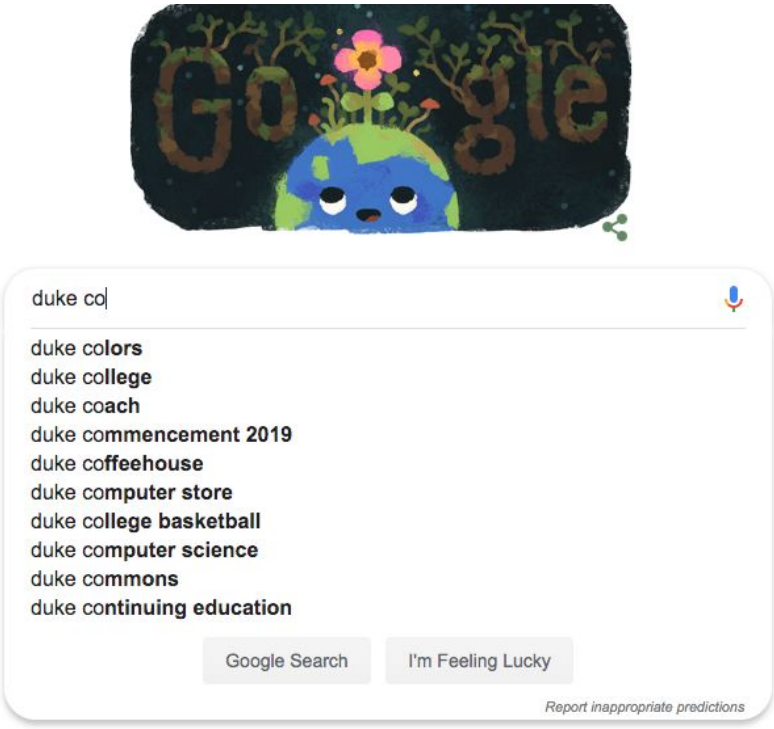
Project Introduction

TL;DR

See this document.

Background

Autocomplete is an algorithm used in many modern software applications. In all of these applications, the user types text and the application suggests possible completions for that text as shown below -- this taken at 23:00 hours on 3/20/19.



Although finding terms that contain a query by searching through all possible results is possible, these applications need some way to select only the most useful terms to display (since users will likely not comb through thousands of terms, nor will obscure terms like "duke cookiemonster" be useful to most users). Thus, autocomplete algorithms not only need a way to find terms that start with or contain the prefix, but a way of determining how likely each one is to be useful to the user and displaying "good" terms first.

According to one study, in order to be useful the algorithm must do all this in at most 50 milliseconds. If it takes any longer, the user will already be inputting the next keystroke (while humans do not on average input one keystroke every 50 milliseconds, additional time is

required for server communication, input delay, and other processes). Furthermore, the server must be able to run this computation for every keystroke, for every user. In this assignment, you will be implementing autocomplete using three different algorithms and data structures. Your autocomplete will be different than the industrial examples described above in two ways:

1. Each term will have a predetermined, constant weight/likelihood, whereas actual autocomplete algorithms might change a term's likelihood based on previous searches.
2. We will only consider terms which start with the user query, whereas actual autocomplete algorithms (such as the web browser example above) might consider terms which contain but do not start with the query.

The article linked here describes one group's recent analysis of different data structures to implement autocomplete efficiently. You'll be implementing a version of what they call a prefix hash tree, though we'll use a prefix hash list which is more efficient when terms aren't updated dynamically.

- https://medium.com/@prefixyteam/how-we-built-prefixy-a-scalable-prefix-search-service-for-powering-autocomplete-c20f98e2eff1

Acknowledgements

The assignment was developed by Kevin Wayne and Matthew Drabick at Princeton University for their Computer Science 226 class. Former head CompSci 201 UTAs, Arun Ganesh (Trinity' 17) and Austin Lu (Trinity '15) adapted the assignment for Duke with help from Jeff Forbes. Josh Hug updated the assignment and provided more of the testing framework. The current version is the result of simplification done in Fall 2018 and then modified again in Spring 2019 based on the article above and experience from previous semesters.

