Daily Coding Problem #211

Problem

This problem was asked by Microsoft.

Given a string and a pattern, find the starting indices of all occurrences of the pattern in the string. For example, given the string "abracadabra" and the pattern "abr", you should return `[0, 7]`.

Solution

One solution would be to traverse the string, comparing the pattern to every slice of the same size. When we find a match, we append the starting index to a list of matches.

In the example above, we would compare "abr" against "abr", "bra", rac", "aca", and so on. If k is the length of the pattern, we are making up to k comparisons for each slice, so our time complexity is `O(k * N)`.

```
def find_matches(pattern, string):
    matches = []
```

```
    k = len(pattern)
    for i in range(len(string) - k + 1):
        if string[i : i + k] == pattern:
            matches.append(i)
    return matches
```

If we could somehow reduce the time it takes to compare the pattern to new slices of the string, we'd be able to make this more efficient. This is the motivation behind using a rolling hash.

To explain this idea, suppose we wanted to match the pattern "cat" against the string "scatter". First let's assign each letter in the alphabet a value: `a = 1, b = 2, ..., z = 26`. Now let's make a very simple hash function which adds up the values of each letter in the hash. For example, `simple_hash("cat")` would be `3 + 1 + 20 = 24`.

To find our pattern matches, we start by comparing "cat" to "sca". Since `simple_hash("sca")` = `19 + 3 + 1 = 23`, we know we have not found a match, and we can continue. This is where the *rolling* part comes in. Instead of computing the hash value for the next window from scratch, there is a constant-time operation we can do: subtract the value of "s" and add the value of "t". And in fact we find that `23 - value(s) + value(t) = 24`, so we have a match!

Continuing in this way, we slide along the string, computing the hash of each slice and comparing it to the pattern hash. If the two values are equal, we check character by character to ensure that there is indeed a match, and add the appropriate index to our result.

```
def value(letter):
    return ord(letter) - 96

def simple_hash(prev, start, new):
    return prev - value(start) + value(new)

def find_matches(pattern, string):
    matches = []
    k = len(pattern)
```

```
    # First get the hash of the pattern.
    pattern_val = 0
    for i, char in enumerate(pattern):
        pattern_val += value(char)

    # Then get the hash of the first k letters of the string.
    hash_val = 0
    for i, char in enumerate(string[:k]):
        hash_val += value(char)

    if pattern_val == hash_val:
        if string[:k] == pattern:
            matches.append(0)

    # Roll the hash over each window of length k.
    for i in range(len(string) - k):
        hash_val = simple_hash(hash_val, string[i], string[i + k])
        if hash_val == pattern_val:
            if string[i + 1 : i + k + 1] == pattern:
                matches.append(i + 1)

    return matches
```

The problem with this solution, though, is that our hash function is not that good. For example, when we match "abr" against "abracadabra", our pattern will match both "abr" and "bra", requiring us to perform extra string matches.

A more sophisticated rolling hash function is called Rabin-Karp, and a simplified version of it works as follows.

Suppose we have a string of length k. Each letter in the string is first mapped to 1 - 26 as above, then multiplied by a power of 26. The first character is multiplied by 26^{k-1}, the second by 26^{k-2}, and so on, all the way to the kth letter, which is multiplied by 26^0. Finally, we add all these values together to get the hash of the string.

So for example, "cat" will evaluate to: $3 * 26^2 + 1 * 26^1 + 21 * 26^0 = 2075$.

Now suppose we are sliding a rolling window over the word "cats", and we next want to find the hash value of "ats". Instead of computing the value from scratch, we carry out the following steps:

- subtract the value of the first letter of the current hash ($3 * 26^2$)
- multiply the current hash value by 26
- add the value of the last letter in the shifted window (19)

Using these steps, the value of "ats" will be $(2075 - 3 * 26^2) * 26 + 19 = 1241$.

This works because we are essentially shifting the powers of 26 leftward. It may be easier to see algebraically that these two equations are equal:

$((3 * 26^2 + 1 * 26^1 + 21 * 26^0) - 3 * 26^2) * 26 + 19$

$1 * 26^2 + 21 * 26^2 + 19 * 26^0$

If we replace our simple hash function with the new and improved one, our solution should look like this:

```
def value(letter, power):
    return (26 ** power) * (ord(letter) - 96)

def rabin_hash(prev, start, new, k):
    return (prev - value(start, k - 1)) * 26 + value(new, 0)

def find_matches(pattern, string):
    matches = []
    k = len(pattern)

    pattern_val = 0
    for i, char in enumerate(pattern):
        pattern_val += value(char, k - i - 1)

    hash_val = 0
    for i, char in enumerate(string[:k]):
```

```
        hash_val += value(char, k - i - 1)

    if pattern_val == hash_val:
        if string[:k] == pattern:
            matches.append(0)

    for i in range(len(string) - k):
        hash_val = rabin_hash(hash_val, string[i], string[i + k], k)
        if pattern_val == hash_val:
            if string[i + 1 : i + k + 1] == pattern:
                matches.append(i + 1)

    return matches
```

In the worst case, if our hash function produces many false positives, we will still have to check each substring against the pattern, so this would take `O(k * N)`.

Practically, however, we should not see too many false positives. In the average case, since our algorithm takes `O(k)` to compute the hash of the pattern and the start of the string, and `O(N)` to roll the hash over the rest of the string, our running time should be `O(k + N)`.
