Finite-state automaton

A **finite automaton** is a quintuplet $M = (Q, q_0, A, \Sigma, \delta)$

- Q : finite set of states
- q_0 : initial state
- $A \subset Q$: set of final states
- Σ : finite alphabet
- δ is a function from $Q \times \Sigma$ in Q called the transition function.

The **suffix function** associated with a pattern $P[1..m]$:

$$\begin{array}{cccl} \sigma & : & \Sigma^* & \longrightarrow & \{0, 1, ..., m\} \\ & & t & \longrightarrow & \sigma(x) = \max\{k \mid P[1..k] \text{ suffix of } t\} \end{array} \quad (1)$$

The number $\sigma(t)$ is the size of the largest prefix of the pattern being searched for, which is the suffix of the text t.

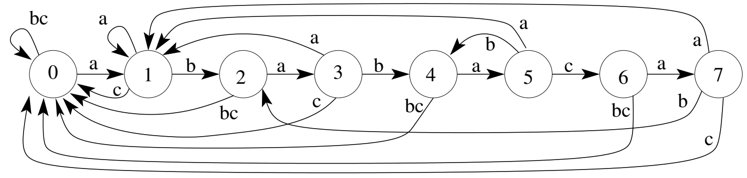
Example : For $P = ab$, one have $\sigma(\epsilon) = 0$, $\sigma(ccaca) = 1$, $\sigma(ccab) = 2$.
If x is suffix of y, $\sigma(x) \leq \sigma(y)$.
b is suffix of ab, $\sigma(b) \leq \sigma(ab)$
a is suffix of aa, $\sigma(a) \leq \sigma(aa)$.

Automaton associated with a pattern. This is the automaton for which we are in state q if and only if the largest prefix we have just read is $P[1..q]$.

- $Q = \{0, 1, ..., m\}$
- $q_0 = \{0\}$
- $A = \{m\}$
- $\delta(q, a) = \sigma(P_q a)$ maximum suffix of the concatenation of P_q with a.

Finite-state automaton

Example : Search for pattern *ababaca*



Once the automaton has been constructed, the text traversal algorithm is as follows :

```
def FiniteAutomatonSearch(T, delta, m)
    n = len(T);
    q = 0 ;
    for (i=1; i<=n; i++) {
      q = delta (q,T[i]);
      if q = m then
        print(''The pattern appears with the offset'', i-m);
    }
```

Complexity : $O(n)$

Run the automaton on the string *ababacaba*.

Computing the transition function.
The idea is based on the meaning of the different states of the automaton : state i corresponds to the state where the first i letters of the searched pattern have just been read. To build the automaton, we go through all the states of the automaton (from 0 to n, where n is the length of the word we're looking for) and for each state i, we go through each letter a of the alphabet. We then calculate the longest prefix of the pattern that is a suffix of $P[1..i].a$. The length of this suffix gives the arrival state of the transition starting from i via letter a.

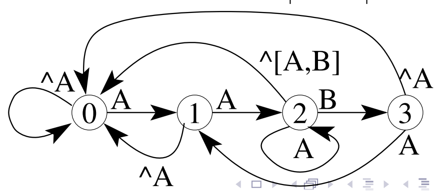
```
1  def Transition_Function_computation (P, Sigma)
2      m = len(P);
3      for q in range(m):
4          for a in Sigma :
5              k = min(m,q+1);
6              while (P[1..k] is not a suffixe of P_q.a ) :
7                  k--;
8              delta(q,a) = k;
9      return(delta);
```

For this function to be correct, the following convention must be used : ε is the suffix for all strings.
Complexity analysis :

- lines 6-7 : $O(m^2)$
- line 4 : $O(|\Sigma|)$
- line 3 : $O(m)$
- Global complexity : $O(m^3|\Sigma|)$
- We can do faster.

