Boyer-Moore Algorithm

The Rabin-Karp algorithm tests all n-m+1 shifts, but attempts to minimize the time required for each test. Our next two algorithms attempt to minimize the number of shifts to be tested. Both are based on the idea of a **safe shift** : when we have tested a shift, whether or not it gave a match, we attempt to shift P forward as far as we can without missing any future perfect matches. Clearly we can always shift by 1, since that is the next shift in sequence. We will look at techniques that may let us shift as much as m (ie after testing shift i, we jump directly to shift i+m). It is important to understand that a safe shift does not guarantee a perfect match on the next comparison - it merely guarantees that no perfect match will be missed.

Finding ways to compute safe shifts and trying to make them as large as possible, is a problem that rewards creativity and insight.

The Boyer-Moore Algorithm is based on the clever idea of testing each shift in **right to left** order. Thus when comparing P to T[1+i .. m+i], we first compare P[m] to T[m+i], then if those match, we compare P[m-1] to T[m+i-1] etc.

```
# testing shift i
x = m
while x >= 1 and P[x] == T[x+i]
    x -= 1
if x == 0:
    print "Match found at shift ",i
```

It may not seem to make sense that comparing from the right can be any more useful than comparing from the left, but an example will hopefully make it clear.

Suppose P = "ATATACAT" and we are comparing it to a piece of T that looks like "CACCGATG". If we compare these from left to right we would hit a mismatch on the first comparison (the "A" at the left end of P and the “C” at the start of the substring of T) - without looking at more characters of T, we can only shift forward one space before stopping to test again. However if we do the comparison from right to left, we again hit a mismatch on the first comparison (the “T” at the end of P and the “G” at the end of the substring of T). If we have taken the time to list the characters in P before we start, we know there is no "G" in P ... so we can shift P forward to just past the "G" in T without missing any potential matches. In this example, this gives a safe shift of 8.

From

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

we shift all the way to

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

based on just one comparison. Obviously we won't always be this lucky, but in a real world application we do often get to skip over many potential shifts after just a couple of comparisons.

This brings up the idea of pre-computation. Boyer-Moore is all about pre-computing information about P. In the version we will look at we will use two techniques to compute safe shifts. Both are guaranteed to produce safe shifts, but possibly of different sizes (for example, sometimes one technique will give a safe shift of size 1 and the other will give a safe shift of size 2 or more). Since we want to shift as far as possible each time, we simply choose the larger of the two.

Technique 1: The "First Mismatch" rule (sometimes called "Bad Character" - like "Angel Eyes" in "The Good, the Bad and the Ugly"). This is based on the same principle as the example above. We create a table that tells us, for each letter in the alphabet and for each position in P, how far we can shift if the first mismatch we find has that letter lined up on that position in P.

Let's work it out for P = "ATATACAT"

The table shows the safe shifts. A detailed explanation of how they are computed is given below the table.

		Letter in T at the first mismatch			
		A	C	G	T
Number of letters successfully matched	0	1	2	8	-
	1	-	1	7	3
	2	1	-	6	2
	3	-	5	5	1
	4	1	4	4	-
	5	-	3	3	1
	6	1	2	2	-
	7	-	1	1	1

In the example I did in class I added a row for matching all the letters – ie. we find a perfect match – in this case there is no mismatch and so there really isn't a way to apply this rule – but a shift of 1 is always safe so we can implement our rule to return 1 in the case of a perfect match.

The "-" entries indicate no value for these cases because they cannot arise. For example we can't have a mismatch after 2 matches if the next letter in T is "C", because that would actually extend the match.

To compute the values in the first row, we imagine that we are comparing P to a substring of T and we get the first mismatch after matching 0 letters – ie. The first letter of T we look at doesn't match the corresponding letter of P.

