Programming Project 2: Recursion

Due date: Oct. 27, 11:59 PM EST.

Summary

This assignment is an exercise in writing a backtracking program using recursion; your program must use recursion to solve the given problem. The program that you are to write is run from the command line and is given at least one argument, which is the name of a file containing a list of words. We will call this file the ***dictionary***. If there is just one argument on the command line, the program enters an interactive loop in which it repeatedly prompts the user to enter a string of characters. The string may be any sequence of characters that can be typed on the keyboard. Henceforth we call this string the ***source word***. The objective of the program, having been given a source word, is to find all words that are rearrangements of one or more of the characters in the source word, that are also in the dictionary. For example, if the source word is

```
carat
```

and the dictionary contains the words `at`, `arc`, `art`, `car`, `cart`, `cat`, `rat`, and `tar`, then the program would display each of these words, one per line, in sorted order:

```
arc
art
at
car
cart
cat
rat
tar
```

The words it displays are words in the dictionary that are either substrings of the source word or words that are permutations of the substrings. We will call these words ***solutions***. A word is a solution if it is a substring of the source word or a permutation of a substring. The program must not display duplicate solutions.

The optional second command-line argument is the pathname of a file that contains a list of words to use as source words, one per line. If it is present, the program does not enter the above-described interactive loop. Instead it reads from this file and processes the words in it one by one. Further details follow below.

Objectives

The objectives of this first programming project are for you to master the following tasks:

- working with multi-file programs using separate compilation of the program units
- working with existing code
- reading data from input files
- writing data to output files
- using and understanding command line arguments
- writing a backtracking program using recursive functions

- using arrays and/or vectors
- writing class interfaces and their implementations

Many of the skills that you need to complete this project are based on the material covered in CSci 135, but some, like recursion, are topics taught in CSci 235.

Solving This Problem

The solution to this problem is not easy to derive on your own, and so I provide an outline and description of how to solve it. Your task is to convert this description into code.

One obvious way to solve this problem is to generate all possible words from the source word and check if each one is in the dictionary. This is called a ***brute force solution***. This, however, is extremely inefficient because, if we let N be the length of the source word, the number of possible words that are substrings of the word or permutations of these substrings is proportional to $N!$. If the dictionary has M words in it, the search for each of these words would require $M \log M$ steps. The total number of steps would be proportional to $N! \cdot M \log M$. This is unacceptable. Your program is not allowed to enumerate all possible words. It must be "smarter" than this.

Suppose that the source word S has length N. Let $0 < k < N$ and suppose that the string $w = a_0a_1a_2...a_k$ is a permutation of some of the letters in the source word. Suppose that $z = b_0b_1b_2...b_{N-k-1}$is a string consisting of the letters in S that are not in w. (It does not matter what order the letters are in z.) There are a few possibilities:

1. The word w itself is in the dictionary.
2. There is at least one word that starts with w in the dictionary but w itself is not in the dictionary.
3. No word is in the dictionary that starts with w.

To illustrate, suppose our source word is `carat` and that w= `car`. The first case would be that the dictionary contains the word `car`. The second would be that it contains the words `cart` and `carat` but not `car`. The third would be that it does not contain any words starting with `car`.

We can begin by initializing an empty list of solutions. Given w, we can first check whether w is in the dictionary. If it is, we add it to the list of solutions we have found. We next check whether w is a prefix of a word in the dictionary. If it is, then w is a partial solution to our problem and we want to extend it to a solution by picking each letter in z and appending it to w and checking whether this is either a solution or a partial solution. This is the recursive part of the algorithm - it grows the partial solution by appending a character to the end of the word and checking recursively that it is a prefix of a solution. If, on the other hand, w is not a prefix of any word in the dictionary, then there is no point in appending characters to it and recursively trying those words. Instead, we just return from this recursive call, passing back the result that w is not a prefix of any word in the dictionary. This is the backtracking step.

