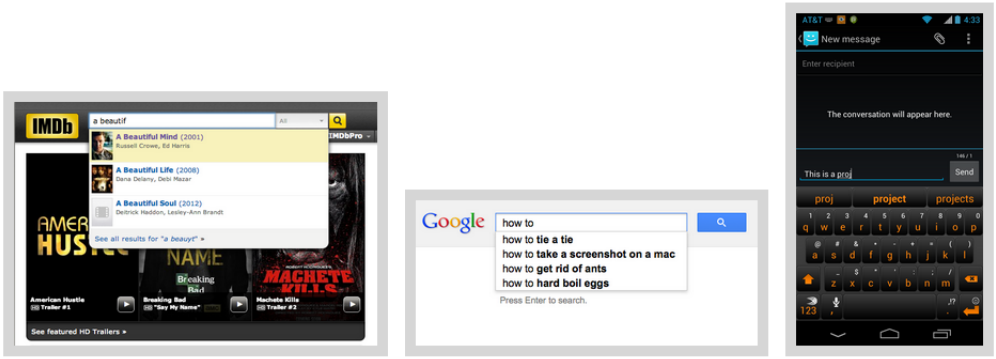
The purpose of this assignment is to write a program to implement *autocomplete* for a given set of N strings and nonnegative weights. That is, given a prefix, find all strings in the set that start with the prefix, in descending order of weight.

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; 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 autocomplete 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 `Term` in `Term.java` 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 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 Problem 3 to find all terms that start with a given prefix (of length r).

method	description
`Term(String query)`	initialize a term with the given query string and zero weight
`Term(String query, long weight)`	initialize a term with the given query string and weight
`static Comparator<Term> byReverseWeightOrder()`	compare the terms in descending order by weight
`static Comparator<Term> byPrefixOrder(int r)`	compare the terms in lexicographic order but using only the first r characters of each query
`int compareTo(Term that)`	compare the terms in lexicographic order by query
`String toString()`	a string representation of the term

Corner cases. The constructor should throw a `java.lang.NullPointerException` if query is `null` and a `java.lang.IllegalArgumentException` if weight is negative. The `byPrefixOrder()` method should throw a `java.lang.IllegalArgumentException` if r is negative.

Performance requirements. The string comparison functions should take time proportional to the number of characters needed to resolve the comparison.

```
$ java Term data/cities.txt 5
Top 5 by lexicographic order:
2200    's Gravenmoer, Netherlands
19190   's-Gravenzande, Netherlands
134520  's-Hertogenbosch, Netherlands
3628    't Hofke, Netherlands
246056  A Coru a, Spain
Top 5 by reverse-weight order:
14608512        Shanghai, China
13076300        Buenos Aires, Argentina
12691836        Mumbai, India
12294193        Mexico City, Distrito Federal, Mexico
11624219        Karachi, Pakistan
```

Problem 2. (*Binary Search Deluxe*) 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 a library of static methods `BinarySearchDeluxe.java` with the following API:

method	description
`static <Key> int firstIndexOf(Key[] a, Key key, Comparator<Key> comparator)`	the index of the first key in $a[]$ that equals the search key, or -1 if no such key
`static <Key> int lastIndexOf(Key[] a, Key key, Comparator<Key> comparator)`	the index of the last key in $a[]$ that equals the search key, or -1 if no such key

Corner cases. Each static method should throw a `java.lang.NullPointerException` if any of its arguments is `null`. You should assume that the argument array is in sorted order (with respect to the supplied comparator).

Performance requirements. The `firstIndexOf()` and `lastIndexOf()` methods should make at most $1 + \lceil \log N \rceil$ compares in the worst case, where N is the length of the array. In this context, a *compare* is one call to `comparator.compare()`.

```
$ java BinarySearchDeluxe data/wiktionary.txt cook
3
```

Problem 3. (*Autocomplete*) In this part, you will implement a data type that provides autocomplete functionality for a given set of string and weights, using `Term` 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 `Autocomplete` in `Autocomplete.java` with the following API:

method	description
`Autocomplete(Term[] terms)`	initialize the data structure from the given array of terms
`Term[] allMatches(String prefix)`	all terms that start with the given prefix, in descending order of weight
`int numberOfMatches(String prefix)`	the number of terms that start with the given prefix

Corner cases. The constructor and each method should throw a `java.lang.NullPointerException` if its argument is `null`.