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

Because we are doing this hypothetically, we don't know what the "?" is, but we know it isn't "T". The options are limited:

If it is an "A", we can shift P by 1 position to line up the "A" just before the "T"

```
T: ..................x x x x x x x A ...........
P:                      A T A T A C A T
```

If it is a "C", we can shift P by 2 positions to line up the "C" two positions before the "T"

```
T: ..................x x x x x x x C ...........
P:                        A T A T A C A T
```

If it is a "G", we can shift P by 8 positions because there is no "G" in P

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

The method for filling in the table is simple: just count backwards in P to find the closest occurrence of the letter we just found in T. You should probably check my table entries – I think they are correct.

Remember that these shift rules are concerned **only** with making safe shifts, not with checking to see if the shifts they recommend will be likely to give a perfect match. For example, if we encounter a "G" after matching 3 letters, the First Mismatch rule says we can safely shift 5 positions, positioning P just to the right of that "G" in T. That is certainly a safe shift, but look what it gives us:

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

Being smart humans we can immediately see that this isn't going to give us a perfect match: we have just lined up "ATA" under "CAT". The First Mismatch rule doesn't care. It bases its recommendation completely on the one piece of information it has – the identity of the letter in T that did not match.

This is why we have a **second** technique.

Technique 2: the "Suffix Match" rule (sometimes called the "Good Suffix" - I don't have a joke for this one). Once again we consider hypothetical situations in which a partial match has been found, but instead of focusing on the letter that didn't match, we base our recommended safe shift on the sequence of letters that **did** match.

Example: Suppose we match 2 letters and then hit a mismatch

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

So the "?" is some letter other than "C". For this rule we don't care what it is so we don't work out different cases. We think about pushing P to the right. Obviously there is no point pushing it 1 position: that lines up "CA" below "AT" and that won't be a perfect match. The same reasoning applies if we push it 2 or 3 positions. But if we push it 4 positions we get

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

and this, based on the limited information we have, **could** be a perfect match – so we put P here and start the comparison process again (starting at the right end of P).

Again, our goal is to skip over shifts that we can see cannot give perfect matches, and stop and check the first shift that **might** give a perfect match. In this way we guarantee that no perfect matches will be missed.

Once again we precompute all the information we need. We construct a table that tells us how far we can safely shift, based on the number of letters that matched (ie. the length of the matching suffix).

For our example of P = "ATATACAT" the resulting table looks like this:

			Safe Shift
		Matched Suffix	
Number of letters successfully matched	0	""	1
	1	T	4
	2	AT	4
	3	CAT	6
	4	ACAT	6
	5	TACAT	6
	6	ATACAT	6
	7	TATACAT	6

Here's one way to compute those Safe Shift numbers (people have come up with other, very clever ways of computing these numbers very quickly, but this simple method will work for our explanation).

Suppose we have matched a suffix of length 1 (ie the "T" in this example). We search back in P to find the closest "T". It's 2 positions over so our safe shift is 2.

Now suppose we have matched a suffix of length 2 (ie "AT") We see if the "T" we found for the safe shift just a moment ago has an "A" just before it – it does! So we have found an instance of "AT" that can be shifted to line up with the "AT" we just found in T. The shift amount is the same.

Now suppose we have matched a suffix of length 3 (ie "CAT") Again we try to extend the internal match – we can't: the "AT" we are working on has a "T" in front of it, not a "C". So we scan to the left for the next "T" and check to see if this has "CA" in front of it. It doesn't. So we do it again. There's one more "T" to examine – it just has an "A" in front of it. There is no occurrence of "CAT" in P other than the one at the end, so there is no way to shift P to line up a "CAT" under the "CAT" we have just found in T.

At this point we might hope that we can shift P over completely, but we can't! The first two letters of P actually match the last two, and the two letters of T above them **might** be the start of a perfect match.

From

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

we go to

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

and note that this safe shift (6) stays the same for all the larger suffix matches.

