Knuth-Morris-Pratt algorithm (KMP)

Salvador Roura

The main use of the KMP algorithm is to efficiently search for all the instances of a word w into a text t.

The naive C++ code to solve this problem would be like this:

```
void find_matches(string w, string t) {
  int m = w.size();
  int n = t.size();
  for (int i = 0; i <= n - m; ++i)
    if (t.substr(i, m) == w) cout << i << endl;
}
```

For instance, if $w =$ `"aba"` and $t =$ `"aabaacaabaa"`, this code prints 1 and 7.

In general (unless $m = 0$ or $m \simeq n$), the code above has cost $\Theta(m \cdot n)$. Similar codes have also cost $\Theta(m \cdot n)$, at least on the average.

We can do better. An alternative solution is the Rabin-Karp algorithm, which uses a rolling hash to solve this problem in $\Theta(m + n)$ cost. (If you are curious, you can find the details online.)

However, the Rabin-Karp algorithm has the following faults (in comparison to KMP):

- The code is a bit error-prone.
- When the hash codes are equal, if we check if w trully matches the current position of t, the the cost can become $\Theta(m \cdot n)$. And if we don't perform this extra check, the algorithm has a small (but non-zero) probability of giving false positives.
- KMP is also a liner-time algorithm, but its constant is usually smaller, bacause it avoids expensive operations like modulos, which are typical of hash functions.

Before we present the KMP algorithm, we need a few definitions:

A *prefix* of a string s is a substring of s that occurs at the beginning of s. For instance, let s = `"aabaacaabaa"`. The prefixes of s are λ (the empty string), `"a"`, `"aa"`, `"aab"`, ..., `"aabaacaaba"`, and `"aabaacaabaa"` (s itself).

A *suffix* of a string s is a substring of s that occurs at the end of s. With the example above, the suffixes of s are λ, `"a"`, `"aa"`, `"baa"`, ..., `"abaacaabaa"`, and `"aabaacaabaa"`.

A *border* of a string s is a substring b of s that is a prefix and also a suffix of s, and such that $b \neq s$. With the example above, the borders of s are λ, `"a"`, `"aa"`, and `"aabaa"`.

There is a simple property that is key to KMP: *The border of a border is a border.* In the example, the borders of `"aabaa"` are indeed λ, `"a"`, and `"aa"`.

Now, we are ready to present the KMP algorithm. This is one of its possible implementations:

```
vector<int> kmp(string s) {
  int n = s.size();
  vector<int> P(n);
  int j = −1;
  for (int i = 0; i < n; ++i) {
    while (j ≥ 0 and s[j] ≠ s[i]) j = (j ? P[j−1] : −1);
    P[i] = ++j;
  }
  return P;
}
```

If we call this procedure with s = `"aabaacaabaa"`, the content of the resulting vector P is

	0	1	2	3	4	5	6	7	8	9	10
s	a	a	b	a	a	c	a	a	b	a	a
P	0	1	0	1	2	0	1	2	3	4	5

For every position i between 0 and $n-1$, $P[i]$ contains the length of the longest border of $s[0..i]$. For instance, $P[4] = 2$ corresponds to `"aa"`, the longest border of $s[0..4]$ = `"aabaa"`.

Another key to KMP is the following: *Iterating P provides all the borders of a string.* In the example, $P[n-1] = P[10] = 5$ provides `"aabaa"`, $P[5-1] = 2$ provides `"aa"`, $P[2-1] = 1$ provides `"a"`, and $P[1-1] = 0$ provides λ.

The algorithm fills the vector P from left to right, and computes each $P[i]$ using the values of $P[j-1]$ for $1 \leq j \leq i$. To understand how it works, suppose that we want to compute P for a string s' equal to s, but with a character appended to its right. Note that the first 11 positions of P are those already computed, and that we only need to calculate $P[i]$ for $i = 11$. The value of j at that moment is 5, just stored at $P[10]$.

First, assume $s'[11] =$ `'c'`, that is, $s' =$ `"aabaacaabaac"`. Because $P[10] = 5$, we already know that $s'[0..4] = s'[6..10]$. The code compares $s'[5]$ against $s'[11]$. Since both are `'c'`, we can deduce $s'[0..5] = s'[6..11]$ with only one character comparison, and so the longest border ending at $s'[11]$ has length 6. Therefore, the **while** stops and we store 6 at $P[11]$.

Now, assume $s'[11] =$ `'b'`, that is, $s' =$ `"aabaacaabaab"`. The comparison of $s'[5]$ against $s'[11]$ fails. Consequently, the code tries the next border, whose length can be found at $P[j\text{-}1] = P[4] = 2$. Now we successfully compare $s'[2]$ against $s'[11]$, so $P[11] = 3$.

Similarly, when $s'[11] =$ `'a'` we get $P[11] = 2$ after three iterations, and when $s'[11] =$ `'d'` we get $P[11] = 0$ after four iterations. Note that this last case requires some care to end the **while** avoiding an access to $P[\text{-}1]$.

What is the cost of the KMP algorithm? An elemental reasoning provides the upper bound $O(n^2)$: we have n iterations, each with cost $O(n)$. Let us do a tighter analysis. First, note that the cost is dominated by the number of times that the $j = (j \ ?\ P[j-1] : -1)$; instruction is executed. Second, every time it is executed, j is decreased by at least 1. But how many times can we decrease j? At most as many times as we increse it (++j), that is, at most n times. Therefore, the cost is no more (nor less) than $\Theta(n)$.

We have yet to show how to use KMP to search for all the instances of a word w into a text t. This is how: For simplicity, assume that there is a special character that does not appear in w nor in t, say `'#'`. First, we concatenate w, the special character and t. Afterwards, we call KMP over this string, and search for the lenght of w into the vector P:

```
void find_matches(string w, string t) {
  string s = w + "#" + t;
  vector<int> P = kmp(s);
  int m = w.size();
  for (int i = m + 1; i < s.size(); ++i)
    if (P[i] == m) cout << i - 2*m << endl;
}
```

For instance, with the first example $w =$ `"aba"` and $t =$ `"aabaacaabaa"`, we build $s =$ `"aba#aabaacaabaa"`. The resulting vector P will only have a 3 at the positions 7 and 13, corresponding to the two matches. We substract m twice from i to make this algorithm functionally equivalent to the naive code, although much faster in general (linear versus quadratic).