1. Run the program **BenchmarkForAutocomplete** and copy/paste the results into the analysis.txt file in the appropriate location. You'll need to run three times, once for each of the files in the Benchmark program: threeletterwords.txt, fourletterwords.txt, and alexa.txt.

a) Threeletterwords.txt

init time: 0.04024 for BruteAutocomplete
init time: 0.02612 for BinarySearchAutocomplete
init time: 0.1693 for HashListAutocomplete

search	size	match	Brute	BinarySearch	HashList
	17576	50	0.00826550	0.05005650	0.00038820
	17576	50	0.00085280	0.00491510	0.00000720
a	676	50	0.00062300	0.00026110	0.00000730
a	676	50	0.00047860	0.00023060	0.00000630
b	676	50	0.00048400	0.00039600	0.00000680
c	676	50	0.00073020	0.00025180	0.00000600
g	676	50	0.00045870	0.00069470	0.00000790
ga	26	50	0.00043760	0.00009820	0.00000730
go	26	50	0.00115110	0.00233650	0.00001280
gu	26	50	0.00075050	0.00014820	0.00001800
x	676	50	0.00031320	0.00034230	0.00000690
y	676	50	0.00028110	0.00034970	0.00000650
z	676	50	0.00054560	0.00035240	0.00000960
aa	26	50	0.00022000	0.00006160	0.00000790
az	26	50	0.00042790	0.00010930	0.00001150
za	26	50	0.00054340	0.00012410	0.00001680
zz	26	50	0.00023270	0.00007070	0.00000690
zqzqwwx	0	50	0.00038360	0.00002860	0.00000270

size in bytes=246064 for BruteAutocomplete
size in bytes=246064 for BinarySearchAutocomplete
size in bytes=1092468 for HashListAutocomplete

b) Fourletterwords.txt

init time: 0.1126 for BruteAutocomplete
init time: 0.04593 for BinarySearchAutocomplete
init time: 1.722 for HashListAutocomplete

search	size	match	Brute	BinarySearch	HashList
	456976	50	0.01631120	0.04504930	0.00114570
	456976	50	0.00659960	0.00725590	0.00001170
a	17576	50	0.00705080	0.00038070	0.00001770
a	17576	50	0.00641180	0.00034900	0.00000720
b	17576	50	0.00459140	0.00031560	0.00000700
c	17576	50	0.00458370	0.00035640	0.00000790
g	17576	50	0.00469790	0.00033500	0.00000760
ga	676	50	0.00467230	0.00009610	0.00000660
go	676	50	0.00458600	0.00009650	0.00000630
gu	676	50	0.00576670	0.00017620	0.00000920
x	17576	50	0.00446400	0.00030680	0.00000690
y	17576	50	0.00448760	0.00034810	0.00000720
z	17576	50	0.00549950	0.00034000	0.00000750
aa	676	50	0.00476290	0.00007180	0.00000670
az	676	50	0.00616120	0.00010330	0.00000720
za	676	50	0.00513010	0.00007220	0.00000620
zz	676	50	0.00474940	0.00006930	0.00000690

zqzqwwx	0	50	0.00882890	0.00004900	0.00000300

size in bytes=7311616 for BruteAutocomplete
size in bytes=7311616 for BinarySearchAutocomplete
size in bytes=40322100 for HashListAutocomplete

c) Alexa.txt

init time: 0.5721 for BruteAutocomplete
init time: 2.547 for BinarySearchAutocomplete
init time: 12.65 for HashListAutocomplete

search	size	match	Brute	BinarySearch	HashList
	1000000	50	0.05561250	0.16265260	0.00052780
	1000000	50	0.05585380	0.14846720	0.00001220
a	69464	50	0.01544220	0.00509310	0.00002380
a	69464	50	0.02096590	0.00570080	0.00001430
b	56037	50	0.02621330	0.00357010	0.00001020
c	65842	50	0.03188290	0.00772730	0.00001040
g	37792	50	0.01628360	0.00353550	0.00003450
ga	6664	50	0.02321350	0.00055620	0.00000690
go	6953	50	0.01609100	0.00056420	0.00000710
gu	2782	50	0.02089610	0.00032370	0.00000650
x	6717	50	0.01521790	0.00080570	0.00000780
y	16765	50	0.01645220	0.00149420	0.00001020
z	8780	50	0.02516870	0.00104960	0.00001060
aa	718	50	0.03285150	0.00068170	0.00001210
az	889	50	0.03065340	0.00029340	0.00000910
za	1718	50	0.02771410	0.00038690	0.00000970
zz	162	50	0.02603460	0.00011930	0.00000770
zqzqwwx	0	50	0.03528190	0.00009590	0.00000360

size in bytes=38204230 for BruteAutocomplete
size in bytes=38204230 for BinarySearchAutocomplete
size in bytes=475893648 for HashListAutocomplete