Git

Fork, clone, and import the cloned project from the file system. Use this URL from the course GitLab site: https://coursework.cs.duke.edu/201spring19/autocomplete ***. Be sure to fork first*** (see screen shot). Then Clone using the SSH URL after using a terminal window to cd into your Eclipse workspace. Recall that ***you should import using*** the File>Import>General>Existing Projects into Workspace -- then navigate to where you cloned the diyad project.

DO NOT DO NOT import using the Git open -- use General>Existing Projects Into Workspace.

Pushing to Git

When you make a series of changes you want to 'save', you'll push those changes to your GitHub repository. You should do this after major changes, certainly every hour or so of coding. You'll need to use the standard Git sequence to commit and push to GitHub:

```
git add .
git commit -m 'a short description of your commit here'
git push
```

Overview of Programming

You're given a main program, **`AutocompleteMain`** that launches a GUI supporting queries. This will run when you clone/import/create your project. You're given an implementation of the **`Autocompletor`** interface, **`BruteAutocomplete`**, that ***examines every term to respond to a search query***. You'll need to:

1. Implement the **`Term`** class used in all **`Autocompletor`** implementations.
2. Implement **`BinarySearchAutocomplete`** that extends **`Autocompletor`**. This requires implementing two methods in **`BinarySearchLibrary`**.
3. Implement **`HashListAutocomplete`** that extends **`Autocompletor`**.

You're given partially completed .java files for some of these classes.

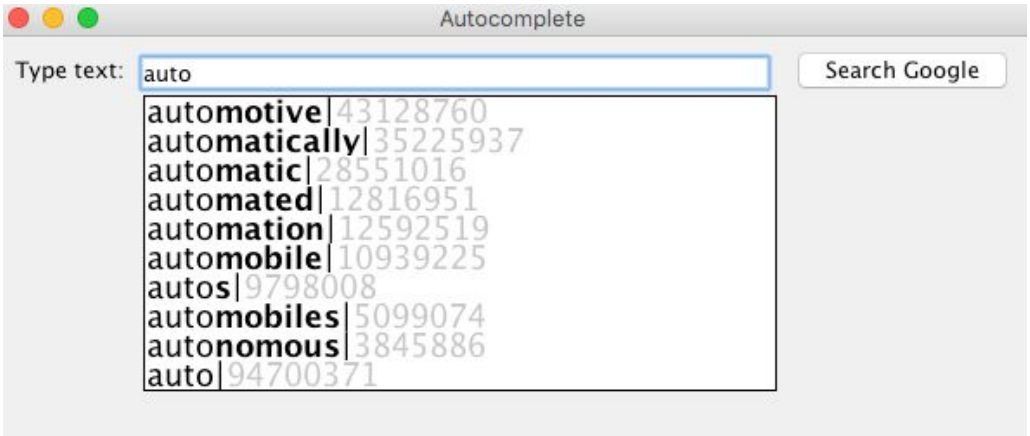
After implementing and testing these classes you'll complete benchmarks and analyses as explained below.

The Main Program

When you fork and clone the project you'll be able to run the main/driver program **`AutocompleteMain`**. It will show phrases that match a specified prefix, ***but these matches will not be in order by weight (with the heaviest weight as the first displayed match until you complete the implementation of the class `Term` described below***.

