Knuth-Morris-Pratt

The Rabin-Karp algorithm achieves its excellent average-case running time by converting the strings to integers, using fast integer operations, and most of all by counting on the fact that in most applications **P** will not occur very often in **T**. However, the worst-case running time of Rabin-Karp is O(n*m), which is the same as the Brute Force and Ignorance algorithm. In fact, the worst-case running time for the two algorithms results from exactly the same situation:when **T** consists of a single character repeated **n** times, and **P** consists of **m** repetitions of the same character. In this case, every shift is a possible match and they must all be checked completely to determine that they are perfect matches.

The Boyer-Moore algorithm does character-by-character comparisons but uses information about the pattern string P to pre-compute safe shifts – situations where we can skip over some shifts because they cannot possibly yield perfect matches. The key insight of the Boyer-Moore algorithm is that by starting the comparisons at the ***right*** end of P we can potentially identify larger safe shifts than if we started comparisons at the ***left*** end of P.

In class I described how we can implement Boyer-Moore so that each character in T is looked at no more than a constant number of times (basically when we shift P over to line up with some characters of T, we don't re-examine the ones that we just lined up). It has been shown that with careful implementation, the worst that can happen is that each character of T is examined no more than 3 times. This gives the algorithm O(n) complexity. Note that this does not include the pre-computation of the two look-up tables, but these depend on m, which is typically much smaller than n.

The Knuth-Morris-Pratt algorithm actually predates the Boyer-Moore algorithm by several years. It was the first O(n) string-matching algorithm discovered. In practice the Boyer-Moore is more generally useful, but KMP is still worth looking at because it has some interesting features.

It's a bit of missing link between BFI and BM ... except that it's not missing. Like BFI, it compares P to substrings of T from left to right. But like BM it uses information about P to find safe shifts. And surprisingly it actually has a better worst-case time bound than BM does.

The Suffix Match rule for Boyer-Moore is based on the idea that if we have found an incomplete match in which the last few characters of P matched the corresponding characters in T then we can shift P to the right to line up another copy of the characters that matched.

KMP does the same thing but since it matches characters from left to right it bases its safe shifts on the size of the ***prefix*** of P that matched. The idea is quite simple: if we are testing a shift and we find that the first i characters of P match their counterparts in T but the $i + 1^{st}$ character does not, then we can shift P to the right as long as we are not in danger of missing a valid match.

An example will clarify (I hope). Suppose P = "ATATGCAT" and during the testing of shift we see this:

```
T : ........A T A T ? . . . . . .   where ? is anything but G

P   :       A T A T A C A T
```

The prefix of P that has matched is "ATAT", then the matching fails on the next character.

Consider the next shift

```
T : ........A T A T ? . . . . . .

P :           A T A T G C A T
```

This can't possibly work because we are lining up the first character of P ("A") with a character in T that we know is a "T". So we can skip over this shift. Consider the next one:

```
T : ........A T A T ? . . . . . .

P :             A T A T G C A T
```

Now within the characters of T that we know ("ATAT") we have pushed P to a point where we ***could*** be at the start of a perfect match. So we can't skip over this shift – we have to stop and test it. But notice that we don't have to test the "AT" in T lined up with the beginning of P – we chose this shift because those characters matched and we can start our testing with the ? character.

Another example: Suppose P = “ACAGAACAGTACA”

Let’s look at a few of the possible prefix-matches

If we see

```
T : . . . . . .A C ? . . . . . .
P :         A C A G A A C A G T A C A
```

We can shift to

```
T : . . . . . .A C ? . . . . . .
P :             A C A G A A C A G T A C A
```

If we see

```
T : . . . . . .A C A G A ? . . . . . .
P :         A C A G A A C A G T A C A
```

We can shift to

```
T : . . . . . .A C A G A ? . . . . . .
P :                 A C A G A A C A G T A C A
```

If we see

```
T : . . . . . .A C A G A A C A ? . . . . . .
P :         A C A G A A C A G T A C A
```

We can shift to

```
T : . . . . . .A C A G A A C A ? . . . . . .
P :                   A C A G A A C A G T A C A
```

If we see

```
T : .......A C A G A A C A G T A C A . . . . . .
P :         A C A G A A C A G T A C A
```

(a perfect match), we can shift to

```
T : .......A C A G A A C A G T A C A . . . . . .
P :                           A C A G A A C A G T A C A
```

So what is the principle at play? Basically, when a prefix of P matches a substring of T (up to and including the full string P), we look for the longest prefix of that prefix that matches an identical suffix of the prefix. (Why am I reminded of the Dr. Seuss poem that references the north-eastern west part of South Carolina? Bonus marks if you know the poem.)

Going back to one of the examples we just looked at: when the matching prefix of P is "ACAGAACA" we see that the longest prefix of this that is also a suffix of this is "ACA". We shift P over until the "ACA" at the beginning is lined up where the other "ACA" used to be. Remember these characters all matched with their counterparts in T so we know the "ACA" at the beginning of P will now be lined up with an "ACA" in T . You should experiment with this until you are convinced that no smaller shift is worth checking because at least one character of P will be mismatched with its counterpart in T.

