P4 Analysis

Question 1

Inside of `BenchmarkForAutocomplete`, uncomment the two other implementation names so that `myCompletorNames` has all three Strings: `"BruteAutocomplete"`, `"BinarySearchAutocomplete"`, and `"HashListAutocomplete"` (if you want to benchmark only a subset of these, perhaps because one isn't working, just leave it commented).

Results for `threeletterwords.txt`

```
init time: 0.005445     for BruteAutocomplete
init time: 0.005893     for BinarySearchAutocomplete
init time: 0.1013       for HashListAutocomplete
search  size   #match  BruteAutoc      BinarySear      HashListAu
        17576   50     0.00388007      0.00478097      0.00027722
        17576   50     0.00081809      0.00263939      0.00000595
a       676     50     0.00053853      0.00028856      0.00000587
a       676     50     0.00065053      0.00023130      0.00000664
b       676     50     0.00056828      0.00023084      0.00000539
c       676     50     0.00058031      0.00019169      0.00000535
g       676     50     0.00059423      0.00018914      0.00000565
ga      26      50     0.00045677      0.00005479      0.00000612
go      26      50     0.00049987      0.00007248      0.00000848
gu      26      50     0.00061706      0.00005596      0.00000918
x       676     50     0.00225924      0.00029582      0.00000956
y       676     50     0.00057455      0.00018539      0.00000643
z       676     50     0.00055304      0.00018155      0.00000645
aa      26      50     0.00042720      0.00004033      0.00000591
az      26      50     0.00045099      0.00004439      0.00000606
za      26      50     0.00054271      0.00004533      0.00000680
zz      26      50     0.00066578      0.00003945      0.00000542
zqzqwwx 0       50     0.00048247      0.00003390      0.00000269
size in bytes=246064    for BruteAutocomplete
size in bytes=246064    for BinarySearchAutocomplete
size in bytes=354276    for HashListAutocomplete
```

Results for `fourletterwords.txt`

```
init time: 0.08132      for BruteAutocomplete
init time: 0.04479      for BinarySearchAutocomplete
```

```
init time: 1.259	for HashListAutocomplete
search	size	#match	BruteAutoc	BinarySear	HashListAu
	456976	50	0.01324871	0.02354284	0.00034045
	456976	50	0.00818294	0.00375467	0.00001185
a	17576	50	0.01154491	0.00050358	0.00000778
a	17576	50	0.01128182	0.00039324	0.00000755
b	17576	50	0.01197515	0.00032534	0.00000947
c	17576	50	0.00736875	0.00023748	0.00000718
g	17576	50	0.00779782	0.00033690	0.00000953
ga	676	50	0.00559799	0.00008266	0.00000683
go	676	50	0.00647554	0.00007439	0.00000682
gu	676	50	0.00656252	0.00009879	0.00000976
x	17576	50	0.00549682	0.00025487	0.00000851
y	17576	50	0.00591523	0.00024746	0.00000817
z	17576	50	0.00542025	0.00022842	0.00000804
aa	676	50	0.00566596	0.00007502	0.00000677
az	676	50	0.00857659	0.00009033	0.00000995
za	676	50	0.00600338	0.00007632	0.00000808
zz	676	50	0.00563589	0.00007218	0.00000766
zqzqwwx	0	50	0.00520338	0.00008377	0.00000673
size in bytes=7311616	for BruteAutocomplete
size in bytes=7311616	for BinarySearchAutocomplete
size in bytes=11075636	for HashListAutocomplete
```

Results for `alexa.txt`

```
init time: 0.3263	for BruteAutocomplete
init time: 1.482	for BinarySearchAutocomplete
init time: 6.503	for HashListAutocomplete
search	size	#match	BruteAutoc	BinarySear	HashListAu
	1000000	50	0.02679856	0.05200530	0.00038964
	1000000	50	0.01382396	0.02623496	0.00002870
a	69464	50	0.01176701	0.00162462	0.00000710
a	69464	50	0.01140002	0.00156591	0.00000685
b	56037	50	0.01171330	0.00056607	0.00000630
c	65842	50	0.01161230	0.00141662	0.00000633
g	37792	50	0.01173627	0.00131376	0.00000726
ga	6664	50	0.01165594	0.00027111	0.00000682
go	6953	50	0.01211824	0.00027189	0.00000773
gu	2782	50	0.01103423	0.00015184	0.00000672
x	6717	50	0.01139233	0.00023349	0.00000670
y	16765	50	0.01211205	0.00044161	0.00000702
z	8780	50	0.01127828	0.00027957	0.00000745
aa	718	50	0.01201244	0.00008824	0.00000699
az	889	50	0.01140576	0.00009228	0.00000617
za	1718	50	0.01080417	0.00012487	0.00000617
zz	162	50	0.01078174	0.00006415	0.00000672
```

```
zqzqwwx 0       50      0.01106502      0.00008386      0.00000322
size in bytes=38204230   for BruteAutocomplete
size in bytes=38204230   for BinarySearchAutocomplete
size in bytes=98824414   for HashListAutocomplete
```