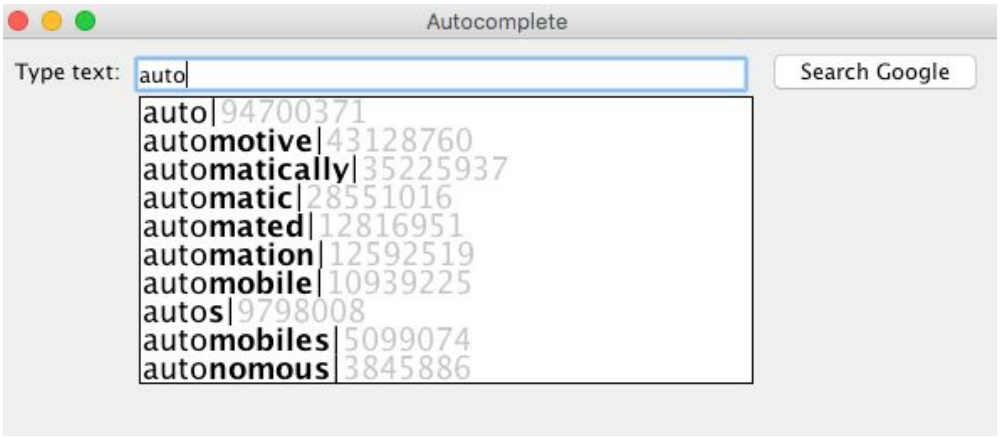
If you run and select the file **`words-333333.txt`** from the data folder you should see the output below shown in the GUI window. Note that the matching term ***auto***, with weight 94700371, is shown as the last entry in this view -- although it is the best/highest weight match of all terms shown.

Incorrect Output When **`AutocompleteMain`** first run



After you've implemented **`Term`** correctly, you should see this output for the same data file and search query as shown here with entries sorted by weight, with the *weightiest* term shown first.

Correct Output After Term class implemented



Implementing Term

You'll need to add code in the **`Term`** constructor and in the bodies of three **`Comparator`** classes that **`Term`** makes available as explained below.

The **`Term`** class serves two purposes.

1. It encapsulates a (word,weight) pair, and allows comparisons by word as the default.

2. **`Term`** exports three **`Comparator`** objects to sort **`Term`** objects in different ways, which will make each **`Autocompletor`** implementation possible. You'll need to code and test these nested/inner classes in **`Term`** to make **`Autocompletor`** implementations work.

The **`Term`** methods **`compareTo`**, **`getWord`**, **`getWeight`**, and **`toString`** are already completed. The **`Term.compareTo`** sorts by lexicographic order using the word field of **`Term`**. This functionality will be utilized in **`BinarySearchAutocomplete`**.

The Constructor

You'll ***start with a working constructor*** that stores values in appropriate instance variables. However, you ***must throw two appropriate exceptions*** as noted in the constructor documentation, e.g.,

```
throw new IllegalArgumentException("negative weight "+weight);
```

Read the documentation to see what exceptions to throw in what circumstances.

`WeightOrder` and `ReverseWeightOrder`

You must implement the **`compare`** method for each of these static inner classes. Recall that **`compare(a, b)`** should return a negative value when **`a`** comes before **`b`** in the desired order, zero when **`a`** and **`b`** are equal in terms of the desired order, and a positive value if **`a`** comes after **`b`**. Return negative, zero, and positive values as appropriate depending on a **`Term`** object's weight. For example, Term ("a", 25.0) will be less than ("a", 35.0) using **`WeightOrder.compare()`**, but larger using **`ReverseWeightOrder.compare()`**. Use the **`Term.getWeight()`** method to help determine an integer value to return.

`PrefixOrder`

`PrefixOrder` is initialized with an integer argument **`r`**, the size of the prefix for comparison purposes. The value is stored in instance variable **`myPrefixSize`** as you'll see in the code.

You must use only the first **`myPrefixSize`** characters of the words stored in **`Term`** objects **`v`** and **`w`** that are passed to **`PrefixOrder.compare`**. However, if the length of either word is less than **`myPrefixSize`**, this comparator only compares up until the end of the shorter word. This means that although "beeswax" is greater than "beekeeper" when compared lexicographically, i.e., with the natural order for strings, the two words are considered equal using a **`PrefixOrder`** with **`r == 3`** since only the first three characters are compared.

