CPS 350: Assignment 3

Two weeks, **200 pts**

This is a team project. At most three students are in one team

One submission per team. No late submission will be accepted

Receive 5 bonus points if turn in the complete work without errors at least one day before deadline

Receive an *F* for this course if any academic dishonesty occurs

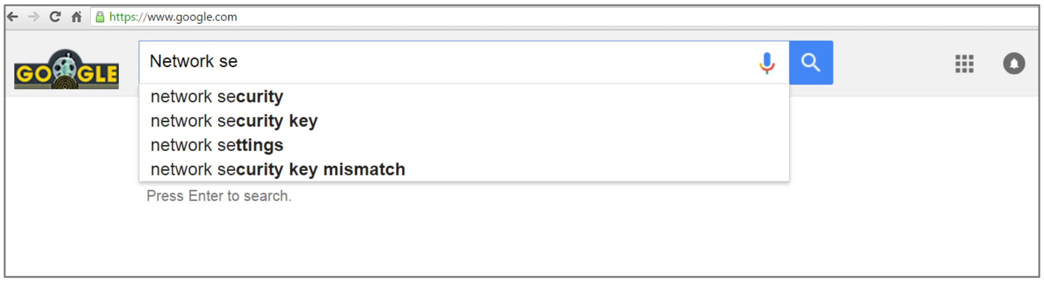
1. Purpose

The purpose of this assignment is to implement sorting algorithms for the autocomplete application.

2. Description

Write a program to implement *autocomplete* for a given set of *N terms*, where a term is a query string and an associated nonnegative weight. That is, given a prefix, find all queries that start with the given prefix, in descending order of weight.

Autocomplete is pervasive 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* 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.



https://www.google.com

Network se

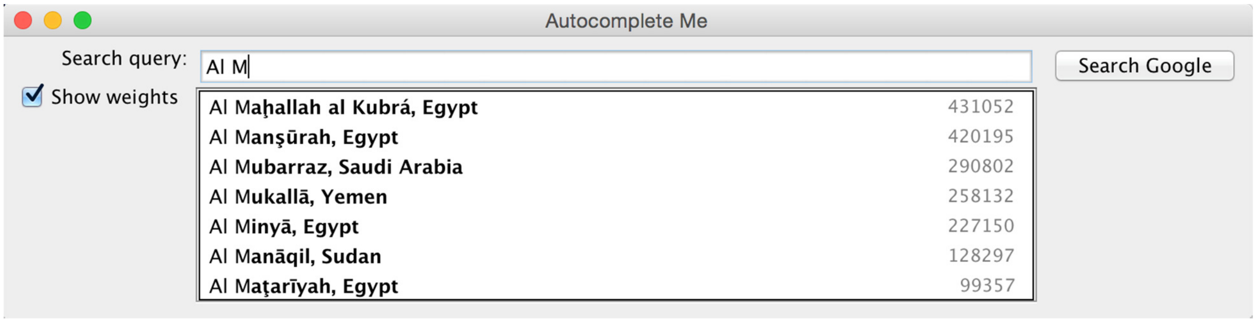
network **security**
network **security key**
network **settings**
network **security key mismatch**

Press Enter to search.

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 50*ms* 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 terms by query string (with running time $O(N \log N)$ in sorting, or even better, where N is the of terms); *binary searching* to find all query strings that start with a given prefix (with running time $O(\log N)$); and *sorting* the matching terms by weight (with running time $O(M \log M)$ in sorting, where M is the number of matching terms). Finally display results for the user. The following shows the top seven queries (city names) that start with AI M with weights equal to their populations.



Search query: Al M

☑ Show weights

Al Maḩallah al Kubrá, Egypt	431052
Al Manşūrah, Egypt	420195
Al Mubarraz, Saudi Arabia	290802
Al Mukallā, Yemen	258132
Al Minyā, Egypt	227150
Al Manāqil, Sudan	128297
Al Maţarīyah, Egypt	99357

2.1. Part 1: autocomplete term (60 pts)

Write an immutable data type **Term.java** that represents an autocomplete term: a query string and an associated integer 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 *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 but using only the first
r characters of each query. */
    public static Comparator<Term> byPrefixOrder(int r)

    /* Compares the two terms in lexicographic order by query.  */
    public int compareTo(Term that)
    // Returns a string representation of this term in the following format:
    // weight (i.e., ??.toString()), followed by a tab, followed by query.
    public String toString()
}
```

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.

2.2. Part 2: binary search (30 pts)

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:

```
public class BinarySearchDeluxe {

    /* Returns the index of the first key in a[] that equals the search key,
or -1 if no such key. */
    public static <Key> int firstIndexOf(Key[] a, Key key, Comparator<Key>
comparator)

    /* Returns the index of the last key in a[] that equals the search key,
or -1 if no such key. */
    public static <Key> int lastIndexOf(Key[] a, Key key, Comparator<Key>
comparator)
}
```

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 $1 + \lceil \log_2 N \rceil$ compares in the worst case, where N is the length of the array. In this context, a *compare* is one call to `comparator.compare()`.

2.3. Part 3: autocomplete (70 pts)

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 all query strings that start with a given prefix; and *sort* the matching terms in descending order by weight. Organize your program by creating an data type `Autocomplete` with the following API:

```
public class Autocomplete {  // implement sorting algorithm in this class

    /* Initializes the data structure from the given array of terms. */
    public Autocomplete(Term[] terms)

    /* Returns all terms that start with the given prefix, in descending
order of weight. */
    public Term[] allMatches(String prefix)
}
```

Corner cases. The constructor should throw a `java.lang.NullPointerException` if its argument is `null` or if any of the entries in its argument array are `null`. 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. In this context, a *compare* is one call to any of the `compare()`or `compareTo()` methods defined in `Term`.

2.4. Input format for testing (30 pts)

We provide a number of 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).

- The file wiktionary.txt contains the 10,000 most common words in Project Gutenberg, with weights proportional to their frequencies.
- The file cities.txt contains over 90,000 cities, with weights equal to their populations.

