

CSC148H Assignment 2

February 27, 2015

Deadlines

- ▶ Due March 5 at 22:00
- ▶ Submit to MarkUs

Overview

- ▶ Part 1 - Anagram Generator (80%)
- ▶ Part 2 - Binary Tree Representations (20%)

Submission

Part 1 - Anagram Generator (80%)

- ▶ `anagram.py`
- ▶ `anagram_design.txt`
- ▶ `test_anagram.py`

Part 2 - Binary Tree Representations (20%)

- ▶ `112nr.py`

Part 1 - Anagram Generator

This assignment question will give you practice with recursive backtracking. You are to create a class called `AnagramSolver` that uses a word list to find all combinations of words that have the same letters as a given phrase.

Tip:

- ▶ Use the class name `AnagramSolver` in a file called `anagram.py`

Part 1 - Methods - Constructor

- ▶ `__init__`: Takes a list of words, and constructs an anagram solver that will use the given list as its dictionary. It is not to modify the list.
- ▶ You should use the sample dictionary provided (see `dict.txt`) when writing your code so that your results will match the sample execution.

Tips:

- ▶ Create your constructor method and call it `__init__`
- ▶ Do not create a method called `constructor`
- ▶ Use the dictionary provided (`dict.txt`)
- ▶ Read through the dictionary words in `dict.txt`

Part 1 - Methods - generateAnagrams

- ▶ `generateAnagrams`: Takes a string `s` and an integer `max`, and returns all combinations of words from its dictionary that are anagrams of the String `s` and that include at most `max` words (or unlimited number of words if `max` is 0).
- ▶ Throws an appropriate built-in exception if `max` is less than 0.

Tips:

- ▶ Pay attention to your method name
- ▶ Adhere to the type contract

Part 1 - Methods - generateAnagrams

- ▶ This method must produce the anagrams in the same format as in the sample log.
- ▶ It is to return a list of lists, where each list is an anagram. The easiest way to do this is to build up your answer in a list. Then you can simply append that list to your master list of anagrams.

Part 1 - Recursive Backtracking

- ▶ You are required to solve this problem by using recursive backtracking.
- ▶ You are to write a recursive method that builds up an answer one word at a time.
- ▶ On each recursive call, you are to search the dictionary from beginning to end and to explore each word that is a match for the current set of letters.
- ▶ The possible solutions are to be explored in dictionary (i.e. alphabetical) order. For example, in deciding what word might come first, you are to examine the words in the same order in which they alphabetically appear in the dictionary.

Part 1 - Letter Manager

- ▶ The low-level details for the anagram problem involve keeping track of various letters and figuring out when one group of letters can be formed from another group of letters.
- ▶ It turns out that the `LetterManager` class that we wrote for assignment 1 provides us with the low-level support we need.

Part 1 - Letter Manager

- ▶ For any given word or phrase, what matters to us is how many of each letter there are. Recall that this is exactly what the LetterManager keeps track of.
- ▶ The **subtract** method of the LetterManager is the key to solving this problem.
- ▶ A Python LetterManager (see letterManager.py) is provided for you to use in this assignment. Do not make any changes to this letterManager.py

Part 1 - Optimization

- ▶ Part of your grade will be based on the efficiency of your solution.
- ▶ Recursive backtracking is, in general, highly inefficient because it is a brute force technique that checks every possibility, but there are still things you can do to make sure that your solution is as efficient as it can be.
- ▶ Be careful not to compute something twice if you don't need to. And don't continue to explore branches that you know will never be printed.
- ▶ You are also required to implement the following two optimizations:

Part 1 - First Optimization

- ▶ There is no reason to convert dictionary words into inventories more than once. You should preprocess the dictionary in your constructor to compute all of the inventories in advance (once per word). You'll want fast access to these inventories as you explore the possible combinations. A Python map will give you fast access.

Part 1 - Second Optimization

- ▶ For any given phrase, you can reduce the dictionary to a smaller dictionary of relevant words. A word is relevant if it can be subtracted from the given phrase. Only a fraction of the dictionary will, in general, be relevant to any given phrase. So reducing the dictionary before you begin the recursion will allow you to speed up the searches that happen on each recursive invocation. To implement this, you should construct a short dictionary for each phrase you are asked to explore that includes just the words relevant to that phrase. An important efficiency decision you have to make is whether to do this once before the recursion begins, or also on a certain number of deeper levels of recursion. You should experiment with different word lengths, make your decision, and document your reasoning.

Part 1 - Output

Your `generateAnagrams` method is to produce exactly the same output in exactly the same order as in the sample below. The input `s` is office key, `max` is 0, and the output is as follows.

```
[['eke', 'icy', 'off'],  
 ['eke', 'off', 'icy'],  
 ['ice', 'key', 'off'],  
 ['ice', 'off', 'key'],  
 ['icy', 'eke', 'off'],  
 ['icy', 'off', 'eke'],  
 ['key', 'ice', 'off'],  
 ['key', 'off', 'ice'],  
 ['key', 'office'],  
 ['off', 'eke', 'icy'],  
 ['off', 'ice', 'key'],  
 ['off', 'icy', 'eke'],  
 ['off', 'key', 'ice'],  
 ['office', 'key']]
```

Part 1 - Anagram Template

- ▶ Download the starter code: `anagram-template.py`.
- ▶ Implement the constructor and the `generateAnagrams` method in this file.
- ▶ Do not modify the header definitions for the `generateAnagrams` method or the constructor.
- ▶ You are free to add any helper functions or any other code that you feel that you need.
- ▶ **DO NOT CHANGE THE NAME OF THE CLASS OR THE SIGNATURE OF THE `generateAnagrams` method.**

Part 1 - Submission

- ▶ Submit your `AnagramSolver` class in a file called `anagram.py`.
- ▶ You should also submit a **text document** that outlines your design decisions called `anagram_design.txt`, and a Python file containing your unit tests called `test_anagram.py`.
- ▶ Inside your unit tests as part of the doc strings, you should document the following: why you felt each test was necessary, and why you feel your test cases are sufficient to conclude that your implementation is correct.

Part 2 - Binary Tree Representations

- ▶ Create a function `l12nr` that takes list of lists representation of a binary tree and converts it to an equivalent binary tree in nodes and references form.
- ▶ Your function should return a binary tree object of the class in `nodesrefs.py` (do not make any changes to this file).
- ▶ Start with `l12nr-template.py` and add your code for the function. Do not modify the function header. You are free to add any helper functions or any other code that you feel that you need.

Submit just **one** Python file containing your `l12nr` function called `l12nr.py`.

Mark Breakdown

Part 1 - Anagram Generator (80%)

- ▶ Correctness (75%)
- ▶ Commenting and Style/Design (15%)
- ▶ Testing (10%)

Part 2 - Binary Tree Representations (20%)

- ▶ Correctness (100%)

Attribution

FOR EACH FILE THAT YOU SUBMIT (.TXT OR .PY), YOU MUST INCLUDE THE FOLLOWING HONOUR CODE AT THE VERY TOP OF THE FILE. FAILURE TO DO THIS, WILL RESULT IN AN AUTOMATIC ZERO.

```
*****  
# Assignment2:  
# UTOR user_name:  
# First Name:  
# Last Name:  
# Student # #  
#  
#  
# Honour Code: I pledge that this program represents my own  
# program code and that I have coded on my own. I received  
# help from no one in designing and debugging my program.  
# I have also read the plagiarism section in the course info sheet  
# of CSC 148 and understand the consequences.  
*****
```