Performance requirements. The constructor should make proportional to $N \log N$ compares (or better) in the worst case, where N is the number of terms. The `allMatches()` method should make proportional to $\log N + M \log M$ compares (or better) in the worst case, where M is the number of matching terms. The `numberOfMatches()` method should make proportional to $\log N$ compares (or better) in the worst case. In this context, a *compare* is one call to any of the `compare()` or `compareTo()` methods defined in `Term`.

```
$ java Autocomplete data/cities.txt 7
M
<enter>
12691836        Mumbai, India
12294193        Mexico City, Distrito Federal, Mexico
```

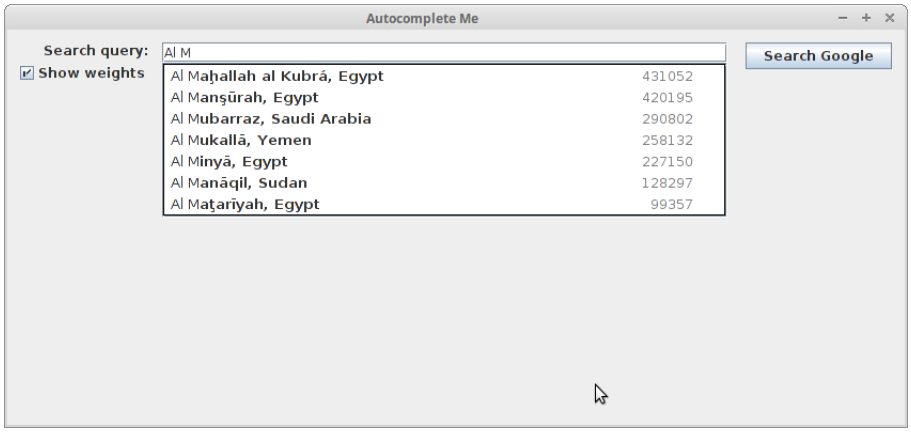
```
10444527        Manila, Philippines
10381222        Moscow, Russia
3730206 Melbourne, Victoria, Australia
3268513 Montr al, Quebec, Canada
3255944 Madrid, Spain
Al M
<enter>
431052  Al Ma allah al Kubr , Egypt
420195  Al Man  rah , Egypt
290802  Al Mubarraz, Saudi Arabia
258132  Al Mukall , Yemen
227150  Al Miny , Egypt
128297  Al Man qil , Sudan
99357   Al Ma ar yah , Egypt
<ctrl-d>
```

Data Under the `data` directory, we provide sample input files for testing. Each file consists of an integer N followed by N pairs of query strings and nonnegative 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 $2^{63}-1$. A query string can be an arbitrary sequence of Unicode characters, including spaces (but not newlines). For example, `wiktionary.txt` contains the 10,000 most common words in Project Gutenberg, with weights proportional to their frequencies.

```
$ more wiktionary.txt
10000
   5627187200   the
   3395006400   of
   2994418400   and
   2595609600   to
   1742063600   in
   1176479700   i
   1107331800   that
   1007824500   was
    879975500   his
                ...
       392323   calves
```

Visualization Client The program `AutocompleteGUI` takes the name of a file and an integer k as command-line arguments 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 up the results from a Google search for that term in a browser.

```
$ java AutocompleteGUI data/cities.txt 7
```



Files to Submit

1. `Term.java`

2. `BinarySearchDeluxe.java`
3. `Autocomplete.java`
4. `report.txt`

Before you submit:

- Make sure your programs meet the input and output specifications by running the following command on the terminal:

  ```
  $ python run_tests.py -v [<problems>]
  ```

  where the optional argument `<problems>` lists the problems (`Problem1`, `Problem2`, etc.) you want to test; all the problems are tested if no argument is given.

- Make sure your programs meet the style requirements by running the following command on the terminal:

  ```
  $ check_style <program>
  ```

  where `<program>` is the `.java` file whose style you want to check.

- Make sure your report isn't too verbose, doesn't contain lines that exceed 80 characters, and doesn't contain spelling/grammatical mistakes

Acknowledgements This project is an adaptation of the Autocomplete Me assignment developed at Princeton University by Matthew Drabick and Kevin Wayne.