (n) time, where n is the size of the Terms array. This is because, you have to check every item in term to see whether it starts with the given prefix. If you look at the other operations within the for loop, i.e. adding, peeking and removing items from the priority queue, they appear to be taking constant time, and so have no significant effect on the runtime taking O(n) time. The other for loop takes k time, or the size of the priority queue, o(m), time. Adding these two runtimes yields the leading term, which in most cases will be O (n) time.

-The same is true for topMatches, as we have to go through all the items in terms to guarantee that we find the max term, thus yielding a runtime of O (n).

BinarySearchAutoComplete:

-In both topMatch and topKMatches of BinarySearch, the operations *firstIndexOf*(myTerms, preTerm, **new** Term.PrefixOrder(prefix.length())); and *lasttIndexOf*(myTerms, preTerm, **new** Term.PrefixOrder(prefix.length())); are used to find the first and last indices of where items that begin with prefix are. This means that we don't have to go through the entire array Terms. In best cases, where an item occurs once or less, the methods will take O(1) constant time, unlike in

Brute where we had to traverse the entire array. In worst case, the methods will take O(m) time, where m is the number of terms that start with prefix(matching terms) in the array Terms. The operations, firstIndexOf and lastIndexOf approximately take O(log n) time, where n is the number of elements in Terms. Depending on which is the leading term between log n and m, the methods will take approximately that amount of time to run.

-**Binary** May not take shorter times than brute on all occasions because, if you pass as a parameter a prefix like "m", a lot of words start with m, so the runtime, even for Binary must be close to something like O(n) and hence may not run faster as we would expect.

EMPIRICAL:
I used cities.txt to get the following data for BruteAutocomplete:

```
Time to initialize - 0.007597176
Time for topMatch("") - 1.4059965E-4
Time for topMatch("mosty, poland") - 9.10954943E-4
Time for topMatch("m") - 1.37630243E-4
Time for topMatch("mo") - 1.38464683E-4
Time for topMatch("notarealword") - 0.001549637201
Time for topKMatches("", 1) -  0.00146329538
Time for topKMatches("", 4) -  0.001428648892
Time for topKMatches("", 7) -  0.001374917844
Time for topKMatches("mosty, poland", 1) -  0.001600032367
Time for topKMatches("mosty, poland", 4) -  0.001552837709
Time for topKMatches("mosty, poland", 7) -  0.001547105131
Time for topKMatches("m", 1) -  0.00154365289
Time for topKMatches("m", 4) -  0.001509406167
Time for topKMatches("m", 7) -  0.001484584338
Time for topKMatches("mo", 1) -  0.001510304964
Time for topKMatches("mo", 4) -  0.001468179009
Time for topKMatches("mo", 7) -  0.001529835864
Time for topKMatches("notarealword", 1) -  0.001515027309
Time for topKMatches("notarealword", 4) -  0.001527574764
Time for topKMatches("notarealword", 7) -  0.001451602771
```

For BinarySearchAutocomplete:

```
Time to initialize - 0.177311227
Time for topMatch("") - 7.26734983E-4
Time for topMatch("mosty, poland") - 1.881835E-5
```

```
Time for topMatch("m") - 6.4879911E-5
Time for topMatch("mo") - 1.0805412E-5
Time for topMatch("notarealword") - 6.327376E-6
Time for topKMatches("", 1) -  0.027883210788888889
Time for topKMatches("", 4) -  0.024141288485576923
Time for topKMatches("", 7) -  0.02452400543137255
Time for topKMatches("mosty, poland", 1) -  3.190126E-6
Time for topKMatches("mosty, poland", 4) -  2.407588E-6
Time for topKMatches("mosty, poland", 7) -  2.216575E-6
Time for topKMatches("m", 1) -  0.001345201968
Time for topKMatches("m", 4) -  0.001344251368
Time for topKMatches("m", 7) -  0.001291016739
Time for topKMatches("mo", 1) -  2.55800717E-4
Time for topKMatches("mo", 4) -  2.52895639E-4
Time for topKMatches("mo", 7) -  2.47921919E-4
Time for topKMatches("notarealword", 1) -  1.444455E-6
Time for topKMatches("notarealword", 4) -  1.288446E-6
Time for topKMatches("notarealword", 7) -  1.334692E-6.
```

My previous hypothesis holds true except for the cases that I had raised concerns about. When you have a prefix like “m”, or “mo”, a lot of words will start with those prefix, hence the length of the new array (The one with items that start with prefix) will be approaching the length of the Terms array, and so it’ll be like looping through the Terms array like we did in BruteAutocomplete. Hence, as you can see, Brute may outperform binary in search cases.