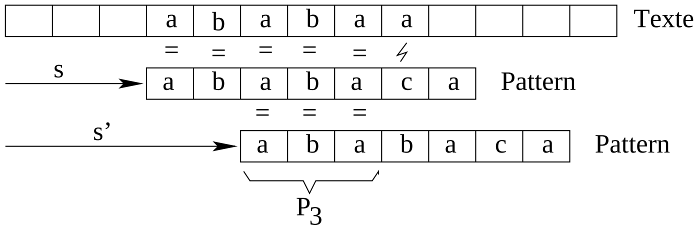
Example : building the automaton for pattern AAB

q	a	k	P_k	$P_q.a$	suffix	δ
q=0	A	$min(m, q+1) = min(3, 1) = 1$	A	A	yes	1
	B	1	A	B	no	
		0	ε	B	yes	0
q=1	A	$min(m, q+1) = min(3, 2) = 2$	AA	AA	yes	2
	B	2	AA	AB	no	
		1	A	AB	no	
		0	ε	AB	yes	0
q=2	A	$min(m, q+1) = min(3, 3) = 3$	AAB	AAA	no	
		2	AA	AAA	yes	2
	B	3	AAB	AAB	yes	3
q=3	A	$min(m, q+1) = min(3, 4) = 3$	AAB	AABA	no	
		2	AA	AABA	no	
		1	A	AABA	yes	1
	B	3	AAB	AABB	no	
		2	AA	AABB	no	
		1	A	AABB	no	
		0	ε	AABB	yes	0



Knuth-Morris-Pratt algorithm

This algorithm achieves complexity in $\Theta(n+m)$ by avoiding the transition function δ. It computes an auxiliar function $\pi[1..m]$ precomputed from the pattern in $O(m)$. The array π allows the transition function δ to be computed on the fly if necessary.
Pattern prefix function : Correspondence between the motif and its own shifts.



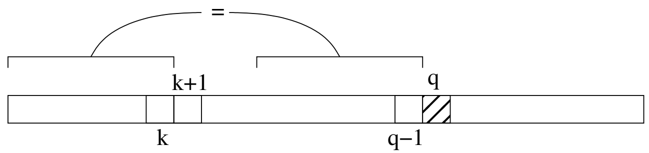
Question : how to calculate s' so that the offset is not invalid ?
Answer : Find a suffix P_k of P_q that is a prefix of P.

The prefix function for the P pattern :
$\Pi \;:\; \{1,2...m\} \longrightarrow \{0,1...m-1\}$
$q \longrightarrow \Pi[q] = max\{k/k < q \text{ and } P_k \text{ is suffix of } P_q\}$
$\Pi[q]$ is in fact the size of the longest prefix of P which is a proper suffix of P_q.

Intuition for calculating the prefix function

Algorithmics for Biology
Jean-Paul Comet
Complexity
Pat. Matching
Rabin-Karp
Automata
KMP
BM
Graphs
Dyn.Prog.
Sequences

- We construct the array Π starting from index 0. The initialisation is simple : $\Pi[1] = 0$
- Now let's assume that we have calculated $\Pi[i]$ from $i = 1$ to $q-1$. To calculate $\Pi[q]$ we have the following situation :



- Since k is the longest prefix that is a suffix of P_{q-1}, the longest prefix that is also a suffix of P_q cannot be longer than P_{k+1}. Furthermore, we have

$$P[k+1] = P[q] \Longleftrightarrow \Pi[q] = k+1$$

- If $P[k+1] \neq P[q]$, look for the largest prefix-suffix of P_q. If we don't look at the last letter, the largest prefix-suffix of P_q is also a suffix of P_k. Now we know the largest prefix-suffix of P_k, which is $\Pi[k]$, already constructed. Once we have $P_{\Pi[k]}$, we need to check if it can be extended to the next letter.

Algorithm for calculating the prefix function

Algorithmics for Biology
Jean-Paul Comet
Complexity
Pat. Matching
Rabin-Karp
Automata
KMP
BM
Graphs
Dyn.Prog.
Sequences

```
Compute_prefix_Function(P)
   m = long(P);
   pi[1] = 0;
   k = 0;
   for (q=2; q<=m ; q++) {
     while (k>0) and P[k+1] != P[q]:
     {
       k=pi[k];
     }
     if P[k+1] == P[q] then k++;
     pi[q] = k;
   }
   return(pi);
```


Example 1

Algorithmics for Biology
Jean-Paul Comet
Complexity
Pat. Matching
Rabin-Karp
Automata
KMP
BM
Graphs
Dyn.Prog.
Sequences

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

Let's build Π :

