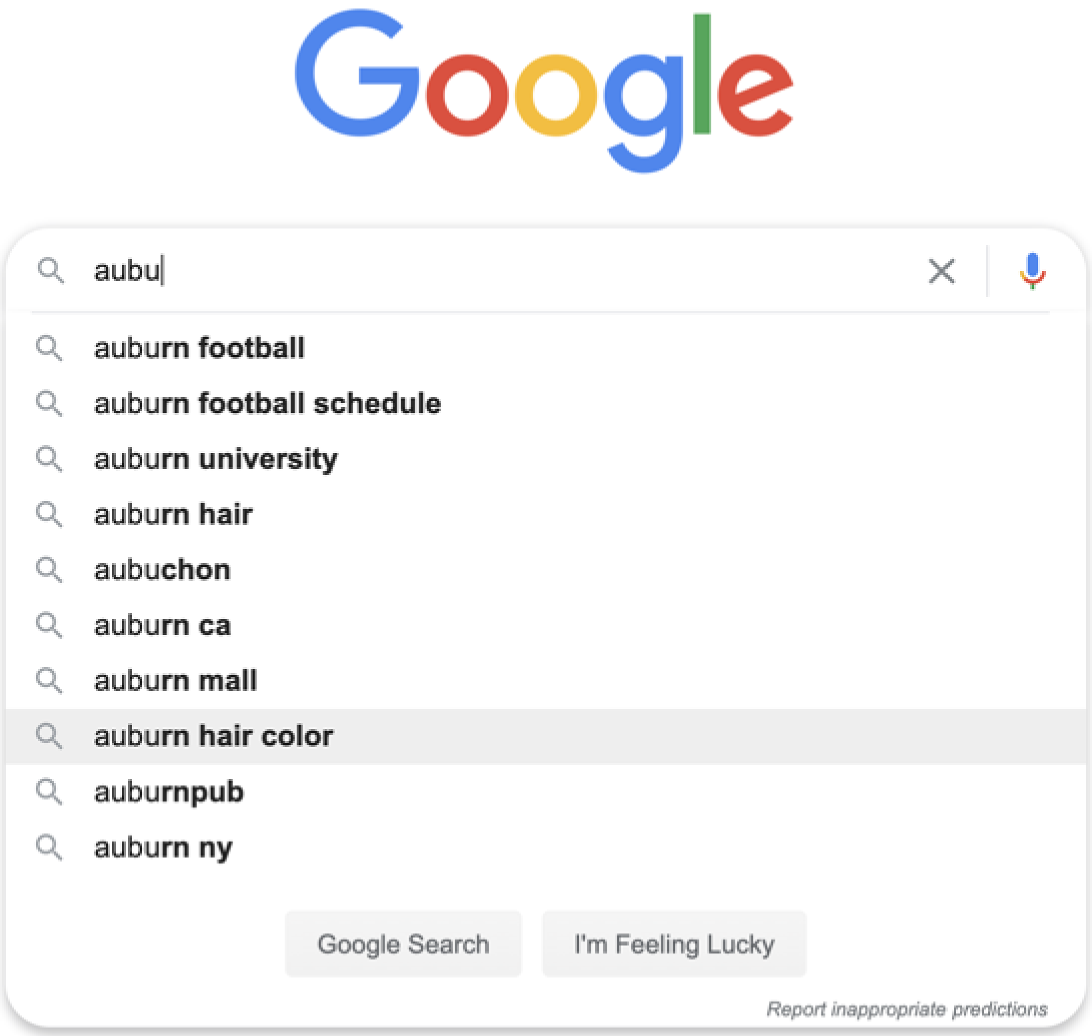
Autocomplete



auburn football
auburn football schedule
auburn university
auburn hair
aubuchon
auburn ca
auburn mall
auburn hair color
auburnpub
auburn ny

Overview and Context

Autocomplete, or **predictive [text**, is now a common feature in software applications including search [engines, messaging apps, source code editors and more. Its more aggressive [cousin **autocorrect** can be extremely frustrating and [sometimes funny. But when done well, providing predicted [completions to something currently being typed can be helpful and can make [the task go faster.

We're probably all familiar with how autocomplete works: As a user types, the software predicts the most likely *completions* of what the user is typing and offers these as selectable options. This works best when the set of possible completions is constrained and there is a clear basis for making predictions of which are most likely. For example, when a user types `myString.` , a source code editor that offers code completion can confidently offer a set of method signatures as possible completions based on the methods available for the object's type. Deciding which method is most likely is a different matter, but we could guess at how that might be done.

Similarly, when a user types “See you ”, a messaging app could offer completion options such as “later” and “soon”, perhaps based on a history of past messages the user has typed.

An autocomplete application not only has to provide useful and accurate predictions, but it has to do so quickly. If making predictions is too slow or if its operation in any way slows down the actual task being performed, autocomplete becomes a liability that would quickly be disabled by the user. User experience studies suggest that response and rendering of interactive results must be done within a 50ms to 0.1 second timeframe. For an autocomplete application to scale up while meeting that hard time constraint, the underlying algorithms will have to have low-order time complexity.