For a **`PrefixOrder`** with **`r == 4`** "beeswax" is greater than "beekeeper" since "bees" is greater than "beek". But "bee" is less than "beekeeper" and "beeswax" when **`r == 4`** since only the first three characters are compared since "bee" has only three characters.

Your code should examine only as many characters as needed to return a value. You should examine this minimal number of characters using a loop and calling **`.charAt`** to examine characters--- you'll need to write your loop and comparisons carefully to ensure that the prefix size is used correctly. See the table below for some examples. Recall that you can subtract characters, so 'a' - 'b' is a negative value and 'z' - 'a' is a positive value.

Here is a reference table for the **`PrefixOrder`** comparator.

r/prefix	v		w	Note
4	bee	<	beekeeper	"bee" < "beek"
4	bees	>	beek	‘s’ > ‘k’
4	bug	>	beeswax	‘u’ > ‘e’
4	bees	=	beeswax	"bees" == "bees"
3	beekeeper	=	beeswax	"bee" == "bee"
3	bee	=	beeswax	"bee" == "bee"

Term Testing and Creating Comparator Objects

You're given a JUnit testing class **`TestTerm`** you can use to test the **`Term`** methods you've written.

You should also run **`AutocompleteMain`** using **`BruteAutocomplete`** ***to verify that the output shown for the query 'auto'*** matches what's shown at the beginning of this document for the query "auto". The terms in the GUI should be in order by weight with "auto" first and "autonomous" last.

Note that because the nested **`Comparator`** classes are static, inner classes you'll create comparator objects using the syntax below:

```
new Term.WeightOrder()

new Term.ReverseWeightOrder()

new Term.PrefixOrder(r)
```

Implementing BinarySearchLibrary

This class is used in the implementation of the **`BinarySearchAutocomplete`** class. Once you've implemented methods **`firstIndex`** and **`lastIndex`** in **`BinarySearchLibrary`** you'll only need to implement the **`BinarySearchAutocomplete.topMatches`** method.

You're given code in **`BinarySearchLibrary.firstIndexSlow`** that is correct, ***but does not meet performance requirements***. This slow implementation will be very slow in some situations, e.g., when a list has all equal values, or many equal values, as could be the case when many words share a prefix and the comparator is a **`Term.PrefixOrder`** comparator. The code in the slow method is **`O(N)`** where there are N equal values since the code could examine all the values. To meet the performance criteria your code must not be **`O(log N)`**, but should make at most $1 + \lceil log_2 N \rceil$ comparisons -- that's one more than the ceiling of **$\log_2(N)$.** ***To do this we strongly suggest using the the loop invariant explained below*** .

Implementing firstIndex

In many implementations of binary search variables low and high are used to delimit the range of possible values for the search target. You can see this example in the code at the end of this section that comes from **`Collections.binarySearch`**. ***That code does not return the first index matching a target.***

Invariant for firstIndex

Instead of using the initialization of low and high shown in the code below, ***you should initialize these values to establish the following loop invariant*** -- that will be true every time the loop test is evaluated.

> **(low, high] is an interval containing target, if target is in the list**.

Notice that this is an open interval on the left, in particular this means that **list.get(low) cannot be equal to target**. To establish this invariant before the loop executes the first time you should consider that ***the open interval `(-1,list.length()-1]` must contain target if it is present since this interval represents every possible index in list***. Since the interval is open on the left, you cannot initialize low to zero since **`(0,99]`** includes 99, but not zero! So for a 100-element array you'd set **`low=-1`** and **`high=99`**, so the interval **`(-1,99]`** ensures that if target is present, it's either **`list.get(0)`** or **`list.get(1)`** or … **`list.get(99)`.**

After calculating the midpoint (see reference code below), you'll need to ***re-establish the invariant by comparing the target to the middle value***. In particular, in the while loop, if you determine (conceptually)

```
list[mid] < target
```

Then you can set **`low`** to **`mid`** since you know that before the loop **`(low,high]`** was the interval and if **`list[mid] < target`** then **`(mid,high]`** still maintains the invariant.