We should look at another example. Suppose P = "GACCATATCAT"

Matching Suffix at Current Shift	Closest copy of matching suffix	Safe shift	Next shift
T: xxxxxxxxxxT.. P: GACCATATCAT	P:GACCATATCAT	3	T: xxxxxxxxxxT... P: GACCATATCAT
T: xxxxxxxxxAT.. P: GACCATATCAT	P:GACCATATCAT	3	T: xxxxxxxxxAT... P: GACCATATCAT
T: xxxxxxxxCAT.. P: GACCATATCAT	P:GACCATATCAT	5	T: xxxxxxxxCAT.. P: GACCATATCAT
T: xxxxxxxTCAT.. P: GACCATATCAT	none	11	T: xxxxxxxTCAT.. P: GACCATATCAT
T: xxxxxxATCAT.. P: GACCATATCAT	none	11	T: xxxxxxxTCAT.. P: GACCATATCAT
T: xxxxxTATCAT.. P: GACCATATCAT	none	11	T: xxxxxxxTCAT.. P: GACCATATCAT
	same		
		all	
			the
			way
			down

So what was different about this example that allowed us to start shifting right past the end of the matched suffix, instead of sticking with the safe shift for the last suffix that actually had a full internal match (in this example, that was "CAT")?

When we failed to find a copy of the matched suffix "TCAT" we **weren't** working on a partial match at the very beginning of P. In our previous example, we **were**. In other words, this P

has no prefix that matches any of its suffixes (that "G" at the beginning makes this true). This makes it safe to shift P all the way past the matched suffix.

Note that if we don't want to make the effort to be clever this way, we won't get an error by using the shift for the largest matched suffix that has an earlier copy in P for all larger suffixes that don't – but our algorithm may not be as efficient as possible because there may be situations where we don't make as big a shift as we could.

Now we can demonstrate why we need both rules for computing safe shifts. Compare the two tables for our original example:

First Mismatch:

		Letter in T at the first mismatch			
		A	C	G	T
Number of letters successfully matched	0	1	2	8	-
	1	-	1	7	3
	2	1	-	6	2
	3	-	5	5	1
	4	1	4	4	-
	5	-	3	3	1
	6	1	2	2	-
	7	-	1	1	1

Suffix Match:

			Safe Shift
		Matched Suffix	
Number of letters successfully matched	0	“”	1
	1	T	4
	2	AT	4
	3	CAT	6
	4	ACAT	6
	5	TACAT	6
	6	ATACAT	6
	7	TATACAT	6

Suppose when we compare P to some substring of T, we match just the last letter (the first one we check). The “Suffix Match” table says “shift 4”, and the “First Mismatch” table says “shift 1, 7 or 3, depending on what the letter in T is at the mismatch point”. We evaluate both rules – they are both recommending shifts that are guaranteed safe – so we choose the larger recommended shift. In some cases the First Mismatch table will give us a larger shift and in other cases the Suffix Match rule will win.

Complexity:

How much work is involved in building those tables? It may look daunting but it's actually pretty straightforward – you should be able to code it based on the explanations given. It's possible to do it all in O(m) time but it's a bit tricky. I encourage you to look up the details – they are easily found.

Note that the tables are computed from P, without any information about T other than the alphabet in use. If we are going to use P as a pattern for searching several target strings (like on CSI where they run a DNA sample against millions of stored DNA sequences) then we just compute the tables once and use them over and over again.

Once we have the tables the actual searching is very efficient. The best situation is where we compare just a few letters for a shift and then hit a mismatch or get a suffix match that allows a really large jump to the next shift to be tested. If this happens on most of the shifts we test, then we will only test about $\frac{n}{m}$ shifts. One of the strengths of the Boyer-Moore algorithm is that it works really well when the alphabet is large (unlike our examples in which the alphabet is tiny). A large alphabet increases the chance of letters mismatching and the chance of suffixes of P not having copies within P – both of which can lead to large shifts. The more large shifts we make, the fewer places we actually have to test.

Boyer-Moore runs very quickly in practice. Searching for all occurrences of a word or phrase in a text document is an ideal application. However, the original form of the Boyer-Moore algorithm had O(m*n) worst-case complexity – the same as the BFI algorithm. A couple of years after the algorithm was published Galil suggested what seems like a minor improvement: when we compute a safe shift, it usually results in 1 or more letters of P aligned with letters in T. Galil's insight was that we know these letters match (because we lined them up) so we don't need to compare them again.

With this modification it has been proved that Boyer-Moore runs in O(n) time, even in the worst case.