Lily Parker
P4 Autocomplete Analysis

```
init time: 0.009282     for BruteAutocomplete
init time: 0.01294      for BinarySearchAutocomplete
init time: 0.1014       for HashListAutocomplete
search  size    #match  BruteAutoc      BinarySear      HashListAu
        17576   50      0.00500517      0.01356221      0.00006496
        17576   50      0.00251358      0.00280346      0.00012346
a       676     50      0.00100704      0.00141254      0.00011908
a       676     50      0.00109050      0.00084788      0.00012767
b       676     50      0.00084888      0.00064929      0.00012246
c       676     50      0.00117579      0.00080021      0.00012246
g       676     50      0.00095046      0.00073792      0.00013592
ga      26      50      0.00089500      0.00061838      0.00013288
go      26      50      0.00089338      0.00060825      0.00012942
gu      26      50      0.00082717      0.00077529      0.00012279
x       676     50      0.00095458      0.00083167      0.00016292
y       676     50      0.00224975      0.00080008      0.00012496
z       676     50      0.00108971      0.00084071      0.00012983
aa      26      50      0.00078771      0.00082213      0.00013767
az      26      50      0.00081163      0.00108929      0.00012729
za      26      50      0.00074533      0.00066700      0.00012263
zz      26      50      0.00076871      0.00070104      0.00013150
zqzqwwx 0       50      0.00144913      0.00018513      0.00050975
size in bytes=246064     for BruteAutocomplete
size in bytes=246064     for BinarySearchAutocomplete
size in bytes=740948     for HashListAutocomplete
```

Threeletterwords.txt

```
init time: 0.06380       for BruteAutocomplete
init time: 0.06596       for BinarySearchAutocomplete
init time: 0.5556        for HashListAutocomplete
search  size    #match  BruteAutoc      BinarySear      HashListAu
        456976  50      0.01770504      0.02324800      0.00005625
        456976  50      0.01140038      0.00793921      0.00014567
a       17576   50      0.01370842      0.00303413      0.00016004
a       17576   50      0.00904525      0.00064017      0.00013829
b       17576   50      0.01320463      0.00041929      0.00013892
c       17576   50      0.01125271      0.00048100      0.00014408
g       17576   50      0.00608592      0.00045058      0.00012996
ga      676     50      0.00695967      0.00047133      0.00013983
go      676     50      0.00678217      0.00036058      0.00014079
gu      676     50      0.00638271      0.00035758      0.00013975
x       17576   50      0.00662617      0.00051429      0.00013225
y       17576   50      0.00665638      0.00078013      0.00015954
z       17576   50      0.00640279      0.00048667      0.00018058
aa      676     50      0.00626563      0.00033521      0.00014813
az      676     50      0.00696921      0.00047321      0.00015171
za      676     50      0.00645829      0.00030692      0.00017438
zz      676     50      0.00654292      0.00029438      0.00014683
zqzqwwx 0       50      0.00975554      0.00045129      0.00082525
size in bytes=7311616     for BruteAutocomplete
size in bytes=7311616     for BinarySearchAutocomplete
size in bytes=29354676    for HashListAutocomplete
```

Fourletterwords.txt

```
init time: 0.1921       for BruteAutocomplete
init time: 1.884        for BinarySearchAutocomplete
init time: 2.978        for HashListAutocomplete
search  size    #match  BruteAutoc      BinarySear      HashListAu
        1000000 50      0.01668342      0.03977404      0.00006458
        1000000 50      0.01090629      0.02483483      0.00015025
a       69464   50      0.01146508      0.00449908      0.00014742
a       69464   50      0.00923146      0.00369958      0.00013346
b       56037   50      0.00912463      0.00128271      0.00013575
c       65842   50      0.00915063      0.00121517      0.00013463
g       37792   50      0.00890479      0.00098167      0.00013621
ga      6664    50      0.00887963      0.00081046      0.00013421
go      6953    50      0.00886533      0.00062154      0.00014025
gu      2782    50      0.00928879      0.00056654      0.00013925
x       6717    50      0.00852158      0.00067679      0.00013329
y       16765   50      0.00860329      0.00115979      0.00013967
z       8780    50      0.00846471      0.00101979      0.00015496
aa      718     50      0.00935608      0.00072113      0.00015579
az      889     50      0.00933421      0.00054754      0.00025392
za      1718    50      0.00867308      0.00059808      0.00015046
zz      162     50      0.00860517      0.00055958      0.00015500
zqzqwwx 0       50      0.00940263      0.00059938      0.00016504
size in bytes=38204230   for BruteAutocomplete
size in bytes=38204230   for BinarySearchAutocomplete
size in bytes=420347294  for HashListAutocomplete
```

Alexa.txt

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.

