Daily Coding Problem #121

Problem

This problem was asked by Google.

Given a string which we can delete at most k, return whether you can make a palindrome.

For example, given 'waterrfetawx' and a k of 2, you could delete f and x to get 'waterretaw'.

Solution

Notice the recursive structure of this problem:

- If the string is already a palindrome, then we can just return true.
- If the string is not already a palindrome, then try getting rid of the first or last character that does not contribute to its palindromicity.

For example, say we're looking at the example string `waterrfetawx`. Since the string is not already a palindrome, we'll try removing the first character or the last character. We'll take the happy path and remove the last character, leaving us with `waterrfetaw`. We can remove `wate` off both ends, leaving us with `rrf`. Then we try removing the first and

last character again, and we find that we can remove two characters (x and f) to get a palindrome.

This leads itself to the following code:

```
def k_palindrome(s, k):
    # If s is already a palindrome, return true
    if len(s) <= 1:
        return True

    # Get rid of matching ends
    while s[0] == s[-1]:
        s = s[1:-1]
        if len(s) <= 1:
            return True

    if k == 0:
        return False

    # Try getting rid of the first and last character to see if we
    # can make a palindrome by removing k - 1 chars.
    return k_palindrome(s[:-1], k - 1) or k_palindrome(s[1:], k - 1)
```

This takes $O(2^{\min(n, k)})$ time, where n is the length of the original string. This is because we call `k_palindrome` twice on each subproblem, and on each call, either k or the string gets reduced by at least 1.

We can do this faster, though. If we can find the longest palindromic subsequence of a string, then we can reduce this problem to finding it by checking the difference in string lengths. If it's greater than k, we return false, otherwise true.

```
def k_palindrome(s, k):
    return len(s) - longest_palindromic_subsequence(s) < k
```

How do we find the longest palindromic subsequence, though? We can use dynamic programming to do this in $O(n^2)$ time:

```
def longest_palindromic_subsequence(s):
    if s == s[::-1]:
        return len(s)
    n = len(s)
    A = [[0 for j in range(n)] for i in range(n)]

    for i in range(n - 1, -1, -1):
        A[i][i] = 1
        for j in range(i + 1, n):
            if s[i] == s[j]:
                A[i][j] = 2 + A[i + 1][j - 1]
            else:
                A[i][j] = max(A[i + 1][j], A[i][j - 1])

    return A[0][n - 1]
```

We define an N by N table A, and `A[i][j]` will represent the length of the longest palindromic substring starting at `i` and ending at `j`. The relationship is as follows:

- If `i == j`: 1
- If `s[i] == s[j]`: take the longest palindromic subsequence from `s[i + 1] to j[i - 1]` and add two (since we have two more characters at the ends)
- Else (`s[i] != s[j]`): take the maximum of the longest palindromic subsequences of ranges `i + 1` to `j`, or `i` to `j - 1`.

Thus we are building the palindrome from the inside out by trying to match all characters at the ends, ignoring characters that don't help. This will take O(N^2) time and space.