Daily Coding Problem #209

Problem

This problem was asked by YouTube.

Write a program that computes the length of the longest common subsequence of three given strings. For example, given "epidemiologist", "refrigeration", and "supercalifragilisticexpialodocious", it should return 5, since the longest common subsequence is "eieio".

Solution

Let's first look at a simpler problem, that of finding the longest common subsequence of two strings. Our first instinct might be to solve this recursively. Note that if the starting characters of both strings match, we can safely include that letter in our solution. Otherwise, it is only possible for at most one of those characters to be part of our solution. Once we know how to deal with these first characters, we can remove these characters and apply the same logic to two shorter strings.

In code, our solution might look like this:

```
def lcs(a, b):
    if not a or not b:
        return 0
    else:
        if a[0] == b[0]:
            return 1 + lcs(a[1:], b[1:])
        else:
            return max(lcs(a[1:], b), lcs(a, b[1:]))
```

This is terribly inefficient, however, as in the worst case we are calling our function on the order of $O(2^N)$ times.

To improve this, we can use bottom-up dynamic programming. We will gradually build up a matrix that stores the lengths of the longest common subsequences of prefixes of `A` and `B`. To do so, we directly compute the value of any cell `lengths[i + 1][j + 1]` by looking at the values of `lengths[i][j]`, `lengths[i][j + 1]`, and `lengths[i + 1][j]`:

```
def lcs(a, b):
    lengths = [[0 for j in range(len(b) + 1)]
                for i in range(len(a) + 1)]

    for i, x in enumerate(a):
        for j, y in enumerate(b):
            if x == y:
                lengths[i + 1][j + 1] = lengths[i][j] + 1
            else:
                lengths[i + 1][j + 1] = max(lengths[i][j + 1], lengths[i + 1]
[j])

    return lengths[-1][-1]
```

This will run in `O(M * N)` time, where `M` and `N` are the lengths of our two strings, and the lengths matrix will take up `O(M * N)` space.

Now, let's generalize this solution to three strings. First, we must change our grid to be three-dimensional, instead of two, as it must store the lengths of common

subsequences of prefixes of all three strings. Second, we must update the logic for filling up the grid. Similar to above, if the first letters of all three strings are equal, the following relationship will hold: `lcs(a, b, c) = 1 + lcs(a[1:], b[1:], c[1:])`. Otherwise, it is impossible for the subsequence to contain all three starting letters, so we must find the longest solution among `lcs(a, b, c[1:])`, `lcs(a, b[1:], c)`, and `lcs(a[1:], b, c)`.

Putting it all together, we have the following algorithm:

```
def lcs(a, b, c):
    lengths = [[[0 for k in range(len(c) + 1)]
                for j in range(len(b) + 1)]
                for i in range(len(a) + 1)]

    for i, x in enumerate(a):
        for j, y in enumerate(b):
            for k, z in enumerate(c):
                if x == y == z:
                    lengths[i + 1][j + 1][k + 1] = lengths[i][j][k] + 1
                else:
                    lengths[i + 1][j + 1][k + 1] = max(
                            lengths[i][j + 1][k + 1],
                            lengths[i + 1][j][k + 1],
                            lengths[i + 1][j + 1][k]
                            )

    return lengths[-1][-1][-1]
```

Analogously to our previous dynamic programming solution, this runs in `O(M * N * P)` time and space, where `M`, `N`, and `P` are the lengths of our three strings.

Privacy Policy

Terms of Service

Press