Otherwise, meaning **`list[mid] >= target`**, then you can set **`high`** to **`mid`** since if before the loop **`(low,high]`** was the interval and **`list[mid] >= target`**, then **`(low,mid]`** still maintains the invariant.

Loop Termination

If **`low`** and **`high`** differ by 1 in the interval **`(low,high]`**, and the list is sorted, then **`list.get(high)`** is **`target`** and **`high`** is the lowest/first index (if the interval isn't empty) since the interval contains a single value. This means you should use a loop guard/test that loops

```
while low+1 != high
```

since you know that **`low <= high`** is always true, then when the loop exits you'll know that **`low == high-1`** and the interval **`(low,high]`** is the same as **`(high-1,high]`** -- which contains a single value -- the index whose value is **`high`**.

You can determine whether to return -1 (target not present) based on the value of **`high`** and the value of **`list.get(high)`**. For example, if **`high`** is not a valid index, the interval is empty. This means after the loop you'll need to make one more comparison[1] of **`a.get(high)`** with **`target`** to see if they are equal. ***This loop will be correct and will meet the performance bounds if you develop it using the invariant.***

Code for lastIndex

You should develop a similar invariant and loop for the method **`lastIndex`** that you'll implement. In this case the interval you'll consider is **`[low,high)`**, i.e., open on the right. Establish the invariant before the loop, and reason about how to assign to low or high depending on how the middle value compares to key. You'll initialize low = 0 before the loop since the interval is open on the left.

Example and Incorrect Binary Search Code

The code below returns ***some value*** equal to target, but ***not necessarily the first value***. This is code from **`Collections.binarySearch`** rewritten to be close to the code you'll write***.***

[1] In making the "one more comparison" you must use the Comparator object that's a parameter to the method, e.g., checking that it returns zero to see if you've found the target.

Do not use this code except to see how to write code using a **`Comparator`**.

```
public static <T> int
binarySearch(List<T> list, T target, Comparator<T> comp) {
    int low = 0;
    int high = list.size()-1;

    while (low <= high) {
        int mid = (low + high)/2;
        T midval = list.get(mid);
        int cmp = comp.compare(midval,target);

        if (cmp < 0)
            low = mid + 1;
        else if (cmp > 0)
            high = mid - 1;
        else
            return mid; // target found
     }
     return -(low + 1);  // target not found
}
```

Testing BinarySearchLibrary Methods

You're given two classes to help verify that your methods are correct and meet performance requirements. The JUnit tests in **`TestBinarySearch`** can help you verify that your methods return the correct values. The output of running **`BinaryBenchmark`** can help verify both correctness and performance requirements. The output using a correct **`BinarySearchLibrary`** class is shown below. Examining the code in that method and the output here should help you understand the output. The values in both **`index`** columns should be the same. The **`cslow`** column is the number of comparisons made by the slow implementation of **`firstIndex`**. The **`cfast`** column is the number of comparisons made by **`firstIndex`**. Note that $\log_2(26000)$ is 14.666.

```
size of list = 26000
prefix index     index      cslow cfast

aaa   0          0          817   15
fff   5000       5000       693   16
kkk   10000      10000      568   16
ppp   15000      15000      443   16
uuu   20000      20000      318   15
zzz   25000      25000      194   16
```

Implementing BinarySearchAutocomplete

Once you've implemented the methods in class **`BinarySearchLibrary`**, the **`BinarySearchAutocomplete`** class is mostly finished. You'll still need to implement **`topMatches`**, a method required as specified in the **`Autocompletor`** interface.

