Stanford CS 106X: Anagrams

Assignment by Marty Stepp and Victoria Kirst. Based on a problem by Stuart Reges of U of Washington.

Description   Implementation   Style   FAQ

This problem focuses on exhaustive search and recursive backtracking. This is an **individual assignment**. Write your own solution and do not work in a pair/group on this program.

Files and Links:

Project Starter ZIP
(open **Anagrams.pro**)

Turn in:
anagramsolver.cpp

Demo JAR
(COMING SOON)

Homework Survey

output logs:
📄 log #1
📄 log #2
📄 log #3
📄 log #4

Problem Description:

An *anagram* is a word or phrase made by rearranging the letters of another word or phrase. For example, "midterm" and "trimmed" are anagrams. If you ignore spaces and capitalization, a multi-word phrase can be an anagram of some other word or phrase. For example, "Clint Eastwood" and "old west action" are anagrams.

In this part of the assignment, you will write recursive code to use a dictionary to find and print all anagram phrases that match a given word or phrase. You are provided with a file **anagrammain.cpp** that contains the `main` function to prompt the user for phrases. Below is a partial sample log of execution; user input is in **blue bold**. See the links area above for several complete logs, and use the graphical console window's **Compare Output** feature to verify your program's output.

```
Welcome to the CS 106X Anagram Solver!
Dictionary file name (blank for dict1.txt)? dict1.txt
Reading dictionary ...

Phrase to scramble (or Enter to quit)? Barbara Bush
Max words to include (Enter for none)? 3
Searching for anagrams ...

{"abash", "bar", "rub"}
{"abash", "rub", "bar"}
{"bar", "abash", "rub"}
```

```
{"bar", "rub", "abash"}
{"rub", "abash", "bar"}
{"rub", "bar", "abash"}
Total anagrams found: 6
```

You must write the following <u>recursive</u> function in **anagramsolver.cpp**:

```
int findAnagrams(const string& phrase, int max, const Set<string>& dictionary)
```

In this function you should recursively find and print all anagrams that can be formed using all of the letters of the given phrase, and that include at most max words total, in alphabetical order and in the format shown in this document and our example logs. You should also return the total number of anagrams found.

Your function must show the anagrams in the same format as our log. A simple way to do this is to build up your answer in some kind of **collection**. You can print the collection and it will have the right format.

You are passed a Set representing a dictionary of all possible words that appear in your phrase. For example, if you are passed the phrase "hairbrush", a max of 3, and a dictionary corresponding to **dict1.txt**, you should produce the following console output, in exactly this order and format. You would also return 8.

```
{"bar", "huh", "sir"}
{"bar", "sir", "huh"}
{"briar", "hush"}
{"huh", "bar", "sir"}
{"huh", "sir", "bar"}
{"hush", "briar"}
{"sir", "bar", "huh"}
{"sir", "huh", "bar"}
```

If the max value passed is 0, you should print **all anagrams** regardless of how many words they contain. (We suggest that as you are developing this code, you initially ignore the max and just print all anagrams, and only once this is working, go back and add the code to enforce the max constraint.)

You should throw an **int exception** if the max value passed is negative. An empty string generates no output.

The provided main program can read its input from different dictionary files. The default is a small dictionary **dict1.txt** to make testing easier. Once your code works with this small dictionary, test it with larger dictionaries such as the provided **dict2.txt**, **dict3.txt**, and the largest, **dictionary.txt**.

Though **loops** are allowed for this problem, your fundamental algorithm must be recursive and not based on looping to find the entire anagram. You must use recursion to handle the self-similar aspects of the problem.

Implementation Details:

Generate all anagrams of a phrase using exhaustive searching and **recursive backtracking**. Many backtracking algorithms involve examining all combinations of a set of choices. In this problem, the choices are the words that can be formed from the phrase. A "decision" involves choosing a word for part of the phrase and recursively exploring what further words can be chosen for the rest of the phrase. If you find a collection of words that use up all of the letters in the phrase, it should be printed as output.

Part of your grade will be based on the **efficiency** of your algorithm. You should not explore words that cannot be formed from the currently remaining letters of your phrase. If your algorithm gets stuck and is unable to match the remaining letters, or if you exceed the maximum number of words allowed, your code should backtrack immediately.