Using our `carat` example, suppose that w =`car` and z =`at`. Suppose `car` is in the dictionary. Then the algorithm would append `car` to the list of solutions. Whether or not `car` is in the dictionary, it next checks whether `car` is a prefix of a word in the dictionary. Suppose the dictionary contains `cart` but neither `cara` nor `carat`. The function that checks whether `car` is a prefix of a dictionary word returns that it is, so our algorithm first appends the `a` to `car` and recursively checks whether `cara` is a solution or partial solution. It discovers that it is not. After it does this, it removes the `a` from `cara`, appends the `t` and recursively checks whether `cart` is a solution or a partial solution. In this recursive call, it discovers that `cart` is a solution and appends `cart` to the list of solutions. But it then discovers that `cart` is not a prefix of any words in the dictionary (besides itself) so it makes no further recursive calls.

In short, the recursive function, given the non-null word w, does three things in this order:

- Checks whether w is in the dictionary;
- Checks whether w is a prefix of a word in the dictionary;

- If w is a prefix of a word in the dictionary, it tries appending each letter from the remaining characters of S not in w to see if this extends to a partial solution (i.e., recursively).

Your task is to convert this description to C++ code. You will discover that you need two different kinds of dictionary searches: one that checks whether a word is in the dictionary, and one that checks if it is a prefix of a word in the dictionary.

The Program Input

Your program is given the pathnames of one or two text files on its command line, as in

```
$ findwords ../../mydict
```

or

```
$ findwords   data/dictionary  testwords
```

You may assume that the first file named on the command line contains a *sorted list* of words, one per line. A word is any sequence of non-blank characters, including digits and punctuation marks. Thus, `#$12ghy` is a valid word and so is `under_score`. There will be no duplicate words in the file, and no uppercase letters in the file. Therefore, you will not find a word such as `Europe` in the file. The words are sorted in the standard collating sequence. For ordinary one-byte characters, this is just the ASCII ordering. You may assume that no word in the dictionary has more than 32 characters.

If a second argument is provided, this is also a list of words, one per line, with no word longer than 32 characters, and with no uppercase letters. We call this the ***source word file***. In the example above, the source word file is named `testwords`.

If no filename is supplied on the command line, it is an error. If either filename does not exist or cannot be opened for reading by the program, for any reason, it is an error. The program should display an error message if any of these conditions occur, and it should be as specific as possible. It is not enough for it to output something like "`could not open file`." The error message should be written to the standard error stream (`cerr` in C++, `stderr` in C) and then it should exit.

The program should store the dictionary in a suitable data structure in memory for efficient access, and if the second file is present, it should open it for reading but not store it in memory.

Computational Tasks

If there is just one argument, then after the program has read and stored the dictionary, it should repeatedly prompt the user to enter a string of at least 2 and at most 32 characters. If the user enters a word that is not of length between 2 and 32 characters, the program should display a suitable message to ask the user to try again. Once the user has entered a word of the correct length, it displays all words that are substrings or permutations of substrings of the user's word that are also in the dictionary. If the user enters any uppercase letters, the program should convert them to lowercase before it tries to form words. It should display these words in sorted order using the ASCII collating sequence. If there are duplicates they should not be printed.

When the program has finished displaying the words, it asks the user whether she wants to continue or to quit. If the user chooses to quit, the program terminates.

If there is a second argument, the program should not do anything interactively. Instead it should read the words from the second file one at a time and compute the valid substrings and permutations in the same way as is described in the preceding paragraph. For each source word, it should append the list of such words to an output file whose name is the source word file with an extension "`.out`". So, for the above example, it would append to the file `testwords.out`. It should first write the source word, then the list of its subwords, then a blank line, as in

```
carat
arc
```

```
art
at
car
cart
cat
rat
tar
```