You'll need to use the API exported by **`BinarySearchLibrary`** --- passing a **`List<Term>`** object and a **`Term`** object to the methods **`BinarySearchLibrary.firstIndex`** and **`lastIndex`** since they expect those types. Use **`Arrays.asList`** to create the list and create a **`Term`** from the String passed to topMatches. The weight for the **`Term`** doesn't matter.

Implementing topMatches Efficiently

The **`topMatches`** method requires that you return the weightiest ***`k`*** matches that match the prefix that's a parameter to **`topMatches`** --- note that ***`k`*** is a parameter to the method. If there are ***`M`*** terms that match the prefix, then the simple method of finding the ***`M`*** matches, copying them to a list, sorting them in reverse weight order, and then choosing the first ***`k`*** of them will run in the total of the times given below. Using this approach will thus have complexity/performance of **`O(log N + M log M)`**. An approach after the table shows how to use a size-limited priority queue to achieve a bound of **`O(log N + M log k)`** to find k matches. ***You must implement this approach***. You can use the code provided in **`BruteAutocomplete.topMatches`** as a model.

Complexity	Reason
`O(log N)`	Call **`firstIndex`** and **`lastIndex`**
`O(M log M)`	Sort all M elements that match prefix
`O(k)`	Return list of top k matches

It's possible of course that **`k < M`** and often **`k`** will be much less than **`M`**. Rather than sorting all **`M`** entries that match the prefix you can use a size-limited priority queue using the same idea that's used in the **`topMatches`** method from **`BruteAutocomplete`**. Reference the code there for ideas. ***You must implement this approach in the code you write***.
If you implement this priority queue approach you'll make **`topMatches`** run in **`O(log N + M log k)`** time instead of **`O(log N + M log M)`** using the ideas from the table above. In benchmarking there is no noticeable difference for the data files you're given for small values of **`M`**, though with larger values of **`M`** there will be a difference. In particular, when the prefix is the empty string, which matches every Term, there will be a difference.

Complexity	Reason

O(log N)	Call **firstIndex** and **lastIndex**
O(M log k)	Keep best **k** elements in priority queue
O(k)	Return list of top k matches

You'll write code to call **firstIndex** and **lastIndex** which are all possible matches. If there are no matches, return an empty list of **Term** objects. If there are matches, model the code you write on the **PriorityQueue** code from **BruteAutocomplete**, but insert **Term** objects between **firstIndex** and **lastIndex** instead of all Terms. You'll then take the best k matches (if there that many). See code

Testing BinarySearchAutocomplete

You're given a JUnit test class **TestBinarySearchAutocomplete** that you should run to verify your methods work correctly. You should also change the code in **AutocompleteMain** to use the **BinarySearchAutocomplete** class -- see the commented out lines as shown below. Then be sure that the output using the target auto matches the output shown at the beginning of this write-up.

```
21      final static String AUTOCOMPLETOR_CLASS_NAME = BRUTE_AUTOCOMPLETE;
22      //final static String AUTOCOMPLETOR_CLASS_NAME = BINARY_SEARCH_AUTOCOMPLETE;
23
```

HashListAutocomplete

Create a class named HashListAutocomplete that implements the Autocomplete interface. This method will provide an O(1) implementation of **topMatches** --- with a tradeoff of requiring more memory. The method outlined here is a hybrid of the approach outlined in the article referenced at the beginning of this write-up.

The class maintains a HashMap of every possible prefix (up to the number of characters specified by a constant **MAX_PREFIX** that you should set to 10. The key in the map is a

```
private static final int MAX_PREFIX = 10;
private Map<String, List<Term>> myMap;
private int mySize;
```

prefix/substring. The value is a weight-sorted list of **Term** objects that share that prefix. The diagram below shows part of such a HashMap. Three prefixes are shown as are the corresponding values are shown as a weight-sorted list of Term objects.