For the empty search String “”, BruteAutocomplete is not actually asymptotically more efficient than BinarySearchAutocomplete with respect to N. It is actually a constant factor that is more efficient because asymptotically O(N log(k)) for BruteAutocomplete is the same as O(log(N) + Mlog(k) for BinarySearchAutocomplete **-** as N increases the comparison between these two runtimes does not consistently increase.

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.

N and M will be equal in the example we are looking at for the empty search String “” (given N is the total number of terms and M is the number of terms that prefix-match a given search term). If we replace the term M with N (because of equality) in the runtime of BinarySearchAutocomplete, we see that the runtime of BinarySearchAutocomplete is O(log(N) + Nlog(k)) and the runtime for BruteAutocomplete is still just O(Nlog(k)). We see here that the runtime of BinarySearchAutocomplete has an extra N term, which makes it overall less efficient for this first example.

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.

As the search string of length N grows, and is no longer an empty string, M will decrease. As this is the case, we can expect that BinarySearchAutocomplete will be more efficient. As M decreases and N increases, BinarySearchAutocomplete multiplies its runtime by a factor of N - (O(N log(k)). The data provided above shows this. Using the search string ‘a’ in “alexa.txt” we see that the BinarySearchAutocomplete compiles in 0.0045ms vs BruteAutocomplete’s 0.0115ms.

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: 0.1960       for BruteAutocomplete
init time: 1.914        for BinarySearchAutocomplete
init time: 2.747        for HashListAutocomplete
search  size    #match  BruteAutoc      BinarySear      HashListAu
        1000000 100     0.01750088      0.03736917      0.00008275
        1000000 100     0.01213321      0.01924521      0.00015363
a       69464   100     0.01158017      0.00518917      0.00014692
a       69464   100     0.01026217      0.00356883      0.00012404
b       56037   100     0.00992971      0.00182396      0.00013046
c       65842   100     0.00996483      0.00173463      0.00013233
g       37792   100     0.00964696      0.00120533      0.00012325
ga      6664    100     0.00969575      0.00089021      0.00012550
go      6953    100     0.00965129      0.00085554      0.00014642
gu      2782    100     0.00990217      0.00069475      0.00012350
x       6717    100     0.00971408      0.00089133      0.00013529
y       16765   100     0.00940975      0.00128050      0.00012738
z       8780    100     0.00932121      0.00078521      0.00014642
aa      718     100     0.00995129      0.00060938      0.00014425
az      889     100     0.01052458      0.00063175      0.00014913
za      1718    100     0.01011925      0.00064850      0.00014800
zz      162     100     0.00945646      0.00052071      0.00015729
zqzqwwx 0       100     0.01015571      0.00047804      0.00015513
size in bytes=38204230   for BruteAutocomplete
size in bytes=38204230   for BinarySearchAutocomplete
size in bytes=420347294  for HashListAutocomplete
```

Alexa.txt // matchSize = 100

This data shows that the dependence of the runtime on k is logarithmic for BruteAutocomplete and BinarySearchAutocomplete. This cannot be confirmed by BruteAutocomplete because it is being multiplied by N - which is constant in this data. However, it is confirmed when we look at the runtimes of BinarySearchAutocomplete, which has a runtime of O(log(N) + Mlog(k) - in this example, since M varies throughout the data the term Mlog(k) will show a notable pattern. The runtime for BinarySearchAutocomplete is greater for alexa.txt when k=100. Essentially, changing the value of k (for example squaring it) when N and M are This means the data does not have a linear or quadratic pattern due to k, but rather it is a logarithmic pattern.

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 is much more efficient in terms of the runtime of its TopMatches method. In HashListAutocomplete TopMatches has a constant runtime, O(1), because it only checks if prefix is a key in a HashMap - this searching in a hashmap is constant time, making this the most effective implementation of the topMatches method.

However, it is important to note the memory runtime tradeoff. Though HashListAutocomplete is far more efficient, it requires more memory than BinarySearchAutocomplete and BruteAutocomplete. In HashListAutocomplete it creates a large HashMap which stores all prefixes, and also characters and doubles within the value. So, though it only utilizes lookup/get for a quick runtime, it also requires more memory usage due to the HashMap.