Dr. Scot Morse
Department of Computer Science
Western Oregon University

May 21, 2015

LAB 4 — String Searching

Do you know how when you are trying to search for something on Google and it pops up with possible search strings while you're still typing?

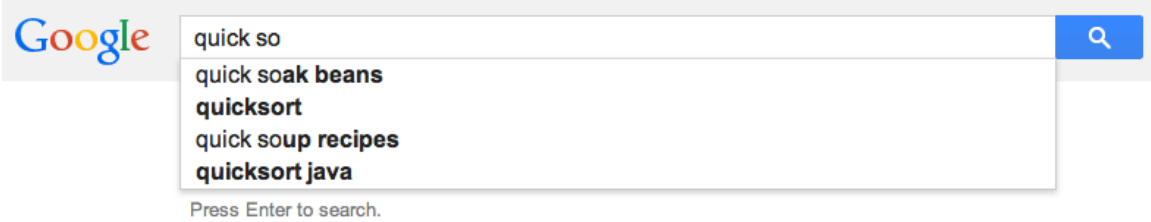

Figure 1: Google's supposedly helpful search suggestions

Yeah. I absolutely hate it! My kids love it because they are just learning to type and it saves them time (and maybe helps them learn how to spell?). But I know what the hell I'm searching for and they can just butt the hell out![1] Same goes with the search as you type, but that's a different story.

Anyway, ever wonder how they do that? Well, for sure algorithms and heaps of statistics on what people are searching for. I would think they are showing you 'popular' matches to the letters you're typing, i.e. given the letters you've typed, what other similar search strings have been entered in the past, and since people are commonly thinking about the same kinds of things, they'll show you those.

Let's do something similar, but a little easier. Let's say we have a massive database of words, in fact every word someone would think of wanting to write (at least if they're writing English, and in the U.S., and not using technospeak, or medical terms, and not with slang, ... and nothing after the year 1913). As they type we want to show them possible words that match what they've already written. That will save them gobs of time won't it?

Figure 2 shows the app, with a search already in progress. To help folks out we'll show the shorter words first, and then make sure they're in alphabetical order from there.

Once search results are returned, the user may click on one word and its definition will be shown.

The code on the class web page will get you the, very plain, GUI and a place to start.

The basic idea here is that we will have a dictionary of words. We wish to search it very quickly and return matches. But we don't have to match exactly. Let's assume it is more helpful to show similar words, perhaps that are spelled correctly for the incorrectly spelled version that the user typed in.

What kind of a data structure could you use? How would you use it? Here are some things to think about:

- Large data set. Ours here will have only 86,036 words, but we would want it to be applicable to hundreds of thousands or millions of records.
- Very fast response time. We want results 'as you type', just like the Google search bar. This means millisecond searches, not seconds.
- Exact matches.
- Inexact matches. Words that start with a certain string. Words that have the same number of characters but some of the characters are wrong.
- Must store the definition as well as the string itself. You should be thinking *key-value pairs* on this one.

[1] ok, I didn't really mean that. You're right, I would never say that to Siri. Can I have my search results now? Pretty please?

