Introduction

This lab is to practice the use of classes for manipulating graphs.

Our data in this problem will consist of strings, specifically the ones contained in the `words.txt` file, a copy of which can be found in the COSC201 code repository.

Let's define two words to be neighbours if one can be obtained from another in one of the following ways:

- By adding or deleting a letter anywhere (e.g., from `fog` to `frog` and vice versa), or
- By replacing any single letter with another one (e.g., from `fog` to `fig`, or from `frog` to `flog`).

Our words will always consist only of lower case characters `a-z`.

Rather than using the `Graph` class that we used in lectures, it's more convenient to think of the words themselves as the vertices of a graph (and each have a set of neighbours).

Problems

1. Using the `Graph` class as a source, make a class `SGraph` (string similarity graph) in which `neighbours` is a `HashMap` from `String` to `HashSet<String>`. Some of the methods will also need to be adjusted.
2. Write a basic method:
   ```
   boolean edgePair(String a, String b)
   ```
   that returns `true` if the two strings are neighbours according to the rules above, and `false` if they are not.
3. When we build an `SGraph` from a word list (such as `words.txt`) according to the adjacency rules above there are two obvious strategies to generate the edges:
   - For each pair of words from the dictionary, use `edgePair` to determine whether they are adjacent and, if so, add the appropriate edge.
   - For each word in the dictionary, compute all of its possible neighbours (e.g., `fpog` is a possible neighbour of `frog`) and, whenever they are in the dictionary, add the appropriate edge.

   For large dictionaries, which method is likely to be more efficient? Ideally, implement both versions and check your prediction.

4. Build the graph based on `words.txt` (of course, build similar graphs based on much smaller files first to test for effectiveness). Using it, answer the following questions:
   - Which word (or words) are most popular in the sense of having the most neighbours? How many neighbours do they have?
   - How many words are *isolated points* i.e., have no neighbours?
   - What is the shortest path from `me` to `you`? Or perhaps there isn't one?

Extension and reflection

- There are many places in this problem where we can think about efficiency. For instance, in constructing the neighbours of a word we could consider only possibilities that are shorter (obtained by deleting a letter) or later in alphabetical order (obtained by changing a letter to a later one), provided we always add to both neighbour lists (so the adjacency between `frog` and `fog` would be added when considering `frog`, while that between `frog` and `flog` would be added when considering `flog`). How much does this speed things up? What other options are there for improving efficiency?
- If all I want to do is to get from `me` to `you` it seems wasteful to construct the entire graph. I strongly suspect that the word `avoid` will not occur in a shortest path. How could one take care to avoid considering words like `avoid` in solving that problem (hint, think of the graph as a *dynamic* rather than a static structure – add edges only when they become relevant).
- How would we work out how many different connected groups of words there are? For instance, `frog`, `flog`, `fog`, and `fig` certainly all belong to one group.