Computer Science I
Program 3: Jumble
Assigned: 2/23/12 (Thursday)
Due: 3/17/12 (Friday 11:55pm over WebCourses)

The Problem

Lindsay plays the daily "Jumble" in the Orlando Slantinel. After she learns the permutation algorithm in her CS1 class, she gets a brilliant idea: she can write a computer program to solve the daily "Jumble"! Her idea is as follows:

For each set of letters, simply create each permutation of them and check each of these permutations against the dictionary. Whenever one is found that matches, output it!

Obviously, carrying this algorithm out by hand would be very tedious, but a computer program could easily solve the problem in very little time!

Write a program that solves "Jumble." First, your program will read in the dictionary from a file called "dictionary.in" (This file will be provided for you.) This file will contain English words in alphabetized order. (Thus, there is no need to sort these words, you can store them already sorted as you read them in.)

You will read the rest of your input from the file "jumble.in". This file will contain a list of jumbled words. For each jumbled word, your program should use the permutation algorithm to create all permutations of the jumbled word, and then use a binary search to search for each permutation in the dictionary. Your program should output (to the screen) all permutations of the jumbled word that appear in the dictionary. If no permutations form a valid word, simply display a message to that effect.

Input File Specification (dictionary.in)

The dictionary file has a single integer, *n*, on its first line, specifying the number of words in the file. The next *n* lines contain one word in the dictionary each, in sorted alphabetical order.

Input File Specification (jumble.in)

The first line of the input file will contain a single integer, ***n***, representing the number of test cases in the file. Followed by a single integer, ***m***, designating whether to run a recursive or iterative algorithm (**1** corresponds to **recursive**, **2** corresponds to **iterative**). Each test case will follow, one per line. Each test case will just be an alphabetic string of lowercase letters only with fewer than 30 letters.

Output Specification

For each input case, print out a header with the following format:

```
Jumble #k:
```

where k represents the number word in the file ($1 \leq k \leq n$).

For each permutation of the letters in a case, print out a single line with the permutation. **Note: For words with repeated letters, it is fine for certain permutations to be printed more than once.**

After completing the output for that test case, print out a blank line. Note that if no permutations of the jumbled word appear in the dictionary, then you should print nothing out except for the input case header.

Implementation Restrictions

You must implement **BOTH** the **recursive** algorithm shown in class to generate all the permutations of the entered letters, in addition to the **iterative** permutation algorithm found here: http://www.cs.ucf.edu/~dmarino/ucf/cop3502/sampleprogs/

You must implement a binary search of the dictionary.

You must dynamically allocate an array of strings that stores the dictionary and free this memory when appropriate.

Sample Input File(jumble.in)

```
3
1
tca
zyfrpq
esuoh
```

Sample Output

```
Jumble #1
A permutation of tca that is a valid word is act.
A permutation of tca that is a valid word is cat.

Jumble #2:

Jumble #3:
A permutation of esuoh is a valid word is house.
```

Theoretical Analysis

You will analyze your code (both the recursive and iterative permutation algorithms) to determine a theoretical Big-O run-time. Please justify your answer with a sentence or two. This can be submitted in your comments on Webcourses or in a separate Word Document.

Deliverables

Turn in a single file, *jumble.c*, over WebCourses that solves the specified problem. Make sure to include ample comments and use good programming style, on top of the requirements that are given in the program description above. If you decide to make any enhancements to this program, clearly specify them in your header comment so the grader can test your program accordingly. Regardless of the enhancements you make, you should make it easy for him to grade the functionality specified in this description **(5pts maximum extra credit**, for permutations with no repeats and/or finding permutations of different lengths)