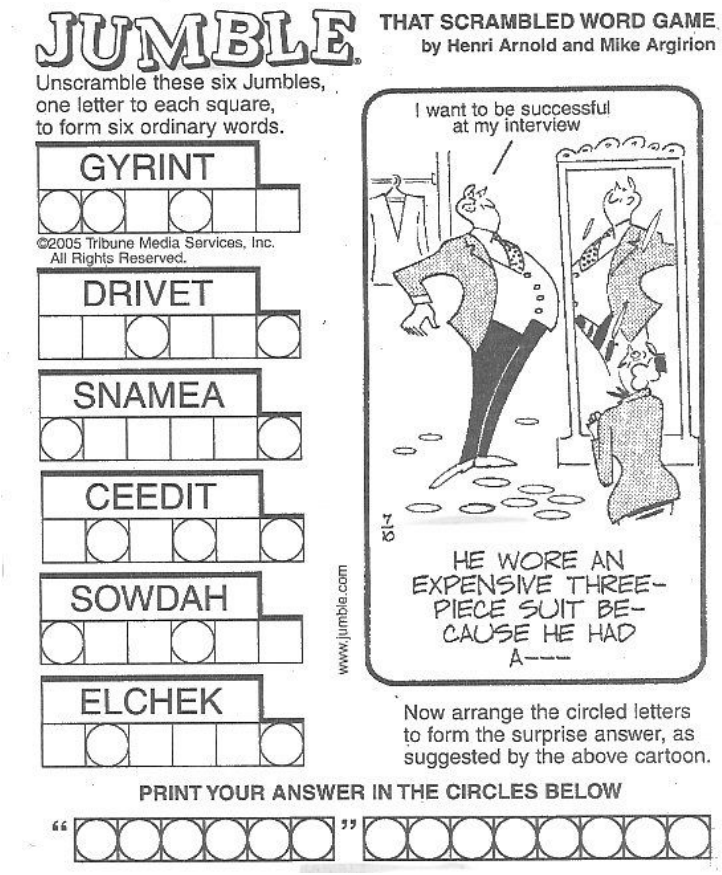
CS251 -
A Jumble Solving Program
Due: Wed, Oct 28 at 2:00pm

In this project you will use a clever application of hash tables to solve "word jumble" puzzles. You have probably seen these puzzles in the newspaper. An example is below.



Your program will implement a strategy described below (which utilizes Hash Tables (or Hash Maps)) to quickly find a list of English words that can be formed by rearranging a given sequence of letters.

Overview of Program Behavior

From the user's perspective, the program is a simple interactive loop which behaves as follows:

1. The user is asked to either:
    a. enter a string of characters (presumably a jumbled version of to one or more English).
    b. or enter a single period "." to terminate the program in which case, *the program first prints the statistics of the HashMap it used and then terminates.*
2. (Assuming the user hasn't decided to terminate the program), a list of *all* English words in a given dictionary that are rearrangements of the user input. The list can appear in any order. If there are no such English words, the program reports "no matches"
3. Goto 1.

Example run:

```
$ ./jumble
Enter a scrambled word or . to terminate: kstae
    Matching words:

            takes
            steak
            stake
            skate

Enter a scrambled word or . to terminate: tra
    Matching words:

            tar
            rat
            art

Enter a scrambled word or . to terminate: bcdz
    No matches.

Enter a scrambled word or . to terminate: .
    Goodbye
    <DISPLAY OF HMAP TABLE STATISTICS HERE>

$
```

Detail: Upper-Case vs. Lower-Case.

For simplicity, the program will convert everything to lower-case -- dictionary entries with upper-case letters will be converted to lower-case; user-input will be converted to lower-case prior to finding matches.

Algorithm

You are required to implement a specific Hash-Table/Hash-Map based algorithm to solve this problem.

A Strawman.
For sake of argument, suppose we built a Hash-Map from the given dictionary where each word in the dictionary becomes an entry in the Hash-Map (as a "KEY"). A Hash-Map like this works great for applications like spell-checking. But does it help us with the Jumble problem? Not much it seems.

For a given jumbled sequence of letters, we *could* enumerate all re-orderings of the sequence and for each of them, check to see if it is in the dictionary.

For example, if we were given "kstae" as in the previous discussion, how many re-orderings would we end up enumerating? The string is of length-5 and no letters are duplicated. Thus there are $5! = 120$ re-orderings we would enumerate (and 120 queries to the Hash Map).

If the given jumbled string is length 6 and all letters are distinct, then we would enumerate all $6! = 720$ reorderings. For example suppose the given jumbled string is "layber" which can be rearranged to form the words "barely", "barley" and "bleary" and we would have 717 queries to the hash-map that would fail.

So...this approach doesn't seem especially elegant or efficient.

Can we avoid explicit enumeration of all reorderings of the given string?

Yes.

A Better Approach.

Short summary of algorithm: From a given dictionary, build a Hash-Map in which

- the keys are sequences of characters/letters *in sorted order;* the keys are not in general themselves words and
- the value associated with each such key is a list of all words in the dictionary that are rearrangements of the value.

Each entry in the Hash-Map is a list of dictionary words which are rearrangements of each other (the VALUE) and the letter-level sort of these words which is used to identify the list (the KEY).

Thus, to find all words that can be formed from a given string, we first sort the letters of the given string and use it as a key to look up the list of words that can be formed from those letters (if any).

More Discussion

Two strings **s1** and **s2** are rearrangements of each other if and only iff **s1** and **s2** are composed of exactly the same letters and with the exact same frequency. For example **"aekst"**, **"stake"** are rearrangements of each other is TRUE. However **"apple"**, **"aalpe"** are not rearrangements of each -- both strings have the same letters, but not in the same frequency.

Now consider any two words that are rearrangements of each other -- like "stake" and "takes". Now consider the letter-level sorts of each of them. Let **sort(s)** represent the the rearrangement of string **s** such that the individual letters are in sorted order (for example, sort("happy") = "ahppy"**):**

sort("stake"): "aekst"
sort("takes"): "aekst"

It shouldn't come as a surprise that they both result in the same string when letter-level sorted -- after all, they have exactly the same letters in the same frequency.

So instead of storing actual words in your hash table you will store sorted re-orderings (as the "key" and a list of all words that have the same sorted ordering (as the "value"). In other words, the table

is implementing a function from sorted strings to sets of English words that are formed from exactly those letters.

Mathematically, the HashMap partitions the dictionary into "equivalence classes" or words that are anagrams of each other. Each such equivalence class has a unique "id string" -- their letter-level sorted version.

When the user enters a jumbled sequence of letters, you then sort the string and do a lookup on it to retrieve all words that can be formed by rearranging the letters.

For example, if the user enters "ketsa", you would use the sorted version "aekst" as the key for a hash table lookup which would return the English words "steak", "stake", "skate" and "takes"

Implementation

You have been provided with a pretty flexible C implementation of a hash-map. So you will just need to build the application program acting as a client of the hash-map module.

> [If you *really* want to implement your own hash-map, you can. However, the hash-map module that you come up with must be:
>
> - General purpose -- i.e., decoupled from the jumble application and usable for other applications.
> - Well designed and well documented.
>
>]

Command line args. Your program will be called jumble will have one required command line argument -- the dictionary file name. Example:

> **$ jumble dict.txt**

The dictionary file will simply contain one word per line for easy parsing.

Submission

You will submit a single archive file containing a makefile with a target called jumble. When we type:

```
$ make jumble
```

an executable named jumble should be produced. This means that you also have to include all hash-map source files necessary.