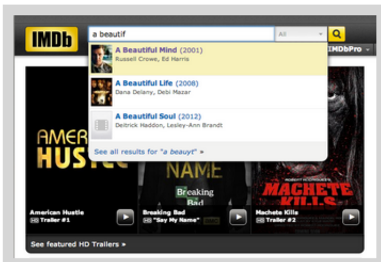
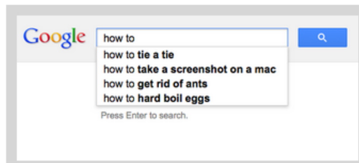
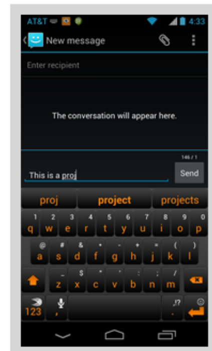
Goal: Implement *autocomplete* feature for a given set of strings and nonnegative weights, ie, given a prefix, find and list all strings in the set that start with the prefix, in descending order of weights.

Background: Autocomplete is an important feature of many 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 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; and 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* and *for every user*!

In this assignment, you will implement the Autocomplete feature by sorting the queries in lexicographic order; using binary search to find the set of queries that start with a given prefix; and sorting the matching queries in descending order by weight.

Problem 1. (*Autocomplete Term*) Implement an immutable comparable data type called `window` that represents an autocomplete term: a string query and an associated real-valued weight. You must implement the following API, which supports comparing terms by three different orders: lexicographic order by query; in descending order by weight; and lexicographic order by query but using only the first r characters. The last order may seem a bit odd, but you will use it in Problem 5 to find all terms that start with a given prefix (of length r).

`Term(String query)`	constructs a term given the associated query string, having weight 0
`Term(String query, long weight)`	constructs a term given the associated query string and weight
`String toString()`	returns a string representation of this term
`int compareTo(Term that)`	returns a comparison of this term and `other` by query
`static Comparator<Term> reverseWeightOrder()`	returns a comparator for comparing two terms in reverse order of their weights
`static Comparator<Term> prefixOrder(int r)`	returns a comparator for comparing two terms by their prefixes of length `r`

Corner Cases:

- The constructor should throw a `NullPointerException("query is null")` if *query* is `null` and an `IllegalArgumentExceptio`
  if *weight* < 0.
- The `prefixOrder()` method should throw an `IllegalArgumentException("Illegal r")` if $r < 0$.

Performance Requirements:

- The string comparison methods should run in time $T(n) \sim n$, where n is number of characters needed to resolve the comparison.

```
× ~/workspace/autocomplete
$ javac -d out src/Term.java
$ java Term data/baby-names.txt 5
Top 5 by lexicographic order:
11      Aaban
5       Aabha
11      Aadam
11      Aadan
12      Aadarsh
Top 5 by reverse-weight order:
22175   Sophia
20811   Emma
18949   Isabella
18936   Mason
18925   Jacob
```

Problem 2. (*Binary Search Deluxe*) When binary searching a sorted array that contains more than one key equal to the search key, the calling program may want to know the index of either the first or the last such key. Accordingly, implement a library called `BinarySearchDeluxe` with the following API:

`static int firstIndexOf(T[] a, T key, Comparator<T> c)`	returns the index of the first key in `a` that equals the search `key`, or -1, according to the order induced by the comparator `c`
`static int lastIndexOf(T[] a, T key, Comparator<T> c)`	returns the index of the last key in `a` that equals the search `key`, or -1, according to the order induced by the comparator `c`

Corner Cases:

- Each method should throw a `NullPointerException("a, key, or c is null")` if any of the arguments is `null`. You may assume that the array `a` is sorted (with respect to the comparator `c`).

Performance Requirements:

- Each method should should run in time $T(n) \sim \log n$, where n is the length of the array `a`.

```
× ~/workspace/autocomplete
$ javac -d out src/BinarySearchDeluxe.java
$ java BinarySearchDeluxe data/wiktionary.txt love
firstIndexOf(love) = 5318
lastIndexOf(love)  = 5324
frequency(love)    = 7
$ java BinarySearchDeluxe data/wiktionary.txt coffee
firstIndexOf(coffee) = 1628
```

```
lastIndexOf(coffee)  = 1628
frequency(coffee)    = 1
$ java BinarySearchDeluxe data/wiktionary.txt java
firstIndexOf(java) = -1
lastIndexOf(java)  = -1
frequency(java)    = 0
```

Problem 3. (*Autocomplete*) In this problem, you will implement a data type that provides autocomplete functionality for a given set of string and weights, using `window` and `BinarySearchDeluxe`. To do so, *sort* the terms in lexicographic order; use *binary search* to find the set of terms that start with a given prefix; and sort the matching terms in descending order by weight. Organize your program by creating an immutable data type called `Autocomplete` with the following API:

`Autocomplete(Term[] terms)`	constructs an autocomplete data structure from an array of `terms`
`Term[] allMatches(String prefix)`	returns all terms that start with `prefix`, in descending order of their weights
`int numberOfMatches(String prefix)`	returns the number of terms that start with `prefix`

Corner Cases:

- The constructor should throw a `NullPointerException("terms is null")` if *terms* is `null`.
- Each method should throw a `NullPointerException("prefix is null)"` if *prefix* is `null`.

Performance Requirements:

- The constructor should run in time $T(n) \sim n \log n$, where n is the number of terms.
- The `allMatches()` method should run in time $T(n) \sim \log n + m \log m$, where m is the number of matching terms.
- The `numberOfMatches()` method should run in time $T(n) \sim \log n$.

```
× ~/workspace/autocomplete
$ javac -d out src/Autocomplete.java
$ java Autocomplete data/wiktionary.txt 5
Enter a prefix (or ctrl-d to quit): love
First 5 matches for "love", in descending order by weight:
  49649600      love
  12014500      loved
  5367370       lovely
  4406690       lover
  3641430       loves
Enter a prefix (or ctrl-d to quit): coffee
All matches for "coffee", in descending order by weight:
  2979170       coffee
Enter a prefix (or ctrl-d to quit): java
No matches
Enter a prefix (or ctrl-d to quit):
```

Data: The `data` directory contains sample input files for testing. The first line specifies the number of terms and the following lines specify the weight and query string for each of those terms. For example, here is an input file:

```
× ~/workspace/autocomplete
$ cat data/wiktionary.txt
10000
    5627187200   the
    3395006400   of
           ...   ...
        392402   wench
        392323   calves
```

Visualization Programs: The program `AutocompleteVisualizer` accepts the name of a terms file and an integer k as command-line arguments, provides a GUI for the user to enter queries, and presents the top k matching terms from the file in real time.

```
× ~/workspace/autocomplete
$ javac -d out src/AutocompleteVisualizer.java
$ java AutocompleteVisualizer data/wiktionary.txt 5
```

Files to Submit:

1. `Term.java`
2. `BinarySearchDeluxe.java`
3. `Autocomplete.java`
4. `notes.txt`

Before you submit your files, make sure:

- You do not use concepts from sections beyond *Quick Sort*.
- Your code is clean, well-organized, uses meaningful variable names, includes useful comments, and is efficient.
- You edit the sections (`#1` mandatory, `#2` if applicable, and `#3` optional) in the given `notes.txt` file as appropriate.

In section `#1`, for each problem, state its goal in your own words and describe your approach to solve the problem along with any issues you encountered and if/how you managed to solve those issues.

Acknowledgement: This assignment is an adaptation of the Autocomplete assignment developed at Princeton University by Matthew Drabick and Kevin Wayne.