Question 2

Let `N` be the total number of terms, let `M` be the number of terms that prefix-match a given `search` term (the `size` column above), and let `k` be the number of highest weight terms returned by `topMatches` (the `#match` column above). The runtime complexity of `BruteAutocomplete` is `O(N log(k))`. The runtime complexity of `BinarySearchAutocomplete` is `O(log(N) + M log(k))`. Yet you should notice (as seen in the example timing above) that `BruteAutocomplete` is similarly efficient or even slightly more efficient than `BinarySearchAutocomplete` on the empty `search` String `""`. Answer the following:

For the empty `search` String `""`, does `BruteAutocomplete` seem to be asymptotically more efficient than `BinarySearchAutocomplete` with respect to `N`, or is it just a constant factor more efficient? To answer, consider the different data sets you benchmarked with varying `size`.

Consider the following data set

File	Number of terms (`N`)	size (`M`)	#match (`k`)	BruteAutocomplete	BinarySearchAutocomplete
threeletterwords	17576	17576	50	0.00388007	0.00478097
threeletterwords	17576	17576	50	0.00081809	0.00263939
fourletterwords	456976	456976	50	0.01324871	0.02354284
fourletterwords	456976	456976	50	0.00818294	0.00375467
alexa.txt	1000000	1000000	50	0.02679856	0.05200530
alexa.txt	1000000	1000000	50	0.01382396	0.02623496

With the exception of the second run of `fourletterwords.txt`, `BruteAutocomplete` takes slightly less time to run than `BinarySearchAutocomplete.`

Explain why this observation (that `BruteAutocomplete` is similarly efficient or even slightly more efficient than `BinarySearchAutocomplete` on the empty `search` String `""`) makes sense given the values of `N` and `M`.

As seen in the table of data, `k` is constant and therefore need not be considered in this analysis. Since `search` is an empty string `""`, all terms in all three files will match that prefix. This means that the number of prefix-match the search term is equal to the total number of terms. In other words, $M = N$.

The runtime complexity of `BruteAutocomplete` is $O(N \log k)$.

The runtime complexity of `BinarySearchAutocomplete` is

$O(\log N + M \log k) = O(\log N + N \log k)$

Since `k` is constant, it can be omitted for purposes of this analysis. Therefore, the runtime of complexity of `BruteAutocomplete` can be approximated by $O(N)$ and the runtime complexity of `BinarySearchAutocomplete` can be approximated by $O(\log N + N)$, which can be approximated to just $O(N)$. This explains why `BruteAutocomplete` is similarly efficient or even slightly more efficient in some cases than `BinarySearchAutocomplete`.

With respect to `N` and `M`, when would you expect `BinarySearchAutocomplete` to become more efficient than `BruteAutocomplete`? Does the data validate your expectation? Refer specifically to your data in answering.

`BinarySearchAutocomplete` will become more efficient when $N > M$. In other words, when the `search` term is not an empty string `""`. Consider the runtime complexities of both implementations in the case that $N >> M$ and k is constant:

- `BinarySearchAutocomplete`: $O(\log N + M \log k) \approx O(\log N)$
  - Logarithmic runtime on N
- `BruteAutocomplete`: $O(N \log k)$
  - Linear runtime on N

Consider the following data set:

file	Number of terms (N)	search	size (M)	`BruteAutocomplete`	`BinarySearchAutocomplete`
threeletterwords.txt	17576	a	676	0.00053853	0.00028856
threeletterwords.txt	17576	zz	26	0.00066578	0.00003945
fourletterwords.txt	456976	a	17576	0.01154491	0.00050358
fourletterwords.txt	456976	zz	676	0.00563589	0.00007218
alexa.txt	1000000	a	69464	0.01176701	0.00162462
alexa.txt	1000000	zz	162	0.01078174	0.00006415