2. Run the program again for alexa.txt with #matches = 10000, paste the results, and then explain to what extent the # matches affects the runtime. The # matches, **matchSize,** is specified in the method **runAM** (for run all matches)

a) Alexa.txt

init time: 0.9097 for BruteAutocomplete
init time: 3.544 for BinarySearchAutocomplete
init time: 20.48 for HashListAutocomplete

search	size	#match	Brute	BinarySearch	HashList
	1000000	10000	0.06263280	0.27318200	0.00054480
	1000000	10000	0.04347120	0.27002000	0.00001210
a	69464	10000	0.04164080	0.04619630	0.00005810
a	69464	10000	0.09061930	0.14369660	0.00001460
b	56037	10000	0.03495590	0.05115260	0.00001440
c	65842	10000	0.06079170	0.06949300	0.00001290
g	37792	10000	0.03660920	0.04751170	0.00001800
ga	6664	10000	0.05017710	0.00942010	0.00001850
go	6953	10000	0.03766340	0.01291530	0.00001380
gu	2782	10000	0.03021000	0.00434240	0.00003030
x	6717	10000	0.06235680	0.01072330	0.00001270
y	16765	10000	0.03862390	0.02073510	0.00001470

z	8780	10000	0.04393730	0.01745340	0.00001450
aa	718	10000	0.04122010	0.00087080	0.00001200
az	889	10000	0.02760090	0.00089570	0.00000860
za	1718	10000	0.02971120	0.00279360	0.00001390
zz	162	10000	0.02961450	0.00019880	0.00000890
zqzqwwx	0	10000	0.04298320	0.00008430	0.00000360

size in bytes=38204230 for BruteAutocomplete
size in bytes=38204230 for BinarySearchAutocomplete
size in bytes=475893648 for HashListAutocomplete

b) From Question 1 Part C and Question 2 Part A we observe that the size of the matches does not affect the runtime of HashListAutocomplete. This is because HashListAutcomplete.topMatches() contains an O(1) call to HashMap.get(key). For BinarySearchAutocomplete we know that the total complexity is O(logN) to call firstIndex and lastIndex to binary search the terms, O(M*logk) to keep k elements in a PriorityQueue, and O(k) to return a list of k matches. Therefore, since we increase k from 50 to 10000, we observe a large increase in runtime. For BruteAutoComplete we know that the total complexity is O(N) to find all the prefix matches in the list of terms, O(M*logk) to keep k elements in the PriorityQueue and O(k) to return a list of k matches. Again, when we change k from 50 to 10000, we observe an increase in runtime although it is smaller relative to BinarySearchAutocomplete because BruteAutocomplete is less efficient overall and contains an O(N) call to find prefix matches rather than an O(logN) call.

3. Explain why the last for loop in **BruteAutocomplete.topMatches** uses a **LinkedList** (and not an **ArrayList**) **AND** why the **PriorityQueue** uses **Comparator.comparing(Term::getWeight)** to get the top k heaviest matches.
    a) A LinkedList is used because it is more efficient than an ArrayList. For an ArrayList, the add function has a runtime of O(n) in case it needs to be resized. However, for a LinkedList, the addFirst operation always has a runtime of O(1). The PriorityQueue uses a comparator so that the terms are sorted by weight from smallest to largest. Then, when PriorityQueue.remove() is called, the smallest Term is added first to the LinkedList. Then, LinkedList.addFirst() will add larger and larger Terms to the beginning of the LinkedList. In the end, you get a LinkedList of the heaviest matches in the beginning and the lightest matches toward the end.
4. Explain why **HashListAutocomplete** uses more memory than the other **Autocomplete** implementations. Be brief.
    a) HashListAutocomplete uses more memory because every single substring of every Term up to size 10 is stored inside the HashMap of the Autocomplete. Furthermore, since every substring up to size 10 is stored. It can contain multiple copies of a Term in the HashMap values.