A non-intuitive fact about this is that as we consider longer and longer matching prefixes of P, the safe shifts that we discover ***never*** decrease in size. The identical prefixes and suffixes in the matching prefix can go up and down in length, but the safe shifts either stay the same or get bigger.

Note that – just as with Boyer-Moore – we can find all these matching prefixes and suffixes just from P. This means we can compute them before we even look at T. (More recent versions of this algorithm compute the safe shift info on the fly during the actual searching, but we're going to stay old-school for this discussion.)

Let's work the whole thing out for P = "ACAGAACAGTACA"

Matching Prefix of P	Length of Prefix	Length of Identical Prefix/Suffix	Safe Shift
""	0	0	1
A	1	0	1
AC	2	0	2
AC**A**	3	1	2
ACAG	4	0	4
ACAG**A**	5	1	4
ACAGA**A**	6	1	5
ACAGA**AC**	7	2	5
ACAGA**ACA**	8	3	5
ACAGA**ACAG**	9	4	5
ACAGAACAGT	10	0	10
ACAGAACAGT**A**	11	1	10
ACAGAACAGT**AC**	12	2	10
ACAGAACAGT**ACA**	13	3	10

Now we can figure out how to compute the safe shifts: The first two rows of the table are the same for every P. If we match a prefix of length 0 or 1, the safe shift is always 1. For a matching prefix of length 2, the safe shift is 1 if the two characters in the matching prefix are identical and 2 if they are different (as in the example).

Suppose we have correctly computed the safe shift for the prefix of length i, where $i \geq 2$. To compute the safe shift for the prefix of length *i+1*:

```
let L be the length of the Identical Prefix/Suffix for the prefix of
length i

if P[L+1] == P[i+1]:

     the Identical Prefix/Suffix grows by 1 character

     the safe shift remains the same

else:

     the Identical Prefix/Suffix shrinks until it is valid.  This
          shrinking is achieved by repeatedly shortening the suffix
          until it matches the corresponding prefix.  Let k be the
          length of the new identical prefix/suffix (k == 0 is
          possible)

     the safe shift increases to i+1-k
```

Here's another example, showing the shrinking mentioned in the "else" clause.

Suppose P = "ACAGAACAAC"

In this table I have underlined the characters that get compared at each step, and the final identical prefix/suffix for each prefix of P

For some reason LibreOffice won't let me put the table on this page so here's a peaceful scene that nicely fills the space.

Prefix of P	length	Length of identical prefix/suffix	Safe shift
A	1	0	1
AC	2	0	2
AC**A** ? Match extends **A**C**A**	3	1	2
A**C**A**G ?** Match fails **A**CA**G** ? Match fails ACAG	4	0	4
ACAG**A** ? Match extends **A**CAG**A**	5	1	4
A**C**AGA**A** ? Match fails **A**CAGA**A** ? Match succeeds **A**CAGA**A**	6	1	5
A**C**AGAA**C** ? Match extends **AC**AGA**AC**	7	2	5
AC**A**GAAC**A** ? Match extends **ACA**GA**ACA**	8	3	5
ACA**G**AACA**A** ? Match fails **A**CAGAA**C**AA ? Match fails **A**CAGAAC**A**A ? Match succeeds A**C**AGAACA**A** ? Match fails **A**CAGAACA**A** ? Match succeeds **A**CAGAACA**A**	9	1	8
A**C**AGAACAA**C** ? Match extends **AC**AGAACA**AC**	10	2	8

The box for the prefix of length 9 needs the most explanation. After the line for the prefix of length 8, we have a Prefix/Suffix pair of "ACA". The next character after the prefix is "G" and the next character after the suffix is "A". "ACAG" (prefix) and "ACAA" (suffix) don't match so we can't extend the current Prefix/Suffix pair. We start shrinking the suffix by chopping characters off its left end, trying to find a match for the start of P. After the first chop, the suffix is "CAA" - this doesn't start with "A" (first character of P) so it fails. We chop again getting a suffix of "AA". This **does** start with "A" so we have a potential Prefix/Suffix pair. We check the next character of the Prefix ("C") with the next character of the Suffix ("A") - they don't match so it's a fail. We chop the Suffix again, leaving just "A". This **does** match the beginning of P so we have our new Prefix/Suffix pair.

Computing the safe shifts for KMP is actually pretty easy once you see what is being done. I encourage you to code this. The safe shift computation for KMP can be done in O(m) time.

Now let's consider the actual search algorithm – it's a lot like BM. Starting from shift 0 we compare characters until we get a mismatch. We use the length of the matched prefix to look up the next safe shift. We continue doing this until we reach the end of T.

It's actually really easy to analyze the complexity of KMP. Each character in T gets examined exactly once, except for the ones that cause mismatches – they get examined at most twice (once when they cause the mismatch, and possibly once again after the safe shift). Thus the searching completes with $\leq 2 * n$ comparisons. This is better than the worst-case performance of BM (but remember, most real applications do not exhibit worst-case behaviour, and BM average-case is very good).

Well, is it possible to do better? In particular, is it possible to find a string matching algorithm for which the worst-case performance examines every character in T exactly once?

The answer is Yes ... but I'm stopping these notes here!