Implementation Constraints

- I have provided the public part of a `Dictionary` class as an abstract class interface. You must create a concrete class that implements this class. The `Dictionary` class interface is at the end of this document. Your `Dictionary` class must encapsulate the dictionary data and provide the two methods in the interface. These are a constructor and a method that checks whether a prefix of a given word is a prefix of a word in the dictionary. You have not yet learned about virtual functions, but by the time you start this, you will have learned about them. You may safely ignore the `virtual` keyword and the "`=0`" for now.
- Your program needs to sort. You may pick any sorting algorithm that you know.
- Not required but strongly recommended is to create a class to represent a word and give that class functions to check the validity of the word and find its subwords that are in the dictionary.
- Your main program is responsible for opening input files, reading from and writing to files, and closing files. It is also responsible for checking the correctness of the command line arguments and issuing any error messages related to the command line. If you are not experienced in using command line arguments, please read my notes on the topic here:

  http://www.compsci.hunter.cuny.edu/~sweiss/resources/cmmdlineargs.pdf.
- Only the main program is allowed to read from or write to files or devices. Class methods must operate on streams.
- All class interfaces and implementations are to be in separate files from each other and from the main program.

Programming Rules

Your program must conform to the programming rules described in the Programming Rules document on the course website. It is to be your own work alone.

Grading Rubric

The program will be graded based on the following rubric.

- A program that cannot run because it fails to compile or link on a cslab host loses 80%. The remaining 20% will be assessed using the rest of the rubric below.
- Meeting the requirements of the assignment: 50%
- Performance 10%
- Design (modularity and organization) 15%
- Documentation: 20%
- Style and proper naming: 5%

This implies that a program that does not compile on a lab computer cannot receive more than 20 points. Some implementations might be more efficient in terms of running time than others. Although you will not have learned enough to write this program in a very efficient way, there are certain programming decisions that can lead to very poor performance or to acceptable performance. The performance component of 5% will be given if the program is reasonably efficient.

Submitting the Assignment

This assignment is due by the end of the day (i.e. 11:59PM, EST) on October 27, 2016. You must create a multiple-file solution. Create a directory named *username*`_project2`. Put all project-related source-code files into that directory. Do not place any executable files or object files into this directory. You will lose 1 point for each file that does not belong there. With all files in your directory, run the command

```
zip -r username_project2.zip ./username_project2
```

This will compress all of your files into the file named `username_project2.zip`.

Before you submit the assignment, make sure that it compiles and runs correctly on one of the `cslab` machines. Do not enhance your program beyond this specification. Do not make it do anything except what is written above.

You are to submit the zip file by running the program `submit235project,` which it requires two arguments: the number of the assignment and the pathname of your zip file: enter the command

```
/data/biocs/b/student.accounts/cs235_sw/bin/submit235project 2 username_project2.zip
```

where `username_project2.zip` is replaced by the name of your zip file. If you decide to make any changes and resubmit, just run the command again and it will replace the old file with the new one. Do not try to put your file into this directory in any other way - you will be unable to do this. Once the deadline has passed, you cannot do this. I will grade whatever version is there at the end of the day on the due date. You cannot resubmit the program after the due date.

The Dictionary Class Interface

```
#ifndef __DICTIONARY_H__
#define __DICTIONARY_H__

#include <string>
#include <vector>
#include <fstream>

using namespace std;

class Dictionary
{
public:
    /** Dictionary()
     *  Constructor fills dictionary object with words from the given input
     *  file stream.
     *  @pre   file stream must be opened for reading &&  must have one word
               per line and be in sorted order.
     *  @post  dictionary object is filled with words from file in sorted order.
     *  @param ifstream & file [inout]  stream to read
     */
    virtual Dictionary( ifstream &  file )=0;

    /** search()
     * @pre    keyword != NULL and prefix_length >= 0
     * @param  string keyword [in]       key to search for in dictionary
     * @param  size_t prefix_length [in] number of initial chars to match
     * @return int
     *         If prefix_length > 0, then if there is a word W in the dictionary
     *         such that:
     *        W[0..prefix_length-1] is identical to keyword[0..prefix_length-1]
     *         then it retutns its index in the dictionary, else -1.
     *         If prefix_length == 0, this returns an index if and only if the
     *         keyword matches a word in the list exactly, and -1 otherwise.
     */
    virtual int  search( string keyword,
                 size_t prefix_length
               )=0;
};
```