"ch"	("chat",50), ("chomp",40), ("champ",30), ("chocolate",10)
"cho"	("chomp",40), ("chocolate",10)
"cha"	("chat",50), ("champ",30)

You should create a constructor similar to those in the other implementations. The constructor body is one line: a call to **`initialize()`** though you'll need to throw exceptions just as the other implementations throw.

For each **`Term`** in initialize, use the first **`MAX_PREFIX`** substrings as a key in the map the class maintains and uses. For each prefix you'll store the Term objects with that prefix in an ArrayList that is the corresponding value for the prefix in the map.

After all keys and values have been entered into the map, you'll write code to sort every value, that is every ArrayList, corresponding to each prefix. You must use a **`Term.ReverseWeightOrder`** object to sort so that the list is maintained sorted from high to low by weight.

The implementation of **`topMatches`** can then be done in about five lines of code or fewer: if the prefix is in the map, get the corresponding value and return a sublist of the first k entries of this list. Here's code that can help:

```
List<Term> all = myMap.get(prefix);
List<Term>  list = all.subList(0, Math.min(k, all.size()));
```

(added 3/24) ***All prefixes passed to `topMatches` should be shortened to `MAX_PREFIX` characters if necessary.***

You'll also need to implement the required **`sizeInBytes`** method. This should account for every **`Term`** object and every String/key in the map. Use the implementations of **`sizeInBytes`** in the other **`Autocomplete`** classes as a model. Each string stored contributes **`BYTES_PER_CHAR*length`** to the bytes need. Each double stored contributes **`BYTES_PER_DOUBLE`**. For strings, you'll account for every Term and for every key in the map.

Analysis

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. On ola's computer the first few lines are what's shown below for "**`data/threeletterwords.txt`**". The unlabeled "search" is for an empty string "" which matches every string stored. These numbers are for a file of every three letter word "aaa, "aab", … "zzy", "zzz", not actual words, but 3-character strings.

```
init time: 0.006699   for BruteAutocomplete
init time: 0.004799   for BinarySearchAutocomplete
init time: 0.07067    for HashListAutocomplete
search     size  #match BruteAutoc     BinarySear HashListAu
           17576 50          0.00238732 0.00219437 0.00019249
           17576 50          0.00056931 0.00136807 0.00000449
a          676   50          0.00044899 0.00015267 0.00000443
a          676   50          0.00042797 0.00013736 0.00000575
b          676   50          0.00051954 0.00015502 0.00000640
```

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 **`matchSize`** is specified in the method runAM (for run all matches)
3. Explain why the last for loop in **`BruteAutocomplete.topMatches`** uses a **`LinkedList`** (and not an **`ArrayList`**) **AND** why the **`PriorityQueue`** uses **`Term.WeightOrder`** to get the top k heaviest matches -- rather than using **`Term.ReverseWeightOrder.`**
4. Explain why **`HashListAutocomplete`** uses more memory than the other **`Autocomplete`** implementations. Be brief.

After completing the analysis questions you should push to Git and submit the entire project on Gradescope. Please note that Analysis is part of the assignment, and Gradescope is the only portal to submit the assignment, including the analysis. We reserve the right to reject your analysis submission if you forget to upload it to Gradescope, even if it's in your Git repo.

Submitting

You'll submit the code to Gradescope after pushing your program and **`analysis.txt`** file to GitHub.

Reflect

http://bit.ly/201spring19-auto-reflect

Grading

Points	Grading Criteria
4	Code for Term and Comparators
8	Code for BinarySearchLibrary firstIndex and lastIndex
6	Code for BinarySearchAutocomplete.topMatches
9	Code for HashListAutocomplete
1	API
6	Analysis code and questions answered. UTAs will grade and comment on this.
2	Comments and code style. UTAs will grade and comment on this.

We will map total points you earn to scores as follows. This means if you lose five points, you receive the same score as losing zero points: an A. We will record the letter grade as your grade for this assignment.

```
32-36:  A
26-31:  B
20-25:  C
14-19:  D
```