As shown in the data set, `BinarySearchAutocomplete` is considerably faster than `BruteAutocomplete` (up to three orders of magnitude in some cases) when $N > M$. For instance, consider the first two rows. As $N - M$ increased, the difference between both implementations also increased as `BinarySearchAutocomplete` took about 5% of the time it took `BinarySearchAutocomplete.`

Question 3

Run the `BenchmarkForAutocomplete` again using `alexa.txt` but doubling `matchSize` to `100` (`matchSize` is specified in the `runAM` method). Again copy and paste your results. Recall that `matchSize` determines `k`, the number of highest weight terms returned by `topMatches` (the `#match` column above). Do your data support the hypothesis that the dependence of the runtime on `k` is logarithmic for `BruteAutocomplete` and `BinarySearchAutocomplete`?

```
init time: 0.3329      for BruteAutocomplete
init time: 1.389       for BinarySearchAutocomplete
init time: 5.381       for HashListAutocomplete
search  size    #match  BruteAutoc      BinarySear      HashListAu
        1000000 100     0.02601607      0.03709101      0.00026319
        1000000 100     0.01509248      0.01010664      0.00000785
a       69464   100     0.01525319      0.00133598      0.00000633
a       69464   100     0.01355942      0.00117596      0.00000530
b       56037   100     0.01356559      0.00096327      0.00000526
c       65842   100     0.01517558      0.00113325      0.00000602
g       37792   100     0.01313459      0.00088867      0.00000765
ga      6664    100     0.01288360      0.00032347      0.00000590
go      6953    100     0.01268222      0.00030222      0.00000666
gu      2782    100     0.01297314      0.00020029      0.00000703
x       6717    100     0.01328651      0.00031608      0.00000689
y       16765   100     0.01409787      0.00041241      0.00000576
z       8780    100     0.01364308      0.00033406      0.00000689
aa      718     100     0.01434753      0.00013118      0.00000635
az      889     100     0.01439476      0.00014925      0.00000767
za      1718    100     0.01368477      0.00017323      0.00000669
zz      162     100     0.01280491      0.00007099      0.00000646
zqzqwwx 0       100     0.01244856      0.00007950      0.00000289
size in bytes=38204230  for BruteAutocomplete
size in bytes=38204230  for BinarySearchAutocomplete
size in bytes=98824414  for HashListAutocomplete
```

Consider the following dataset containing the times for `BruteAutocomplete` and `BinarySearchAutocomplete` to run for different, randomly-picked `search` values (`b`, `go`, `az`):

Number of terms (N)	size (M)	#matches (k)	`BruteAutocomplete`	`BinarySearchAutocomplete`
1000000	56037	50	0.01171330	0.00056607
1000000	56037	100	0.01356559	0.00096327
1000000	6953	50	0.01211824	0.00027189

1000000	6953	100	0.01268222	0.00030222
1000000	889	50	0.01140576	0.00009228
1000000	889	100	0.01439476	0.00014925

Comparing the values of produced by `BruteAutocomplete` and `BinarySearchAutocomplete` when $k = 50$ and when $k = 100$ (when it is doubled), it is easy to see that the time values do not double or "nearly" double. In fact, they increase by a pretty modest amount. Most times it increases by a factor of 10% to 20%. This indicates that the runtime complexities of `BruteAutocomplete` and `BinarySearchAutocomplete` do not have a linear dependence on k (k) or nearly-linear $(k \log k)$, but in fact logarithmic: $\log k$. Therefore, the data do support the hypothesis that the dependence on k is logarithmic for both implementations.

Question 4

Briefly explain why `HashListAutocomplete` is much more efficient in terms of the empirical runtime of `topMatches`, but uses more memory than the other `Autocomplete` implementations.

`HashListAutocomplete` utilizes a `HashMap`, which in turn uses a hash table to store values. Getting values from a hash table has a constant runtime $O(1)$. However, every key used in the hash table occupies space in memory. Since `HashListAutocomplete` uses every possible prefix as a key in the `HashMap`, it not only stores each term (a string and double) it retrieved from the file, but it also stores all the possible keys (strings). On the other hand, the other two implementations use binary searching and brute searching to get values. While these algorithms do not run on constant time, they only require storing the terms. Thus, they take up less memory than `HashListAutocomplete`, but they are not as efficient. Each implementation offers a trade off between memory and efficiency.