Daily Coding Problem #181

Problem

This problem was asked by Google.

Given a string, split it into as few strings as possible such that each string is a palindrome.

For example, given the input string `racecarannakayak`, return `["racecar", "anna", "kayak"]`.

Given the input string `abc`, return `["a", "b", "c"]`.

Solution

The naive, brute force way to solve this would be to recursively look at each possible split of `s` and get their palindrome splits. We'd keep track of the minimum cut at each split. The base case is when the string itself is a palindrome, leading to a min cut of 1.

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

def split_into_palindromes(s):
    if not s:
        return []
    if is_palindrome(s):
        return [s]

    min_cut = None
    for i in range(1, len(s)):
        curr_cut = split_into_palindromes(s[:i]) + split_into_palindromes(s[i:])
        if min_cut is None or len(curr_cut) < len(min_cut):
            min_cut = curr_cut
    return min_cut
```

However, this takes $O(2^n)$ time, since at each step we're recursively calling ourselves twice, which is far too slow. Let's try to speed it up.

Since every function call relies on getting substrings that are palindromes of the original string `s`, we can build up and use a cache to keep track of all possible subpalindromes so that lookups are faster. So, we'll build a n by n matrix `A` from the bottom up such that every value `A[i][j]` is `true` if `s[i:j]` is a palindrome and `false` if it isn't.

We can then use dynamic programming and keep an array `P` of length `n` with the following invariant:

- `P[i]` contains the smallest number of palindromes made from the string `s[:i]`.

To accomplish this, we'll define the following recurrence:

- `P[i] = minimum length of P[j] + [s[j:i]] if s[j:i] is a palindrome`

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

def split_into_palindromes(s):
    A = [[None for _ in range(len(s))] for _ in range(len(s))]

    # Set all substrings of length 1 to be true
    for i in range(len(s)):
        A[i][i] = True

    # Try all substrings of length 2
    for i in range(len(s) - 1):
        A[i][i + 1] = s[i] == s[i + 1]

    i, k = 0, 3
    while k <= len(s):
        while i < (len(s) - k + 1):
            j = i + k - 1
            A[i][j] = A[i + 1][j - 1] and s[i] == s[j]
            i += 1
        k += 1
        i = 0

    P = [None for _ in range(len(s) + 1)]
    P[0] = []
    for i in range(len(P)):
        for j in range(i):
            matrix_i = i - 1

            if A[j][matrix_i]:
                if P[i] is None or (len(P[j]) + 1 < len(P[i])):
                    P[i] = P[j] + [s[j:i]]

    return P[-1]
```

This takes $O(n^3)$ time, since we have to build up the matrix (which is $O(n^2)$ time), and also create a new list with size up to O(n) for each inner loop of the cache. This also takes $O(n^2)$ space.