The importance of both **correctness** (making useful predictions) and **efficiency** (making predictions very quickly) is demonstrated at scale in the autocomplete applications used by search engines like Google Search. Google Search must compute and render meaningful predictions *with each keystroke* for billions of searches each day. Now *that’s* scalability.

So how would an autocomplete application actually work? Well, that’s exactly what this assignment will help us to figure out.

Problem Description

You must develop the components of an autocomplete application. To keep things simple, we will assume that our autocomplete application will work in the context of something like a search engine. Autocomplete will take in a string (we’ll call it a **prefix**) and will output a list of likely completions (we’ll call each a **query**). So, a user could type a portion of a query and autocomplete would offer a list of predictions of the completed query that the user intends to complete. The strings can contain any Unicode character except newline.

Our autocomplete application can only work in one context at a time. For example, if we want to let users search for movie titles, we would have to load needed movie title data first. Once loaded, autocomplete can predict possible full movie titles based on a given prefix. Likewise, if we want to provide word completion similar to messaging apps, our autocomplete application would have to load needed word data first. Once loaded, autocomplete can predict possible whole words based on a given prefix.

The data that the autocomplete application needs will be a set of (query, weight) pairs that represent all possible completions. Each query will be a complete search query (like a complete movie title) that a user might want to search for. Each weight will be a non-negative integer that is a distinguishing attribute of the query that will be used for the purposes of ranking queries: The larger the weight, the more “likely” the query. For example, in a word-based autocomplete for a messaging app the individual queries would be words, and weights would be the frequency of occurrence; thus making more frequently used words the more likely predictions of autocomplete.

Here are the first few lines of a data source of English words.

```
the     23135851162
of      13151942776
and     12997637966
to      12136980858
a       9081174698
in      8469404971
for     5933321709
is      4705743816
on      3750423199
that    3400031103
```

Each line consists of a word (query) and an integer (weight). The integer records the total number of occurrences of the associated word in some large corpus of English text. Note the data is arranged in descending order of weight. This data shows that “the” is the most frequently occurring English word, appearing over 23 billion times in the corpus from which the data was drawn. So, if “t” is the prefix, the autocomplete should return “the”, “to”, and “that” - in that order - as the first three predicted completions.

Solution Approach and Framework

We will develop our autocomplete application in terms of three classes: `Term`, `BinarySearch`, and `Autocomplete`.

The `Term` class

`Term` is used to represent a (query, weight) pair. You must write this class so that it is completely consistent with the following API.

```java
public class Term implements Comparable<Term> {

    /**
     * Initialize a term with the given query and weight.
     * This method throws a NullPointerException if query is null,
     * and an IllegalArgumentException if weight is negative.
     */
    public Term(String query, long weight) { }

    /**
     * Compares the two terms in descending order of weight.
     */
    public static Comparator<Term> byDescendingWeightOrder() { }

    /**
     * Compares the two terms in ascending lexicographic order of query,
     * but using only the first length characters of query. This method
     * throws an IllegalArgumentException if length is less than or equal
     * to zero.
     */
    public static Comparator<Term> byPrefixOrder(int length) { }

    /**
     * Compares this term with the other term in ascending lexicographic order
     * of query.
     */
    @Override
    public int compareTo(Term other) { }

    /**
     * Returns a string representation of this term in the following format:
     * query followed by a tab followed by weight
     */
    @Override
    public String toString(){ }

}
```

Note that `Term` supports comparison by three different orders: (1) lexicographic order (the natural order), descending order of weight (one alternate ordering), and lexicographic order using only the first `length` characters of the query string (a family of alternate orderings). By providing three different comparison methods, we are able to sort terms in three different orders. All three orders will be important in delivering the autocomplete functionality.

The `BinarySearch` class

`BinarySearch` provides two search methods, both of which are variants of the classic binary search presented in lecture. 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 last matching key.

```
public class BinarySearch {

    /**
     * Returns the index of the first key in a[] that equals the search key,
     * or -1 if no such key exists. This method throws a NullPointerException
     * if any parameter is null.
     */
    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 exists. This method throws a NullPointerException
     * if any parameter is null.
     */
    public static <Key> int lastIndexOf(Key[] a, Key key, Comparator<Key> comparator) { }
}
```

Both methods must use the binary search algorithm and make on the order of *log N* comparisons to `a[middle]` where *N* is the number of elements in `a[]` .

The `Autocomplete` class

`Autocomplete` uses the `Term` and `BinarySearch` class to provide complete autocomplete functionality for a given set of strings and weights.

```
public class Autocomplete {

        /**
         * Initializes a data structure from the given array of terms.
         * This method throws a NullPointerException if terms is null.
         */
        public Autocomplete(Term[] terms) { }

        /**
         * Returns all terms that start with the given prefix, in descending order of wei
         * This method throws a NullPointerException if prefix is null.
         */
        public Term[] allMatches(String prefix) { }

}
```

The constructor must store the contents of `terms` in its own internal array and then sort this array in natural order (lexicographic order of query). The `allMatches` method must use the binary search methods to identify the range of methods that begin with the given prefix and return these elements in an array sorted in descending order of weight. This returned array represents the predicted completions for the given prefix.

Acknowledgments

This assignment is based on a problem originally described by Matthew Drabick and Kevin Wayne in 2014.