
Science Student Success Centre
Carleton ‹Coding› Challenge

4 - 9PM, January 31st, 2018

<Purpose> The final goal for this coding challenge is to create an Android app for a simple anagram game.

- The game provides the user with a word from the dictionary.
- The user tries to create as many words as possible that contain all the letters of the given word plus one additional letter. Note that adding the extra letter at the beginning or the end without reordering the other letters is not valid. For example, if the game picks the word 'ore' as a starter, the user might guess 'rose' or 'zero' but not 'sore'.
- The user can give up and see the words that they did not guess.

<How It Works> The problem will be described in an iterative approach so you will need to complete one milestone before moving on to the next.

<Starter Code> We have provided some starter code that contains a 10 000 word dictionary and handles the UI portions of the game. You are responsible for the actual implementation! In order to ensure that the game is not too difficult, the computer will only propose words that have at least 5 possible valid anagrams.

Starter Code Explanation:

- AnagramsActivity: The starter code implements several methods:
    - onCreate: this method gets called by the system when the app is launched. It is made up of some boilerplate code plus code that opens the word list to initialize the dictionary and code to connect the text box to the processWord helper.
    - processWord: A helper that adds words to the UI and colors them.
    - onCreateOptionsMenu: boilerplate
    - onOptionsItemSelected: boilerplate
    - defaultAction: This is the handler that is called when the floating button is clicked. Depending on the game mode, it either starts the game or shows the missing answer to the previous game.
- AnagramDictionary: This class will store the valid words from the text file and handle selecting and checking words for the game. This is where your code will among the following methods:
    - AnagramDictionary: The constructor. It should store the words in the appropriate data structures
    - isGoodWord: Asserts that the given word is in the dictionary and isn't formed by adding a letter to the start or end of the base word.
    - getAnagrams: Creates a list of all possible anagrams of a given word.
    - getAnagramsWithOneMoreLetter: Creates a list of all possible words that can be formed by adding one letter to the given word.
    - pickGoodStarterWord: Randomly selects a word with at least the desired number of anagrams.

THE CHALLENGE:

MILESTONE 1

This first section involves having a simplified version of the game where the user guesses anagrams of the given word.

TASKS:

1. Advance the implementation of the AnagramDictionary's constructor. Each word that is read from the dictionary file should be stored in an ArrayList (called wordList). We will store duplicates of our words in some other convenient data structures later but wordList will do for now.
2. Implement getAnagrams which takes a string and finds all the anagrams of that string in our input (Hint: Create a helper function that takes a sorts the characters of a string into alphabetical order, e.g. "post" -> "opst").
3. You will need to create two new data structures in the constructor to make the game more efficient. The constructor should have a HashSet that will allow you to rapidly verify whether a user inputted word is a valid anagram. You should also group anagrams (via a HashMap) together for easy searching (e.g key: "opst" value: ["post", "spot", "pots", "tops", ...]).

   As you process the input words, check whether your HashMap already contains an entry for that key. If it does, add the current word to the ArrayList at that key. Otherwise, create a new ArrayList, add the word to it and store it in the HashMap with the corresponding key.

MILESTONE 2

Milestone 2 is all about ensuring that the words picked are suitable for the anagram game.

TASKS:

1. Your task is to implement isGoodWord which checks: the provided word is a valid dictionary word, and the word does not contain the base word as a substring.

Example: With base word “post”:

Input	**Output**
isGoodWord("nonstop")	true
isGoodWord("poster")	false
isGoodWord("lamp post")	false
isGoodWord("spots")	true
isGoodWord("apostrophe")	false

2. Next, implement getAnagramsWithOneMoreLetter which takes a string and finds all anagrams that can be formed by adding one letter to that word. (You’ll need to update defaultAction in AnagramsActivity to invoke getAnagramsWithOneMoreLetter instead of getAnagrams).

3. Now, it’s time to implement pickGoodStarterWord to make the game more interesting. Pick a random starting point in the wordList array and check each word in the array until you find one that has at least MIN_NUM_ANAGRAMS anagrams.

MILESTONE 3

1. At this point, the game is functional but can be quite hard to play if you start off with a long base word. To avoid this, let's refactor AnagramDictionary to give words of increasing length. Your first word should be 4 letters, then 5, then 6, etc., until you reach the maximum word length. Then reset to 4.
2. Now you have a complete game, so it's time to extend. Be creative and add some cool functionality! If you can't think of anything, you can try: Two-letter mode - switch to allowing the user to add two letters to form anagrams.