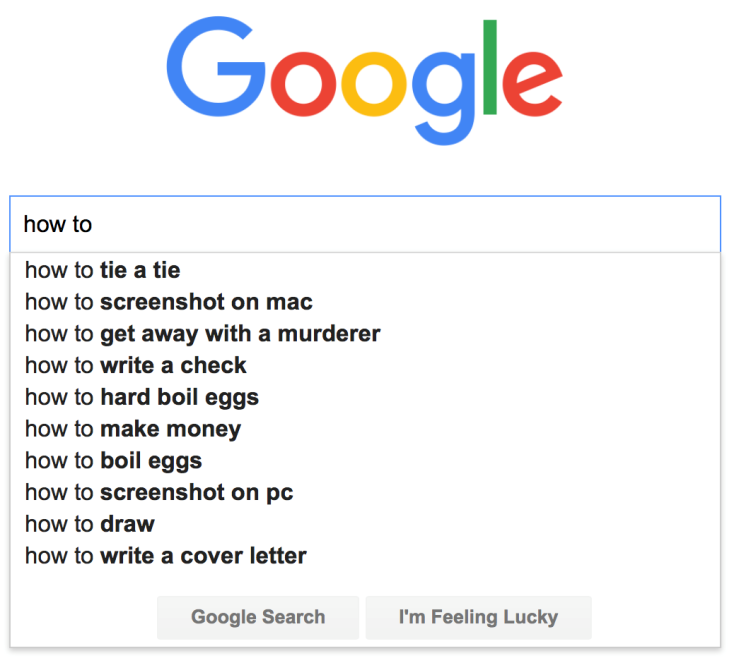
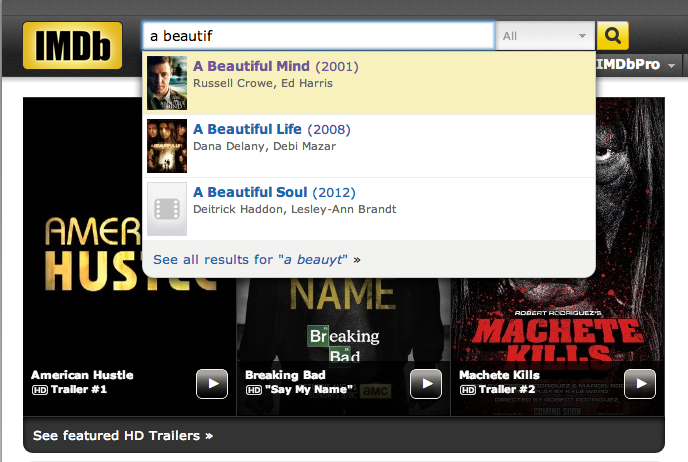
**Autocomplete**

***Autocomplete*** is pervasive and indispensable in modern applications. As the user types, the program predicts the complete *query* (typically a word or phrase) that the user intends to type.  Autocomplete is most effective when there are a limited number of likely queries. For example, the [Internet Movie Database](http://www.imdb.com/) uses it to display the names of movies as the user types; search engines use it to display suggestions as the user enters web search queries; cell phones use it to speed up text input.



In these examples, the application predicts how likely it is that the user is typing each query and presents to the user a list of the top-matching queries, in descending order of weight. These weights are determined by historical data, such as box office revenue for movies, frequencies of search queries from other Google users, or the typing history of a cell phone user. For the purposes of this assignment, you will have access to a set of all possible queries and associated weights (and these queries and weights will not change).

The performance of autocomplete functionality is critical in many systems. For example, consider a search engine which runs an autocomplete application on a server farm. According to one study, the application has only about 50ms to return a list of suggestions for it to be useful to the user. Moreover, in principle, it must perform this computation *for every keystroke typed into the search bar*, *for every user on the planet*.

In this assignment, you will implement autocomplete by *sorting* the terms by query string, *binary searching* to find all query strings that start with a given prefix; and *sorting* the matching terms by weight.

**Part 1: an autocomplete "term".** Write a data type Term.java that represents an autocomplete term: a query String and an associated whole-number weight.

You must implement the following API, which supports comparing terms by three different orders: [lexicographic order](http://docs.oracle.com/javase/7/docs/api/java/lang/String.html#compareTo(java.lang.String)) by query string (the natural order); in descending order by weight (an alternate order); and lexicographic order by query string, but using only the first r characters (a family of alternate orderings). The last order may seem a bit odd, but you will use it in *Part 3* to find all query strings that start with a given prefix (of length r).

