Goal 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 called Term 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 3 to find all terms that start with a given prefix (of length $r$).

Corner Cases

- The constructor should throw a NullPointerException("query is null") if query is null and an IllegalArgumentException("Illegal weight") if weight < 0.
- The `byPrefixOrder()` 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.

```bash
~/workspace/project3
$ 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
```

**Directions:**
- **Instance variables:**
  - Query string, `String query`.
  - Query weight, `long weight`.
- `Term(String query)` and `Term(String query, long weight)`
  - Initialize instance variables to appropriate values.
- `toString()`
  - Return a string containing the weight and query separated by a tab.
- `compareTo(Term other)`
  - Return a negative, zero, or positive integer based on whether `this.query` is less than, equal to, or greater than `other.query`.
- `static Comparator<Term> byReverseWeightOrder()`
  - Return an object of type `ReverseWeightOrder`.
- `static Comparator<Term> byPrefixOrder(int r)`
  - Return an object of type `PrefixOrder`.
- `Term :: ReverseWeightOrder`
  - `int compare(Term v, Term w)`
    * Return a negative, zero, or positive integer based on whether `v.weight` is less than, equal to, or greater than `w.weight`.
- `Term :: PrefixOrder`
  - Instance variable:
    * Prefix length, `int r`.
  - `PrefixOrder(int r)`
    * Initialize instance variable appropriately.
  - `int compare(Term v, Term w)`
* Return a negative, zero, or positive integer based on whether a is less than, equal to, or greater than b, where a is a substring of v of length \( \min(r, v.query.length()) \) and b is a substring of w of length \( \min(r, w.query.length()) \).

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 called `BinarySearchDeluxe` with the following API:

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>static int firstIndexOf(Key[] a, Key key, Comparator&lt;Key&gt; c)</code></td>
<td>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</td>
</tr>
<tr>
<td><code>static int lastIndexOf(Key[] a, Key key, Comparator&lt;Key&gt; c)</code></td>
<td>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</td>
</tr>
</tbody>
</table>

**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 run in time \( T(n) \sim \log n \), where \( n \) is the length of the array a.

Directions:

- `static int firstIndexOf(Key[] a, Key key, Comparator<Key> c)`
  - Modify the standard binary search such that when \( a[mid] \) matches key, instead of returning \( mid \), remember it in, say \( index \) (initialized to -1), and adjust \( hi \) appropriately.
  - Return index.

- `static int lastIndexOf(Key[] a, Key key, Comparator<Key> c)` can be implemented similarly.

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 called `Autocomplete` with the following API:

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>Autocomplete(Term[] terms)</code></td>
<td>constructs an autocomplete data structure from an array of terms</td>
</tr>
<tr>
<td><code>Term[] allMatches(String prefix)</code></td>
<td>returns all terms that start with prefix, in descending order of their weights.</td>
</tr>
<tr>
<td><code>int numberOfMatches(String prefix)</code></td>
<td>returns the number of terms that start with prefix</td>
</tr>
</tbody>
</table>

**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 $T(n) \sim \log n$.

Directions:
• Instance variable:
  – Array of terms, Term[] terms.
• Autocomplete(Term[] terms)
  – Initialize this.terms to a defensive copy (ie, a fresh copy and not an alias) of. terms
  – Sort this.terms in lexicographic order.
• Term[] allMatches(String prefix)
  – Find the index $i$ of the first term in terms that starts with prefix.
  – Find the number of terms (say $n$) in terms that start with prefix.
  – Construct an array matches containing $n$ elements from terms, starting at. index $i$
  – Sort matches in reverse order of weight and return the sorted array.
• int numberOfMatches(String prefix)
  – Find the indices $i$ and $j$ of the first and last term in terms that start with prefix.
  – Using the indices, compute the number of terms that start with prefix, and return that value.

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

**Visualization Program** The program AutocompleteVisualizer accepts the name of a 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 in real time.

```
x ~/workspace/project3
$ java AutocompleteVisualizer data/wiktionary.txt 5
```

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

**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 outside of what has been taught in class.
- Your code is adequately commented, follows good programming principles, and meets any specific requirements such as corner cases and running times.
- You edit the sections (#1 mandatory, #2 if applicable, and #3 optional) in the given notes.txt file as appropriate. Section #1 must provide a clear high-level description of the project in no more than 200 words.