Daily Coding Problem #276

Problem

This problem was asked by Dropbox.

Implement an efficient string matching algorithm.

That is, given a string of length `N` and a pattern of length `k`, write a program that searches for the pattern in the string with less than `O(N * k)` worst-case time complexity.

If the pattern is found, return the start index of its location. If not, return `False`.

Solution

A naive pattern matching algorithm would loop through the string, checking each index to see if the pattern begins there. If so, we check whether the next element of the string matches the second element of the pattern, and so on, continuing until the values diverge, or a complete match is found.

However, suppose we have a string consisting of a thousand `A`s followed by a `B`, and our pattern is `AAAAAB`. With this naive solution we will need to make six comparisons for each index we start at, making the runtime `O(N * k)`.

It would be helpful if, instead, we had a way of reusing the knowledge of indices we have already visited. This is exactly the idea behind the Knuth-Morris-Pratt algorithm.

The first step to understanding Knuth-Morris-Pratt is something called the partial match table. This may sound a bit abstract at first, but the reason for it will become clear once we consider the main part of the algorithm. For each index of the pattern, let us consider the substring up to that index. We would like to find the length of the longest string that is both a prefix and a suffix of this substring. To be clear, we only count "proper" prefixes and suffixes; that is, we do not count the string itself.

Let's take an example pattern, such as `ABABC`:

- `i = 0`, `substring = 'A'`. There are no proper prefixes or suffixes, so we return `0`.
- `i = 1`, `substring = 'AB'`. The only prefix is `A`, and the only suffix is `B`. These do not match, so we return `0`.
- `i = 2`, `substring = 'ABA'`. `A` is both a prefix and a suffix, so we return `len('A')`, or `1`.
- `i = 3`, `substring = 'ABAB'`. `AB` is both a prefix and a suffix, so we return `len('AB')`, or 2.
- `i = 4`, `substring = 'ABABC'`. Any suffix must end in `C`, but no proper suffix ends in `C`, so we return `0`.

The full table of partial matches returned, then, would be `[0, 0, 1, 2, 0]`.

```python
def build_table(pattern):
    matches = [0 for _ in range(len(pattern))]

    length = 0
    for i in range(1, len(pattern)):
        if pattern[i] == pattern[length]:
            length += 1
            matches[i] = length

        else:
            while length != 0:
                length = matches[length - 1]
            matches[i] = 0

    return matches
```

There are a few ways to build this table, but an efficient implementation, such as the one above, will run in `O(k)` time, where `k` is the length of the pattern.

For the main part of our algorithm, we will initialize counters for the indices in the string and the pattern. If the values at corresponding indices are equal, we can increment both counters. Once the value of the pattern counter equals to the length of the pattern, we know we have found a complete match. The logic here is very similar to that of the naive solution.

The difference comes when the values from the string and pattern diverge. For example, suppose we are searching for the pattern above, `ABABC`, in the string `ABABABC`. All goes well until we reach the fifth element of the string, `A`, and find that it does not match the corresponding pattern element, `C`. With the naive approach, we would need to start over from the second element in the string.

This is where the partial match table comes in. Since we matched four letters of our pattern, we check the fourth index of our table, or `matches[3]`. The value is `2`, meaning that there is a two-letter prefix of our pattern that ends at `string[3]`. As a result, we can continue by matching `string[4]` against the third character of our pattern. This is displayed in the diagram below, with the vertical lines representing the prefix.

```
A B A B A B C
x x | | ?
    A B A B C
```

By continuing in this way, we never have to restart our search at an earlier index. Furthermore, for each index of the string, we perform constant-time operations that do not depend on the length of the pattern. As a result, this algorithm will run in `O(N)` time, where `N` is the length of the string.

The implementation is as follows:

```
def find_match(string, pattern):
    matches = build_table(pattern)

    i, j = 0, 0
    while i < len(string):
        if string[i] == pattern[j]:
                i += 1; j += 1

                if j == len(pattern):
                    return i - j

            else:
                if j > 0:
                    j = matches[j - 1]
                else:
                    i += 1

        return None
```