The following diagram shows a partial decision tree for generating anagrams of the phrase "barbara bush". Notice that some paths of the recursion lead to dead ends. For example, if the recursion chooses "aura" and "barb", the letters remaining to use are [bhs], and no choice available uses these letters, so it is not possible to generate any anagrams beginning with those two choices. In such a case, your code should backtrack and try the next path.

Note that the same word can appear more than once in an anagram. For example, from "barbara bush" you might extract the word "bar" twice.
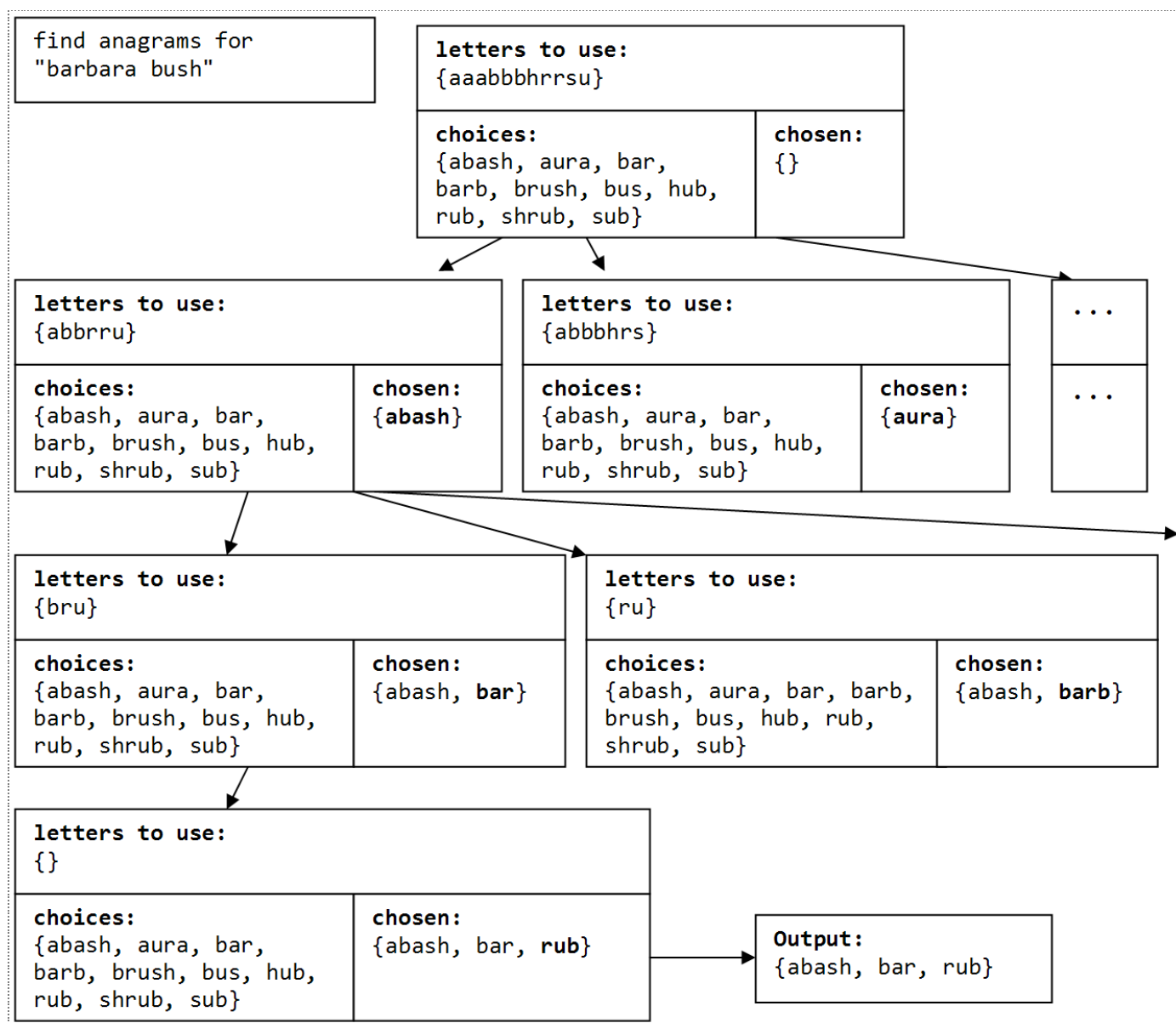

Diagram of anagram search for phrase "barbara bush"

One difficult part of this program limiting to the **max number of words** that can appear in the anagrams. We suggest you ignore the max at first and implement this last, initially printing *all* anagrams regardless of the number of words.

