I&C SCI 46 Fall 2021 Project 3: Lewis Carroll Distance

Due Tuesday, **November 2, 9:59 AM.**

Introduction

In 1879, Lewis Carroll proposed the following puzzle to the readers of *Vanity Fair*: transform one English word into another by going through a series of intermediate English words, where each word in the sequence differs from the next by only one substitution. To transform *head* to *tail*, one can use four intermediates: head → heal → teal → tell → tall → tail. We refer to the smallest number of substitutions necessary to transform one word to another as the *Lewis Carroll* distance between the two words.

Getting Started

Before you begin work on this project, there are a couple of chores you'll need to complete on your VM to get it set up to proceed.

Refreshing your VM environment

Even if you previously downloaded your VM, you will probably need to refresh its environment before proceeding with this project. Log into your VM and issue the command ics46 version to see what version of the environment you currently have stored on your VM. Note, in particular, the timestamp; if you see a version that does not make reference to project 1, you will want to refresh your environment by running the command ics46 refresh to download the latest one before you proceed with this project.

If you're unable to get outgoing network access to work on the VM — something that afflicts a handful of students each quarter — then the refresh command won't work, but an alternative approach is to download the latest environment from the link below, then to upload the file on to your VM using SCP. (See the Project #0 write-up for more details on using SCP.) Once the file is on your VM, you can run the command **ics46 refresh_local NAME_OF_ENVIRONMENT_FILE**, replacing **NAME_OF_ENVIRONMENT_FILE** with the name of the file you uploaded; note that you'd need to be in the same directory where the file is when you run the command.

Creating your project directory on your VM

A project template has been created specifically for this project, containing a similar structure to the basic template you saw in previous projects

Decide on a name for your project directory, then issue the command **ics46 start YOUR_CHOSEN_PROJECT_NAME project3** to create your new project directory using the project1 template. (For example, if you wanted to call your project directory birds, you would issue the command ics46 `start birds project3` to create it.) Now you're ready to proceed!

Requirements

- Fill in WordSet.cpp
    - You will need to fill in the function `polynomialHashFunction`, which takes three parameters.
        - This function will interpret the string parameter, which will contain only lower-case letters, as coefficients for a polynomial of degree equal to the length of the string. You will evaluate it at the point "base," which can be thought of (if you prefer) as interpreting the string as a base-that integer. Treat 'a' as 1, 'b' as 2, and so on. It returns the smallest positive integer that is a representation of that string, in that base, mod the given modulus.
            - For example, `polynomialHashFunction`("abz", 5, 10) should return the result of $(1 * 5^2 + 2 * 5 + 26) \% 10$.
        - Be careful about when you take the modulus within the hash function.
        - Be careful about when you take the modulus outside the hash function.
        - Do not assume any parameters match the named constants in the file -- we will use those later.
        - The provided code DOES NOT accurately compute this function, but it does show you how to easily access numeric values for the letters.
    - Your implementation **must be** done via a Cuckoo hash table, as described in lecture. Note that we *are not* creating a Cuckoo filter.
    - The first table's hash function is to be done with base=BASE_H1, with a similar rule for the second hash function for BASE_H2.
    - You must use a dynamically-allocated C-style array, not a `std::vector` or similar container, as the basis in your WordSet.
        - You should start your array at size 11.
        - Resize only when the element about to be inserted cannot be added to the existing structure. An element cannot be inserted when *evictionThreshold* evictions have been made during its attempted insertion. That value is an optional second parameter to the constructor.
        - When you resize and rehash, the next table size should be the *smallest* prime number that is no smaller than twice the current table size. For example, if your current table size is 11, your next one is 23. If your current table size were somehow 13, your next one would be 29.
    - Your implementation must fit the interface given.
    - Your implementation **does not need to be templated** -- nor should it be, for the purposes of this assignment.
        - In fact, doing so will cause an issue for some of our provided tests.

    - You **do** need to implement the destructor. Memory leaks will cause a grade penalty.
    - Do not hard code your table for the uses we'll have in the next section.

- Write function std::vector<`std::string> convert(const std::string & s1, const std::string & s2, const WordSet & words)` in `convert.cpp`
    - This function will return the conversion between s1 and s2, according to the lowest *Lewis Carroll* distance. The first element of the vector should be s1, the last s2, and each consecutive should be one letter apart. Each element should be a valid word. If there are two or more equally least Lewis Carroll distance ways to convert between the two words, you may return any of them.

    - If there is no path between s1 and s2, return an empty vector.

    - It is recommended that you compute the distance via a breadth-first search. To visualize this, imagine a graph where the words are vertices and two vertices share an (undirected) edge if they are one letter apart.
        - If you do not know what this means, please ask -- Shindler would be happy to explain.
        - You *may* use std::queue -- you do not need to use the one you wrote for project 2.

    - A good thing to do the first time you see a word in the previous part is to place it into a std::map<string, string>, where the key is the word you just saw and the value is the word *that you saw immediately before it*. This will allow you to later produce the path: you know the last word, and you know the prior word for each word in the path except the first. Furthermore, if the key isn't in that map, this tells you that you haven't seen it before.

    - Your implementation does not have to be the most efficient thing ever, but it cannot be "too slow." In general, any test case that takes over a minute on the grader's computer may be deemed a wrong answer, even if it will later return a correct one.

You **may not** use any classes that are part of the C++ standard template library in implementing your Wordset. You *may* use simple functions, such as std::swap or computing a logarithm if you so choose.
You may use std::queue, std::map, std::stack, and/or std::set in convert.cpp, but you *may not* use them in place of your WordSet: you must use that in the appropriate places. *A solution for convert.cpp that does not use WordSet to determine if something is a word, and for the graph-like operations described, will receive no credit*.