**public class Term implements Comparable<Term>**

**{**

// Initializes a term with the given query string and weight.

**public Term(String query, long weight)**

// Compares the two terms in descending order by weight.

**public static Comparator<Term> byReverseWeightOrder()**

// Compares the two terms in lexicographic order, for the first *r* characters

**public static Comparator<Term> byPrefixOrder(int r)**

// Compares the two terms in lexicographic order by query.

**public int compareTo(Term other)**

// Returns a string representation of this term in the following format:

// the weight, followed by a tab, followed by the query.

// (the GIU requires the format specified above)

**public String toString()**

// unit testing (required)

**public static void main(String[] args)**

**}**

You MUST include tests of all methods in this class' main method. For example, you should test the toString method, sorting by natural order, sorting via both Comparators, etc. Do not proceed to the next section unless you are *sure* EVERYTHING is working properly. Use the Arrays class' methods for sorting.

**Part 2: binary search.** When binary searching a sorted array that contains more than one key equal to the search key, the client may want to know the index of either the *first* or the *last* such key. Accordingly, implement the following API (the Arrays class does not have this functionality):

**public class Searcher**

**{**

// Returns index of first term in *a* that equals the search key, or -1 if none

**public static int firstIndexOf(Term[] a, Term key, Comparator<Term> comp)**

// Returns index of last term in *a* that equals the search key, or -1 if none

**public static int lastIndexOf(Term[] a, Term key, Comparator<Term> comp)**

// unit testing (required)

**public static void main(String[] args)**

**}**

**Note:** when binary searching, the standard algorithm will return *any* index that matches the key in a data set with more than one matching value. In other words, the index returned won't necessarily be the first or last index - it will be the first *encountered*. The simplest way of getting the first or last index with a binary search is as follows:

* Perform a binary search the find the index of the key
* Perform a linear search from this index to walk to the end of the matching keys (forwards or backwards as required)

Of course, the worst-case time complexity for this algorithm is O(n) (for a list that contains *only* equal values). Very crafty programmers can modify the standard binary search algorithm, using a more specific definition of matching, to maintain the O(log n)-ness of a binary search.

You MUST include tests of all methods in this class' main method. Do not proceed to the next section unless you are *sure* everything is working properly

**Part 3: autocomplete.** In this part, you will implement a data type that provides autocomplete functionality for a given set of Strings and weights, using Term and Searcher. Algorithm help:

First sort the terms in lexicographic order; use a binary search to find all the query strings that start with a given prefix, then sort the matching terms in descending order by weight.

Complete your program by creating an immutable data type Autocomplete with the following API:

**public class Autocomplete**

**{**

// Initializes the data structure from the given array of terms.

// *terms* will contain all the terms used for matching what the user enters

// to autocompleted suggestions

**public Autocomplete(Term[] terms)**

// Returns all terms that start with *prefix*, in descending order of weight.

**public Term[] allMatches(String prefix)**

// Returns the number of terms that start with the given prefix.

**public int numberOfMatches(String prefix)**

// unit testing (required)

**public static void main(String[] args)**

**}**

**Input format.** Provided is an input file for testing. The file consists of an integer N followed by N pairs of query strings and non-negative weights. There is one pair per line, with the weight and string separated by a tab. A weight can be any integer between 0 and 263 − 1 (a valid long). A query string can be an arbitrary sequence of Unicode characters, including spaces (but not newlines).

* The file [wiki.txt](ftp://ftp.cs.princeton.edu/pub/cs226/autocomplete/wiktionary.txt) contains the 10,000 most common words in Project Gutenberg, with weights proportional to their frequencies. Sample:

10000

5627187200 the

3395006400 of

2994418400 and

2595609600 to

1742063600 in

1176479700 i

1107331800 that

1007824500 was

879975500 his

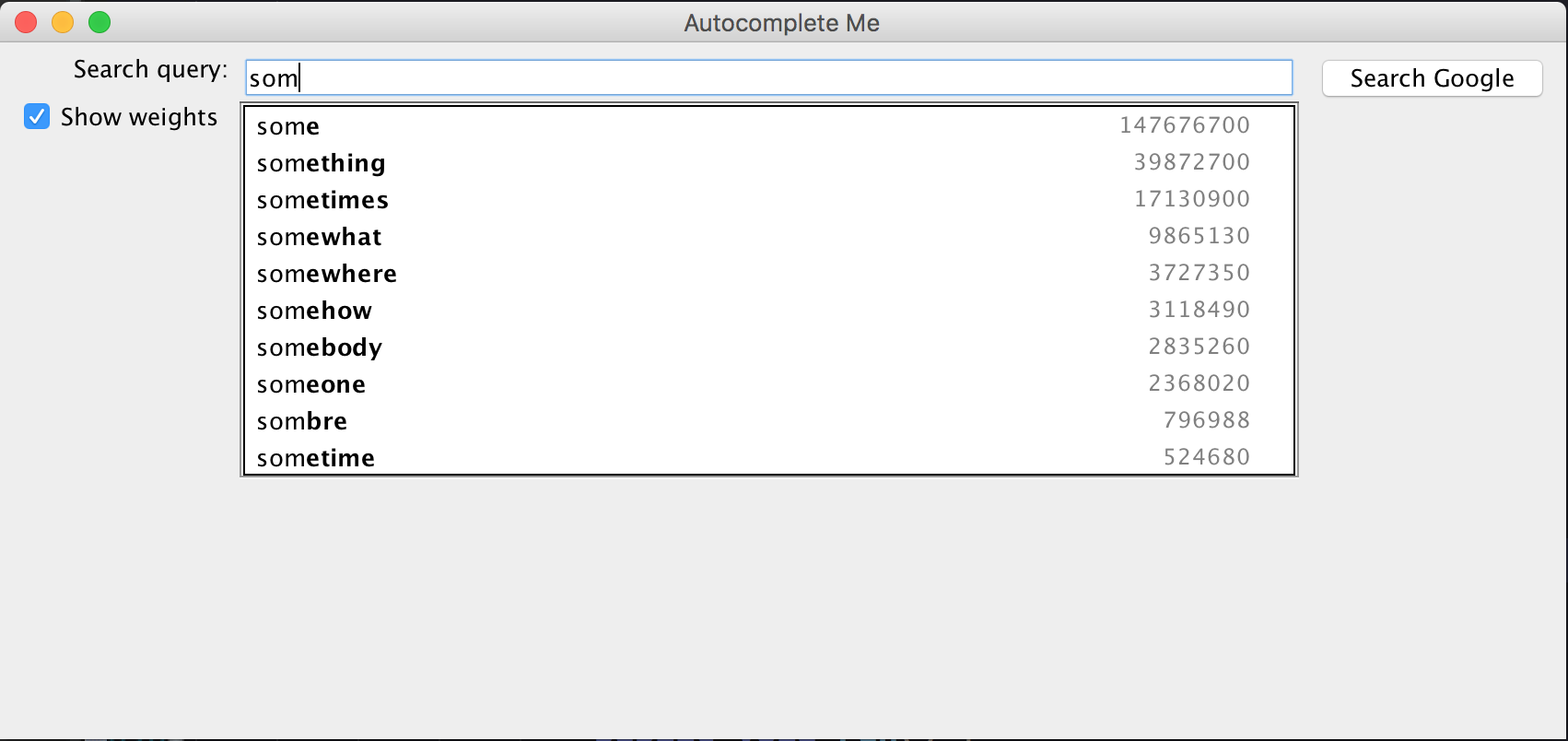
...

A sample client that takes the name of an input file and an integer k and creates a console-based auto-complete program can be found in the lab folder. It reads the data from the file, then it reads an autocomplete query from standard in, and prints out the top k matching terms in descending order of weight.

Here are a few sample executions, given the "wikti.txt" terms list and a max number of suggestions of 5:

|  |  |  |
| --- | --- | --- |
| **auto**  619695 automobile  424997 automatic  **comp**  13315900 company  7803980 complete  6038490 companion  5205030 completely  4481770 comply  **the**  5627187200 the  334039800 they  282026500 their  250991700 them  196120000 there |  |  |

Finally, import the AutocompleteGUI.java file into your project. This class contains the name of a file (that stores the terms the autocomplete program is using for suggestions) and an integer k (the number of suggestions to display) and provides a GUI for the user to enter queries. It presents the top k matching terms in real time. When the user selects a term, the GUI opens the results from a Google search for that term in the default browser.



*Adapted from the* ***Autocomplete Me*** *assignment   
http://introcs.cs.princeton.edu/java/assignments/autocomplete.html*