Letter Inventories: Managing Letters in a Phrase

An important aspect of simplifying many backtracking problems is separating recursive code from code to manage low-level details. Some of the low-level details for anagrams involve keeping track of letters and figuring out when one group of letters can be formed from another. A helpful concept you may want to incorporate in your solution is one that we'll call a "**letter inventory**." You are not required to implement your solution using a letter inventory, but we discuss it here as a hint that may help you reach a correct solution.

For this problem, let's define a *letter inventory* as a count of each letter from A-Z found in a given string, ignoring whitespace, capitalization, and non-alphabetic characters. For example, a letter inventory for the string `"Hello, THERE!!"` would keep count of 3 Es, 2 Hs, 2 Ls, 1 O, 1 R, and 1 T. (Note that we strip the spaces and punctuation.) There are many ways that you could represent such a thing; for example, as a string `"eeehhllort"`, or as a map `{'e':3, 'h':2, 'l':2, 'o':1, 'r':1' 't':1}`.

It would be useful to be able to add and subtract phrases from a letter inventory, as well as asking whether an inventory contains a phrase. For example, adding `"hi ho!"` to the previous inventory would produce `"eeehhhhilloort"`. Subtracting `"he he he"` would produce `"hilloort"`. If we asked whether this inventory contains `"root"`, the result is `true`. If we asked whether this inventory contains `"hostel"`, the result is `false`.

Of course, if you implement the concept of a letter inventory in a slow/inefficient way, it can substantially hurt the performance of your program. Be mindful of implementation choices that must loop over collections/strings repeatedly, since your code will be performing these operations a very large number of times. Any code you use to manage letter inventories must be your own work, though you can use collections to help you as needed.

Style Details:

Recursion and backtracking: Part of your grade will come from appropriately utilizing recursive backtracking to implement your word-finding algorithm as described previously. We will also grade on the elegance of your recursive algorithm; don't create special cases in your recursive code if they are not necessary. Avoid **"arm's length" recursion**, which is where the true base case is not found and unnecessary code/logic is stuck into the recursive case. Efficiency of your recursive backtracking algorithms, such as avoiding dead-end searches by pruning, is very important.

Redundancy in recursive code is another major grading focus; avoid repeated logic as much as possible. As mentioned previously, it is fine (sometimes necessary) to use "helper" functions to assist you in implementing the recursive algorithms for any part of the assignment.

Variables: While this constraint is not new to this assignment, we want to stress that you should not make any **global variables** or **static variables** (unless they are constants declared with the `const` keyword). Do not use globals as a way of getting around proper recursion and parameter-passing on this assignment.

Loops/Collections: Loops and collections *are* allowed on this problem. But your fundamental algorithm must be recursive and not based on looping to perform the entire anagram search. You must use recursion to handle the self-similar aspects of the problem.

Commenting: Of course you should have a comment header at the top of your code file and on top of each function. But we want to remind you that you should also have **inline comments** inside functions to explain

complex sections of the code. Don't forget to place descriptive inline comments as needed on any complex code in the bodies to describe nontrivial parts of your algorithms.

Frequently Asked Questions (FAQ):

For each assignment problem, we receive various frequent student questions. The answers to some of those questions can be found by clicking the link below.

Anagrams FAQ (click to show)

Survey: After you turn in the assignment, we would love for you to fill out our anonymous CS 106X homework survey to tell us how much you liked / disliked the assignment, how challenging you found it, how long it took you, etc. This information helps us improve future assignments.

Honor Code Reminder: Please remember to follow the **Honor Code** when working on this assignment. Submit your own work and do not look at others' solutions. Also please do not give out your solution and do not place a solution to this assignment on a public web site or forum. If you need help, please seek out our available resources to help you.