Exercise

Simulate the string-matching automaton algorithm for

$$P \equiv \text{abababa}$$

and

$$T = \text{abacbababababaacbacaababababaababababababacac}.$$

Hints. You will need to:

1. Define the automaton (in particular compute its transition function).
2. Simulate step-by-step the algorithm FINITE-AUTOMATON-MATCHING.

We will go through this together during the lecture but, please, try it on your own first.

Knuth, Morris, Pratt algorithm

The major inefficiency of the automaton algorithm is in the computation of the automaton itself.

Knuth, Morris, and Pratt's algorithm achieves running time linear in $n + m$ by using just an auxiliary function π, defined over the states of the automaton, precomputed from the pattern in time $O(m)$.

Roughly speaking, for any state q and any character $a \in \mathcal{A}$, $\pi[q]$ contains the information that is independent of a and is needed to compute "on the fly" $\delta(q, a)$.

Prefix function

The *prefix function* for a pattern P, is the function $\pi : \{1, \ldots, m\} \to \{0, \ldots, m-1\}$ such that

$$\pi[q] = \max\{k : P_k \text{ is a proper suffix of } P_q\}$$

Example. Let $P = 113\ 111\ 513\ 113$. The corresponding prefix function is

q	1	2	3	4	5	6	7	8	9	10	11	12
$\pi[q]$	0	1	0	1	2	2	0	1	0	1	2	3

To define, say, $\pi[6]$ we consider $P_q \equiv 113\ 111$, and then all prefixes $P_{q-1}, P_{q-2}, \ldots, \varepsilon$. We find out that $P_2 (\equiv 11)$ is a suffix of P_q. Hence $\pi[6] = 2$.

Algorithm

```
KMP-MATCHING (T, P)
        n ← length(T)
        m ← length(P)
        π ← COMPUTE-PREFIX-FUNCTION (P)
        q ← 0
        for i ← 1 to n
(*)         while (q > 0 ∧ P[q + 1] ≠ T[i])   q ← π[q]
(*)         if (P[q + 1] = T[i])   q ← q + 1
            if q = m
                print "pattern occurs with shift i − m"
(*)             q ← π[q]
```

COMPUTE-PREFIX-FUNCTION (P)

$m \leftarrow \text{length}(P)$

$\pi[1] \leftarrow 0$

$k \leftarrow 0$

for $q \leftarrow 2$ **to** m

  while $(k > 0 \wedge P[k+1] \neq P[q]) \quad k \leftarrow \pi[k]$

  if $(P[k+1] = P[q]) \quad k \leftarrow k + 1$

  $\pi[q] \leftarrow k$


```
class KMPAlgorithm extends MatchAlgorithm{

    void match(int counter[],
                StringBuffer T,
                StringBuffer P,
                String alphabet) throws Exception{

        counter[0] = 0;
        int i, k, q;
        int pi[] = new int[100];

        // computation of the pi function
        int m = P.length();
        pi[0] = 0;
        k = 0;
        for( q = 1 ; q < m  ; q++ ) {
            while( ( k > 0 )  &&
                    ((P.toString()).charAt(k) != (P.toString()).charAt(q)))
                 k = pi[k - 1];
            if ((P.toString()).charAt(k) == (P.toString()).charAt(q))
                 k = k + 1;

            pi[q] = k;
        }
```



```
        // pattern matching
        int n = T.length();
        q = 0;
        for( i = 0 ; i < n ; i++ ) {
            counter[1] = 1;
            if (stopRequested)
                 return;

            while(( q >  0 ) &&
                   ((P.toString()).charAt(q) != (T.toString()).charAt(i)) )
                 q = pi[q - 1];
            if ((P.toString()).charAt(q) == (T.toString()).charAt(i))
                 q = q + 1;

            counter[0] = (( i - q + 1 )*6 );
            if (( (i-q) < n - m ) && ( q != m )) pause(1,1);

            if ( q == m ){
                 counter[1] = 0;
                 counter[0] = (( i - m + 1 )*6 );
                 pause(1,1);
                 q = pi[q - 1];
            }
        }
    }
}
```


Exercises

1. Simulate the behaviour of the three algorithm we have considered on the pattern $P \equiv$ abc and the text $T =$ aabcbcbabcabcabc.
2. Count the number of instructions executed in each case and find out how the algorithms rank with respect to running time.
3. Repeat exercise 1. and 2. with the text $T =$ abababababababab. Comment on the results!

One more exercise

Let $T =$ abdcababdcabdcb and $P =$ abdcabd.

(1) We first compute the prefix function.

q	1	2	3	4	5	6	7
$\pi[q]$	0	0	0	0	1	2	3

(2) Next we simulate KMP-MATCHING, starting with $n = 12$, $m = 7$ and $q = 0$.

i	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
T	a	b	d	c	a	b	a	b	d	c	a	b	d	c	b
q	0														

$i = 1$, q is NOT positive so the **while** loop is skipped, $P[q+1]$ is equal to $T[i]$ so q becomes one, and we move to the next iteration

i	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
T	a	b	d	c	a	b	a	b	d	c	a	b	d	c	b
q	0	1													

$i = 2$, q IS positive, but $P[q+1] = T[i]$ so the **while** loop is skipped again, and q is increased to two, and we move to the next iteration.

i	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
T	a	b	d	c	a	b	a	b	d	c	a	b	d	c	b
q	0	1	2												

$i = 3$, $i = 4$ up to $i = 6$ same story, q is successively increased, and each time we move to the next iteration.

i	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
T	a	b	d	c	a	b	a	b	d	c	a	b	d	c	b
q	0	1	2	3	4	5	6								

$i = 7$, q IS positive AND $P[q+1] \neq T[i]$, we run $q \leftarrow \pi[q]$ inside the **while** loop twice (after the first time we ackowledge overall failure but we try to find $P_3 \equiv$ abd, after the second time we have completely given up and we decide we will start almost from scratch, having matched $P_1 \equiv$ a).

i	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
T	a	b	d	c	a	b	a	b	d	c	a	b	d	c	b
q	0	1	2	3	4	5	6 2 0	1							

$i = 8$ up to $i = 12$, nothing exciting happens, q keeps increasing ...

i	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
T	a	b	d	c	a	b	a	b	d	c	a	b	d	c	b
q	0	1	2	3	4	5	6 2 0	1	2	3	4	5	6		

... $i = 13$ again we skip the **while** loop and increase q and ... ops there is a match! So we run $q \leftarrow \pi[q]$ in the final **if** statement. Therefore q is set to three.

i	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
T	a	b	d	c	a	b	a	b	d	c	a	b	d	c	b
q	0	1	2	3	4	5	6 2 0	1	2	3	4	5	6 7 3	3	

$i = 14$, gives a match, hence q is increase but for $i = 15$, q IS positive and a mismatch occurs, hence we enter the **while** loop and reset q to zero ... and that's the end of it!

i	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
T	a	b	d	c	a	b	a	b	d	c	a	b	d	c	b
q	0	1	2	3	4	5	6 2 0	1	2	3	4	5	6 7 3	3	4 0

Remarks

BRUTE-MATCHING would have performed 23 character-wise comparisons.

If we disregard repeated comparisons (we can always save the result of a comparison in a boolean variable and reuse it!) and we do not take into account the preprocessing to compute the values of π, KMP-MATCHING performs only one comparison in each of its 15 iterations of the main **for** loop plus two more the first time we run the **while** loop and one more the second time. That's 18 in total.

FINITE-AUTOMATON-MATCHING would have been even quicker than KMP-MATCHING, but it would have required longer preprocessing time.