```
% more wiktionary.txt% more cities.txt
10000                93827
   5627187200  the         14608512  Shanghai, China
   3395006400  of          13076300  Buenos Aires, Argentina
   2994418400  and         12691836  Mumbai, India
   2595609600  to          12294193  Mexico City, Distrito Federal, Mexico
   1742063600  in          11624219  Karachi, Pakistan
   1176479700  i           11174257  İstanbul, Turkey
   1107331800  that        10927986  Delhi, India
   1007824500  was         10444527  Manila, Philippines
```

```
879975500   his          10381222  Moscow, Russia
             ...                     ...
   392323   calves              2  Al Khāniq, Yemen
```

Below is a sample client that takes the name of an input file and an integer *k* as command-line arguments. It reads the data from the file; then it repeatedly reads autocomplete queries from standard input, and prints out the top *k* matching terms in descending order of weight.

```
public static void main(String[] args) {

    // always print messages on screen when debugging
    // System.out.println(…);

    // read in the terms from a file
    String filename = args[0];  // first argument from command line
    In in = new In(filename); ←
    int N = in.readInt();                     defined in In.java, to read data
    Term[] terms = new Term[N];               from files and URLs
    for (int i = 0; i < N; i++) {
        long weight = in.readLong();          // read the next weight
        in.readChar();                        // scan past the tab
        String query = in.readLine();         // read the next query
        terms[i] = new Term(query, weight);   // construct the term
    }

    // read in queries from standard input and print the top k matching terms
    int k = Integer.parseInt(args[1]); // 2nd argument from command line
    Autocomplete autocomplete = new Autocomplete(terms);
    while (StdIn.hasNextLine()) { ←               defined in StdIn.java, to
        String prefix = StdIn.readLine();         read data from keyboard
        Term[] results = autocomplete.allMatches(prefix);
        for (int i = 0; i < Math.min(k, results.length); i++)
            System.out.println(results[i]);
    }
}
```

Here are a few sample executions:

```
% java Autocomplete wiktionary.txt 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
```

first argument

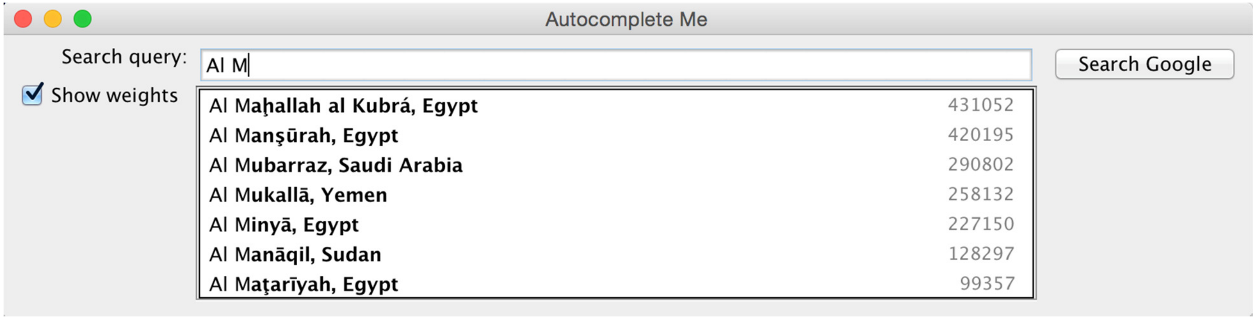
2nd argument

```
% java Autocomplete cities.txt 7
M
    12691836   Mumbai, India
    12294193   Mexico City, Distrito
Federal, Mexico
    10444527   Manila, Philippines
    10381222   Moscow, Russia
     3730206   Melbourne, Victoria,
Australia
     3268513   Montréal, Quebec, Canada
     3255944   Madrid, Spain
Al M
      431052   Al Maḩallah al Kubrá, Egypt
      420195   Al Manşūrah, Egypt
      290802   Al Mubarraz, Saudi Arabia
      258132   Al Mukallā, Yemen
      227150   Al Minyā, Egypt
```

```
                                        128297  Al Manāqil, Sudan
                                         99357  Al Maţarīyah, Egypt
```

Interactive GUI (optional, but fun and no extra work):
Compile `AutocompleteGUI.java`. The program 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 cities.txt 7
```



Search query: Al M

☑ Show weights

```
Al Maḩallah al Kubrá, Egypt          431052
Al Manşūrah, Egypt                   420195
Al Mubarraz, Saudi Arabia            290802
Al Mukallā, Yemen                    258132
Al Minyā, Egypt                      227150
Al Manāqil, Sudan                    128297
Al Maţarīyah, Egypt                   99357
```

3. Grading notes

If your program does not compile, you receive zero points for that program. Additional deductions:

1. (5 points) Your code does not follow the style guide discussed in class/textbook.
2. (30 points) Your code does not have author name, date, purpose of this program, **comments** on the variables and methods, etc.

4. Turn in

One submission for a team. **Zip/submit your entire project**. You may **NOT** call any library functions other than those in `java.lang` and `java.util`. That is, you have to implement your own sorting algorithm. Finally, submit **a report file (10 points)** and answer the following questions:

a) At least two sample runs (i.e., snapshots that show you successfully run your program).
b) Known bugs of this assignment.
c) Describe any serious problems you encountered.
d) List any other comments here. Feel free to provide any feedback on how much you learned from doing the assignment, and whether you enjoyed doing it.