Daily Coding Problem #226

Problem

This problem was asked by Airbnb.

You come across a dictionary of sorted words in a language you've never seen before. Write a program that returns the correct order of letters in this language.

For example, given `['xww', 'wxyz', 'wxyw', 'ywx', 'ywz']`, you should return `['x', 'z', 'w', 'y']`.

Solution

It may be hard to know where to start with a problem like this, so let's look at the example for some guidance.

Note that from the last two words, ywx and ywz, we can tell that x must come before z. This is because the first two letters of each word match, and therefore the tiebreaker must be the third letter.

In fact, if we make pairwise comparisons of all the adjacent words in our dictionary, using this same process, we can discover the rest of the rules that govern the ordering of letters. We will store these rules in a graph, where each letter is a key, whose values (if they exist) are letters that must come after it.

Since we compare each letter at most twice, the time complexity of this part is `O(N)`, where `N` is the number of characters in our input.

```python
def create_graph(words):
    letters = set(''.join(words))
    graph = {letter: [] for letter in letters}

    for pair in zip(words, words[1:]):
        for before, after in zip(*pair):
            if before != after:
                graph[before].append(after)
                break

    return graph
```

For the example above, our graph would be `{'w': ['y'], 'x': ['w', 'z'], 'y': [], 'z': ['w']}`.

We now want to find a way of combining these rules together. Luckily, there exists a procedure for carrying out just such an operation, called topological sort. The idea is that we can carry out a depth first search through the graph, adding a letter to our result only once we've finished visiting the letters that come after it. Because we insert new letters at the front of the list, we guarantee that the result will be ordered correctly.

For example, in the dictionary above, we would only add `w` once we've visited `y`, so there is no way that `y` could come before `w` in the result.

```python
from collections import deque

def visit(letter, graph, visited, order):
    visited.add(letter)
    for next_letter in graph[letter]:
        if next_letter not in visited:
            visit(next_letter, graph, visited, order)

    order.appendleft(letter)

def toposort(graph):
    visited = set()
    order = deque([])

    for letter in graph:
        if letter not in visited:
            visit(letter, graph, visited, order)

    return list(order)
```

Topological sort is `O(V + E)`, where `V` is the number of letters in the alphabet and `E` is the number of rules we found earlier. Since neither of these is greater than `O(N)` (where `N` is the number of characters in our input), the overall complexity is still `O(N)`.

The glue holding this code together is given below.

```
def alien_letter_order(words):
    graph = create_graph(words)
    return toposort(graph)
```