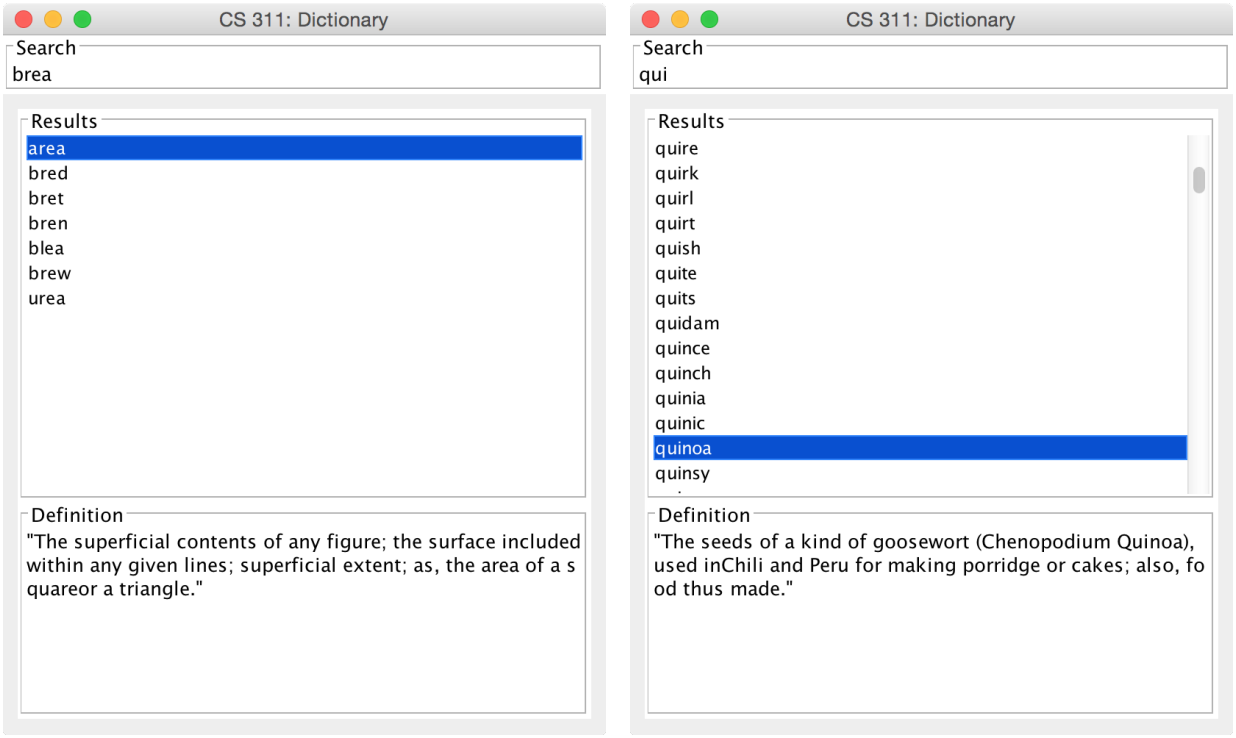

Figure 2: Our app in action (*quinoa*, yum!) in various stages of completion. The left figure has implemented approximate searching for strings of the same length. The right figure includes longer strings.

- String based.

At this point, it is useful to ask yourself why a hash table won't work.

The answer is a String based data structure called a Trie. Read section 5.2 before going any farther. Think about how a Trie will help you solve this problem in an efficient manner.

Problem 1.

Implement a Ternary Search Trie (TST), in a class called `HybridTST.java`, with the following interface (see Algorithm 5.5 for comparison and see the code for Javadocs):

```
public interface TrieInterface<E>
{
    public E get(String key);
    public boolean contains(String key);
    public void put(String key, E val);
    public int size();
    public boolean isEmpty();
    public Iterable<String> keys();
    public Iterable<String> keysWithPrefix(String prefix);
    public Iterable<String> keysThatMatch(String pattern);
    public String longestPrefixOf(String query);
    public int getTreeHeight();
    public double getAverageNodeDepth();
}
```

You must implement this TST using the Hybrid approach discussed on pages 750-751. This hybrid approach uses an array or table to store a large multi-way node at the root of the 'tree'. For your trie,

assume that the only characters allowed in a key come from the EXTENDED_ASCII character set (see page 699), for which only 8-bit values are allowed. Note, this does not mean we are using a particular `CharSet` as in the last lab; rather, regardless of the character set (remember Java defaults to Unicode), the allowed characters have to come from the range indicated.

Test your TST using the provided JUnit test file.

Problem 2.

Extend the provided GUI application to create a fast searchable dictionary. Use your `HybridTST` to store all the words and definitions from the `dictionary.json` dictionary.[2] See the `ReadJSON.java` file for an example of how to parse a simple JSON file. Note, you'll need the `javax.json-1.0.4.jar` library for the JSON parsing classes to work.

Here's the behavior we want to support. The user begins typing in a word. With just one letter typed in there will be many results, so let's not do anything just yet. After the next letter there are probably still too many results, so wait. After three or so then we want to start running searches and returning results. This should be a live search, done immediately after each key press. If at any time the user hits the return key then you should run the search on what is typed and return the results, even if they're really big. Sort your results by length and then alphabetically for equal length words. It's OK to limit the results to a couple pages of scrolling if you want. I'm not sure how well the `JList` holds up to many entries, but give it a shot.

If what they've typed is spelled correctly and in the dictionary then return that word first in the search results. Then follow it with similar matches that they might have meant to type. If it is spelled incorrectly or not in the dictionary then return similar matches to help them.

If they spell it correctly then display the definition of the word below. If it is spelled incorrectly then display the definition of the first word in the search results. If the user clicks on a search result then display the definition of that word.

Your goal should be to write a nice usable application that is fast. There is a JUnit test for the TST, but we will grade the GUI manually by trying out searches. The best apps receive the most points.

Turn it in

We will turn this lab in on paper and electronically via the same turn-in web page as for Lab 1. Here you'll have multiple source code files to submit. When you're ready, select only those source code files you want to submit and zip them. Name your zip file `Lab4.zip`. Here are things to be mindful of.

- **Don't** put your code in a folder before you zip it.
- **Don't** put the `dictionary.json` text file in your zip.
- **Don't** zip your Eclipse folder.
- **Don't** export your code from Eclipse.
- **Don't** put any .class files in your zip.
- **Don't** use rar, 7z, cab, dmg, tgz, bzip2, xar or any other type of compressed archive.
- **Do** double check that the files you handed in are everything that is needed to run your program. Try unzipping it elsewhere and see if everything is there. Compile it to make sure.
- **Do** use the command line to compile and run your code, because that is what I will do.

Submitted by Dr. Scot Morse on May 21, 2015.

[2] This dictionary file contains the words and definitions from *Webster's Unabridged Dictionary, 1913 ed.* (`http://www.gutenberg.org/ebooks/29765`) that is very nicely processed into JSON format by a Julia program written by Adam Savitzky via GitHub: `https://github.com/adambom/dictionary`.