Daily Coding Problem #22

Problem

This problem was asked by Microsoft.

Given a dictionary of words and a string made up of those words (no spaces), return the original sentence in a list. If there is more than one possible reconstruction, return any of them. If there is no possible reconstruction, then return null.

For example, given the set of words 'quick', 'brown', 'the', 'fox', and the string "thequickbrownfox", you should return ['the', 'quick', 'brown', 'fox'].

Given the set of words 'bed', 'bath', 'bedbath', 'and', 'beyond', and the string "bedbathandbeyond", return either ['bed', 'bath', 'and', 'beyond] or ['bedbath', 'and', 'beyond'].

Solution

We might be initially tempted to take a greedy approach to this problem, by for example, iterating over the string and checking if our current string matches so far. However, you should immediately find that that can't work: consider the dictionary {'the', 'theremin'} and the string 'theremin': we would find 'the' first, and then we wouldn't be able to match 'remin'.

So this greedy approach doesn't work, since we would need to go back if we get stuck. This gives us a clue that we might want to use backtracking to help us solve this problem. We also have the following idea for a recurrence: If we split up the string into a prefix and suffix, then we can return the prefix extended with a list of the rest of the sentence, but only if they're both valid. So what we can do is the following:

- Iterate over the string and split it into a prefix and suffix
- If the prefix is valid (appears in the dictionary), then recursively call on the suffix
- If that's valid, then return. Otherwise, continue searching.
- If we've gone over the entire sentence and haven't found anything, then return empty.

We'll need a helper function to tell us whether the string can actually be broken up into a sentence as well, so let's define `find_sentence_helper` that also returns whether or not the sentence is valid.

```
def find_sentence(dictionary, s):
    sentence, valid = find_sentence_helper(dictionary, s)
    if valid:
        return sentence

def find_sentence_helper(dictionary, s):
    if len(s) == 0:
        return [], True

    result = []
    for i in range(len(s) + 1):
```

```
        prefix, suffix = s[:i], s[i:]
        if prefix in dictionary:
            rest, valid = find_sentence_helper(dictionary, suffix)
            if valid:
                return [prefix] + rest, True
    return [], False
```

This will run in O(2^N) time, however. This is because in the worst case, say, for example, s = "aaaaab" and dictionary = ["a", "aa", "aaa", "aaaa", "aaaaa"], we will end up exploring every single path, or every combination of letters, and the total number of combinations of characters is 2^N.

We can improve the running time by using dynamic programming to store repeated subcomputations. This reduces the running time to just O(N^2). We'll keep a dictionary that maps from indices to the last word that can be made up to that index. We'll call these starts. Then, we just need to do two nested for loops, one that iterates over the whole string and tries to find a start at that index, and a loop that checks each start to see if a new word can be made from that start to the current index.

Now we can simply take the start at the last index and build our sentence backwards:

```
def find_sentence(s, dictionary):
    starts = {0: ''}
    for i in range(len(s) + 1):
        new_starts = starts.copy()
        for start_index, _ in starts.items():
            word = s[start_index:i]
            if word in dictionary:
                new_starts[i] = word
        starts = new_starts.copy()

    result = []
    current_length = len(s)
```

```
    if current_length not in starts:
        return None
    while current_length > 0:
        word = starts[current_length]
        current_length -= len(word)
        result.append(word)

    return list(reversed(result))
```

Now this runs in O(N^2) time and O(N) space.
