String Matching via a Rolling Hash Function

The string matching problem is as follows:

Given a text string of length n, and a pattern to search for in the text of length m, determine the locations (if any), in the text that the pattern appears.

This, if we have:

```
text = SALLYSELLSSEASHELLSBYTHESEASHORT

pattern = SEA
```

then the algorithm should identify the two locations highlighted in yellow as containing the pattern.

The naïve brute force algorithm runs in O(nm) time. (In practice, it'll often be better, because many tries in matching up the pattern will fail on the first comparison, so the rest can be skipped.)

The Knuth-Morris-Pratt algorithm solves this problem in O(n+m) time, but hashing can also do so (probabilistically), as well.

Here is the idea:

We define the hash of a string to be its base k value under mod p, where p is a large prime. (If the string is all uppercase letters, we could choose k = 26.) Thus, the hash value of SEA would be ('S' x 26^2 + 'E' x 26^1 + 'A') mod p, where we map each letter to a value 0 to 25.

Once we calculate the hash value of the pattern, all that is left to do is calculate the hash value of each substring of length m in the text. In this example, the first few are "SAL", "ALL", and "LLY". Any time a hash value is different, we have **proof** the substring in that location doesn't match. If the hash values are equal, then it's a potential match. (We could just check it out to make sure, to remove the probabilistic element.)

Notice that running a for loop to calculate this hash value for every substring doesn't save us any time since if we spent O(m) time calculating the hash for each substring, we still arrive at an O(nm) run time.

But notice that the hashvalue of "SAL" and "ALL" are related:

hash("SAL") = ('S' x 26^2 + 'A' x 26^1 + 'L') mod p
hash("ALL") = ('A' x 26^2 + 'L' x 26^1 + 'L') mod p

The portion in green is just the portion in yellow, multiplied by 26. Thus, to calculate ALL's hash value, we need to do the following:

hash("ALL") = [26 x ('A' x 26^1 + 'L')] + 'L'
= [26 x (hash("SAL") – 'S' x 26^2)] + 'L'

Thus, to "transition" from hash("SAL") to hash("ALL"), instead of computing hash("ALL") from scratch, we can subtract out the contribution of 'S', then multiply this by 26, finally, adding in the contribution of 'L'.

Though the savings aren't entirely apparent in this example, if the pattern string was longer, it should be easy to see that this computation is O(1) extra work, no matter how long the pattern is.

Thus, this brings down the run time to O(n+m), as desired.

The code example HashExample.java provides one possible implementation.