4.**BRUTEAUTOCOMPLETE:**

Increasing the size of prefix does not have a change on the runtime of topKMatches and topMatches because we would still have to go through all the terms to check if each term begins with prefix.

Increasing the size of the text file does increase runtime as runtime is directly proportional to the number of terms in the file. As I explained in number 3, the runtimes of topKMatches and topMatches is dependent on the number of terms.

Empirical Data:
For fourletterwords.txt:

```
Found 456976 words
Time to initialize - 0.021848455
```

```
Time for topMatch("") - 9.45311452E-4
Time for topMatch("nenk") - 0.00384072738
Time for topMatch("n") - 0.00266556238
Time for topMatch("ne") - 0.002796695716
Time for topMatch("notarealword") - 0.002940368376
Time for topKMatches("", 1) -  0.003408300414
Time for topKMatches("", 4) -  0.003911494288
Time for topKMatches("", 7) -  0.00470403832
Time for topKMatches("nenk", 1) -  0.004363451862
Time for topKMatches("nenk", 4) -  0.003420315715
Time for topKMatches("nenk", 7) -  0.003388471354
Time for topKMatches("n", 1) -  0.003411337412
Time for topKMatches("n", 4) -  0.003424384626
Time for topKMatches("n", 7) -  0.003374622573
Time for topKMatches("ne", 1) -  0.003310962655
Time for topKMatches("ne", 4) -  0.003313352391
Time for topKMatches("ne", 7) -  0.003335809171
Time for topKMatches("notarealword", 1) -  0.002836846568
Time for topKMatches("notarealword", 4) -  0.002937704381
Time for topKMatches("notarealword", 7) -  0.003004760102
```

For fourletterwordshalf.txt:

```
Found 228488 words
Time to initialize - 0.011863713
Time for topMatch("") - 5.25518618E-4
Time for topMatch("aenk") - 0.003489176255
Time for topMatch("a") - 4.45618377E-4
Time for topMatch("ae") - 4.80046918E-4
Time for topMatch("notarealword") - 0.002411005097
Time for topKMatches("", 1) -  0.003677762955
Time for topKMatches("", 4) -  0.005664479551528879
Time for topKMatches("", 7) -  0.003486491744
Time for topKMatches("aenk", 1) -  0.004186056012
Time for topKMatches("aenk", 4) -  0.003458777139
Time for topKMatches("aenk", 7) -  0.004415511759
Time for topKMatches("a", 1) -  0.004357138805
Time for topKMatches("a", 4) -  0.004742441335
Time for topKMatches("a", 7) -  0.0043871727
Time for topKMatches("ae", 1) -  0.003378927778
Time for topKMatches("ae", 4) -  0.003330117476
Time for topKMatches("ae", 7) -  0.004139504721
Time for topKMatches("notarealword", 1) -  0.002526175433
Time for topKMatches("notarealword", 4) -  0.002474208714
```

```
Time for topKMatches("notarealword", 7) -  0.002509882282
```

If you were to look at each data point for each data file, most of them double, when the size of the file,i.e. **fourletterwords.txt:,** has been doubled, proving my hypothesis that the relationship between the size of the textfile/number of terms and the runtime of topMatch and topKMatches is linear.

BINARYSERACHAUTOCOMPLETE:
Increasing the size of the prefix reduces the runtime, as the number of terms that begin with the prefix will reduce as you increase prefix length.E.g. there is bound to be more terms beginning with "a", than "ang". Consequently, the runtime is reduced, because, like I said previously, the run time of topKmatches and topmatches is dependent on the number of matching terms.

Increasing the size of the text file increases the runtime of topmatch and topkmatches because there is a higher probability of finding more matching terms when more terms are involved.

EMPIRICAL ANALYSIS:
FOR fourletterwords.txt.

```
Found 456976 words
Time to initialize - 0.047946352
Time for topMatch("") - 0.002896503891
Time for topMatch("nenk") - 9.0047751E-5
Time for topMatch("n") - 1.07387787E-4
Time for topMatch("ne") - 5.213605E-6
Time for topMatch("notarealword") - 2.557206E-6
Time for topKMatches("", 1) -  0.3904016579230769
Time for topKMatches("", 4) -  0.24000931585714286
Time for topKMatches("", 7) -  0.2563908138
Time for topKMatches("nenk", 1) -  2.982112E-6
Time for topKMatches("nenk", 4) -  1.648026E-6
Time for topKMatches("nenk", 7) -  1.690155E-6
Time for topKMatches("n", 1) -  0.004702946586
Time for topKMatches("n", 4) -  0.004285874355
Time for topKMatches("n", 7) -  0.003819595594
Time for topKMatches("ne", 1) -  7.0878845E-5
Time for topKMatches("ne", 4) -  7.0556954E-5
Time for topKMatches("ne", 7) -  7.3141636E-5
Time for topKMatches("notarealword", 1) -  2.026899E-6
Time for topKMatches("notarealword", 4) -  7.46793E-7
Time for topKMatches("notarealword", 7) -  1.392118E-6
```

