Daily Coding Problem #34

Problem

This problem was asked by Quora.

Given a string, find the palindrome that can be made by inserting the fewest number of characters as possible anywhere in the word. If there is more than one palindrome of minimum length that can be made, return the lexicographically earliest one (the first one alphabetically).

For example, given the string "race", you should return "ecarace", since we can add three letters to it (which is the smallest amount to make a palindrome). There are seven other palindromes that can be made from "race" by adding three letters, but "ecarace" comes first alphabetically.

As another example, given the string "google", you should return "elgoogle".

Solution

Notice that whenever we add a character, it should ideally match the one on the other side of the string. We can use the following recurrence to solve this problem:

- If `s` is already a palindrome, then just return `s` -- it's already the shortest palindrome we can make
- If the first character of `s` (let's call it a) is the same as the last, then return `a + make_palindrome(s[1:-1]) + a`
- If the first character of `s` is different from the last (let's call this b), then return the minimum between:
    - `a + make_palindrome(s[1:]) + a`
    - `b + make_palindrome(s[:-1]) + b` or the lexicographically earliest one if their lengths are equal.

So a naive recursive solution might look like this:

```
def is_palindrome(s):
    return s == s[::-1]

def make_palindrome(s):
    if is_palindrome(s):
        return s
    if s[0] == s[-1]:
        return s[0] + make_palindrome(s[1:-1]) + s[-1]
    else:
        one = s[0] + make_palindrome(s[1:]) + s[0]
        two = s[-1] + make_palindrome(s[:-1]) + s[-1]
        if len(one) < len(two):
            return one
        elif len(one) > len(two):
            return two
        else:
            return min(one, two)
```

Recall that the min of two strings in python will return the lexicographically earliest one!

However, this algorithm runs in O(2^N) time, since we could potentially make two recursive calls each time. We can speed up using dynamic programming, as usual. We can either memoize our results so that we don't duplicate any work, or use a table and do bottom-up programming.

Let's start with memoization. We can keep a cache and store all our results when we compute them in the cache. If we come across a string we've seen before, then we just need to look it up in the cache.

```
cache = {}

def is_palindrome(s):
    return s == s[::-1]

def make_palindrome(s):
    if s in cache:
        return cache[s]

    if is_palindrome(s):
        cache[s] = s
        return s
    if s[0] == s[-1]:
        result = s[0] + make_palindrome(s[1:-1]) + s[-1]
        cache[s] = result
        return result
    else:
```

```
        one = s[0] + make_palindrome(s[1:]) + s[0]
        two = s[-1] + make_palindrome(s[:-1]) + s[-1]
        cache[s] = min(one, two)
        return min(one, two)
```

However, this is inefficient due to buildup in the call stack. We can build a 2D table instead. We'll store, in each index, the shortest palindrome that can be made in the substring defined from `i` to `i + j`. Then instead of calling ourselves recursively, we'll just look up the values in our table:

```
def make_palindrome(s):
    if len(s) <= 1:
        return s
    table = [['' for i in range(len(s) + 1)] for j in range(len(s) + 1)]

    for i in range(len(s)):
        table[i][1] = s[i]

    for j in range(2, len(s) + 1):
        for i in range(len(s) - j + 1):

            term = s[i:i + j]
            first, last = term[0], term[-1]
            if first == last:
                table[i][j] = first + table[i + 1][j - 2] + last
            else:
                one = first + table[i + 1][j - 1] + first
                two = last + table[i][j - 1] + last
                if len(one) < len(two):
                    table[i][j] = one
                elif len(one) > len(two):
                    table[i][j] = two
                else:
                    table[i][j] = min(one, two)

    return table[0][-1]
```

Because we store a part of our input string in each index of our matrix, the time and space complexity for this solution is O(N^3).