Project 3 - Recursive Anagram Parser

Project Description

Given a newline-delimited text dictionary, write a program that takes an input string and prints out all anagrams of it that exist in the dictionary. *All repetition must be done using recursion: do, while, for, or goto statements are not allowed*.

Difficulties Encountered

Printing permutations

- This easily took me the longest time to do because this involved multi-layer recursion that was hard to intuit. I was hooked in the idea of doing this by using only one function rather than splitting it into an “outer loop” and an “inner loop” like the provided pseudocode said to do. I eventually figured out how to merge the two but it took a lot of troubleshooting and in my final step, decrementing something that I didn’t think should have actually been decremented.

Runtime efficiency

- The first time I got my program to actually work, I had done no sort of optimization. That is, I ran the FULL set of permutations against the entire 201k dictionary. Needless to say, I didn’t even need to see my program run to completion to know that I was doing something wrong.I had thought that the minimum number of iterations would be (factorial(input)) * (dictionary size), but then I realized that I could do so many checks that, while recursive, would cost less than running the permutation code even once.

1. Eliminating options based on word size was an easy efficiency booster.
2. Checking to make sure every letter in the input string appeared at least once in the dictionary word was an easy second-check
3. Checking the ASCII totals of the string and the dictionary word against each other was a final check that, if passed, made the dictionary word a VERY good contender and thus, worth running the permutation code, especially for longer words.
4. A last check that I did to handle duplicates made sure that permutations wouldn’t be run if the dictionary word already appeared in results. This specifically targets duplicates in dictionary. See the next section to see why I had two sections of code to handle duplicates.

Duplicates

- I was stuck on how to do this for a while because I wanted to kill two birds with one stone. The spec said that if the input word was “kloo”, then “look” shouldn’t appear twice. I realized that if I could find a way to break out of this function entirely once a match was found, then I would do two things simultaneously:

1. If the input string was very long, I’d potentially save a LOT of permutations since, if there is a match, there is no inherent reason to check the rest of the permutations.
2. Any further permutations that happened to ALSO equal the dictionary word would cause a duplicate to be read into results, which is undesirable.

- Obviously, this only worked for “internal” duplication, disregarding duplicates in words.txt. I later added the 4th check above to handle input duplication.

G32 fiasco

- Honestly, finding out why my code didn't work was sheer luck of finding someone else having the same problem in office hours. Basically, both our code was working perfectly in Visual Studio (and for me, also XCode as I checked on a friend's Mac), but G32 wasn't reading any anagrams at all. No compilation errors, no warnings, no runtime errors, no memory leaks, no indication of anything wrong with things behind the scenes.
- We happened to think of the bright idea to check the equality of strings read into dict[] and the literals that they *should* be, finding that they were different. After a little more debugging, we then found that when we read in "10th" into dict[], it had a length of 5 rather than 4.
- It was then that we realized that in using getline(), G32 was reading the '\n' into the actual word and NOT ignoring it when checking equality, unlike BOTH VS and XC, which happen to IGNORE the newline character when they check equality. After adapting my code to using the insertion operator -- >> -- everything worked as intended.

Testing Methodology

This project isn't easily testable using assert statements, so I'll list out test scenarios that fall into two categories: the provided dictionary and a custom one
Provided 201k dictionary

- The given test cases: rat, regardless, and kloo. These should have exactly the behavior as mentioned in the spec:
    - Rat is an example of an ideal case.
    - Regardless is an example of a no-results case and of a long input, which shouldn't take that long to run, with good optimization.
    - Kloo is the internal duplicates case, where permutations that separately match a dictionary word shouldn't read twice into results.
- Change MAXRESULTS
    - Something less than 3, then run "rat"... only 2 results should be displayed
- Change MAXDICTWORDS
    - Have it set to the default 30000 to check that your makeDictionary function actually stops when it reaches the end of the file.
    - Something less than 25144, or whatever the number of words is, and then input a word that is excluded. The results should not contain the excluded word because it shouldn't be read in.
    - Exactly equal to the number of words, just to make sure this works.

Custom Dictionary: on top of providing a smaller data set to work with that makes it easier to debug when the code hasn't been optimized, there are a few things that you should specifically try out:

- Duplicates in words.txt
    - This checks the external scenario of duplicates; even if a word appears twice in the dictionary, it shouldn't appear more than once in results
- Empty words.txt
    - This should run without problem. If it doesn't, there is likely something wrong with makeDictionary
- Playing around with MAXRESULTS and MAXDICTWORDS more
    - Since you can make up words, you should have at least one test case where there are X words in words.txt, and MAXRESULTS and MAXDICTWORDS are set to X. This just makes sure this boundary case is accounted-for.

Other

- Inputting a long string from words.txt backwards is basically the worst-case scenario for the most rudimentary algorithm, since it would have to loop all the way to the end before finding out (finally) that it matches up. As prof.Ambrosio said, though, as long as it doesn't crash the stack, long run times are fine.
    - After having completed this project, I have realized that based on the optimization checks that I perform, there is a much easier solution than recursing through all the permutations of a word. Using find() can actually produce a very simple, very quick solution.
        - Recursively go through each of dictionary string's characters, and every time you are able to .find() that character in the input string, remove the character from the input string. The base case is when the input string is empty: check to see if you've "used up" all the dictionary string's characters. If so, then it's an anagram. If not, this indicates that the word isn't an anagram.
- UPDATE: I read through the spec and made the full use of the string library, and because of this I ended up using the algorithm in the previous bullet, which works insanely fast for any length of word.

Description of Files

anagrams.cpp

- My optimized anagram parser.