FOR fourletterwordshalf.txt.

```
Found 228488 words
Time to initialize - 0.058206117
Time for topMatch("") - 0.002238261221
Time for topMatch("aenk") - 2.7983109E-5
Time for topMatch("a") - 5.7717941E-5
Time for topMatch("ae") - 3.67522E-6
Time for topMatch("notarealword") - 3.605301E-6
Time for topKMatches("", 1) -  0.11983433742857143
Time for topKMatches("", 4) -  0.10554147735416666
Time for topKMatches("", 7) -  0.10807051261702127
Time for topKMatches("aenk", 1) -  2.961697E-6
Time for topKMatches("aenk", 4) -  2.943751E-6
Time for topKMatches("aenk", 7) -  2.461265E-6
Time for topKMatches("a", 1) -  0.004492015073
Time for topKMatches("a", 4) -  0.004151339426
Time for topKMatches("a", 7) -  0.003973777827
Time for topKMatches("ae", 1) -  9.3274743E-5
Time for topKMatches("ae", 4) -  7.7314939E-5
Time for topKMatches("ae", 7) -  1.03379325E-4
Time for topKMatches("notarealword", 1) -  1.77134E-6
Time for topKMatches("notarealword", 4) -  1.270294E-6
Time for topKMatches("notarealword", 7) -  6.5656E-7
```

TRIEAUTOCOMPLETE:
For **TOPMATCH,** Increasing the size of the prefix reduces the runtime of the method. This is because we have to check through fewer matching terms to determine the term with the maximum weight. Increasing the size of the file increases the number of matching terms hence runtime is increased.

The same is true for **TOPKMATCHES.** The logic above also applies here.

EMPIRICAL DATA/ANALYSIS:

FOR fourletterwords.txt:

```
Found 456976 words
Time to initialize - 0.2180896
Created 475255 nodes
```

```
Time for topMatch("") - 4.035261E-6
Time for topMatch("nenk") - 1.345971E-6
Time for topMatch("n") - 4.85626E-6
Time for topMatch("ne") - 1.947016E-6
Time for topMatch("notarealword") - 6.28752E-7
Time for topKMatches("", 1) -  0.6398891331
Time for topKMatches("", 4) -  0.66176596575
Time for topKMatches("", 7) -  1.250245705
Time for topKMatches("nenk", 1) -  9.83007E-7
Time for topKMatches("nenk", 4) -  7.26013E-7
Time for topKMatches("nenk", 7) -  7.4223E-7
Time for topKMatches("n", 1) -  0.007025832845505618
Time for topKMatches("n", 4) -  0.00608705046593674
Time for topKMatches("n", 7) -  0.005804946511600928
Time for topKMatches("ne", 1) -  8.4160351E-5
Time for topKMatches("ne", 4) -  8.7438858E-5
Time for topKMatches("ne", 7) -  7.8010422E-5
Time for topKMatches("notarealword", 1) -  1.76991E-7
Time for topKMatches("notarealword", 4) -  1.52729E-7
Time for topKMatches("notarealword", 7) -  1.661E-7
```

FOR fourletterwordshalf.txt:

```
Found 228488 words
Time to initialize - 0.086639575
Created 237628 nodes
Time for topMatch("") - 4.60546E-6
Time for topMatch("aenk") - 2.16832E-6
Time for topMatch("a") - 3.602013E-6
Time for topMatch("ae") - 3.72248E-6
Time for topMatch("notarealword") - 1.24306E-7
Time for topKMatches("", 1) -  0.20918901125
Time for topKMatches("", 4) -  0.20461510828
Time for topKMatches("", 7) -  0.18993437325925927
Time for topKMatches("aenk", 1) -  2.21512E-7
Time for topKMatches("aenk", 4) -  1.44742E-7
Time for topKMatches("aenk", 7) -  1.49283E-7
Time for topKMatches("a", 1) -  0.005698433412300683
Time for topKMatches("a", 4) -  0.005993408073053892
Time for topKMatches("a", 7) -  0.006178320166666667
Time for topKMatches("ae", 1) -  1.0692285E-4
Time for topKMatches("ae", 4) -  1.07445874E-4
Time for topKMatches("ae", 7) -  1.01676523E-4
```

```
Time for topKMatches("notarealword", 1) -  7.2905E-8
Time for topKMatches("notarealword", 4) -  1.04738E-7
Time for topKMatches("notarealword", 7) -  7.4337E-8
```