Π :	1	2	3	4	5	6	7	8	9	10
	0	0	1	2	3	4	5	6	0	1

- $q=2,\ k=0$ no while (k=0)
  if $P[1] = P[2]$ (a=b) ? no, $k = 0 \Longrightarrow \Pi[2] = 0$
- $q=3,\ k=0$ no while (k=0)
  if $P[1] = P[3]$ (a=a) ? yes, $k = 1 \Longrightarrow \Pi[3] = 1$
- $q=4,\ k=1$ no while (P[k+1]=P[q])
  if $P[2] = P[4]$ (b=b) ? yes, $k = 2 \Longrightarrow \Pi[4] = 2$
- $q=5,\ k=2$ no while (P[k+1]=P[q])
  if $P[3] = P[5]$ (a=a) ? yes, $k = 3 \Longrightarrow \Pi[5] = 3$
- $q=6,\ k=3$ no while (P[k+1]=P[q])
  if $P[4] = P[6]$ (b=b) ? yes, $k = 4 \Longrightarrow \Pi[6] = 4$
- $q=7,\ k=4$ no while (P[k+1]=P[q])
  if $P[5] = P[7]$ (a=a) ? yes, $k = 5 \Longrightarrow \Pi[7] = 5$
- $q=8,\ k=5$ no while (P[k+1]=P[q])
  if $P[6] = P[8]$ (b=b) ? yes, $k = 6 \Longrightarrow \Pi[8] = 6$
- $q=9,\ k=6$ enter into the while
  while $P[7] \neq P[9]$ $(a \neq c)$ $k = \Pi[6] = 4$
  while $P[5] \neq P[9]$ $(a \neq c)$ $k = \Pi[4] = 2$
  while $P[3] \neq P[9]$ $(a \neq c)$ $k = \Pi[2] = 0$
  if $P[1] = P[9]$ (a=c) ? no, $k = 0 \Longrightarrow \Pi[9] = 0$
- $q=10,\ k=0$ no while (k=0 & P[k+1]=P[q])
  if $P[1] = P[10]$ (a=a) ? yes, $k = 1 \Longrightarrow \Pi[10] = 1$

Example 2

Consider the following pattern :

1	2	3	4
S	N	N	S

Let's build the function Π :

Π :	1	2	3	4
	0	0	0	1

- $q = 2,\ k = 0$ no while (k=0)
  if $P[1] = P[2]$ $(S \neq N)$? no $\Longrightarrow \Pi[2] = 0$
- $q = 3,\ k = 0$ no while (k=0)
  if $P[1] = P[3]$ $(S \neq N)$? no $\Longrightarrow \Pi[3] = 0$
- $q = 4,\ k = 0$ no while (k=0)
  if $P[1] = P[4]$ $(S = S)$? yes, $k = 1 \Longrightarrow \Pi[4] = 1$

CÔTE D'AZUR
Algorithmics for Biology
Jean-Paul Comet
Complexity
Pat. Matching
Rabin-Karp
Automata
KMP
BM
Graphs
Dyn.Prog.
Sequences

Validity of the prefix function

Let's consider $\pi^*[q] = \{q, \pi[q], \pi^2[q], ...\pi^t[q]\}$ where t is the first natural number such that $\pi^t[q] = 0$.

Lemma

Let P be a pattern of length m and having the prefix function π. Then, for $q = 1, 2, ...m$, one has $\pi^*[q] = \{k \mid P_k \text{ suffix of } P_q\}$

Proof :

1. First inclusion : $i \in \pi^*[q] \Rightarrow P_i$ suffixe de P_q.
   If $i \in \pi^*[q], \exists u \mid \pi^u[q] = i$
   - for $u = 0$, $i = q$ and thus $P_i = P_q$ and P_i is suffix of P_q
   - let us suppose $P_{\pi^u[q]}$ suffix of P_q for each $u < u_0$
     $P_{\pi^{u_0}[q]} = P_{\pi[\pi^{u_0-1}[q]]}$ and one have $P_{\pi^{u_0-1}[q]}$ suffix of P_q and $P_{\pi^{u_0}[q]}$ suffix of $P_{\pi^{u_0-1}[q]}$.
     Since the suffix relationship is transitive, we have $P_{\pi^{u_0}[q]}$ suffix of P_q.
   - Conclusion : $i \in \pi^*[q] \Rightarrow P_i$ suffix of P_q.

□

CÔTE D'AZUR
Algorithmics for Biology
Jean-Paul Comet
Complexity
Pat. Matching
Rabin-Karp
Automata
KMP
BM
Graphs
Dyn.Prog.
Sequences

Validity of the prefix function

Let's consider $\pi^*[q] = \{q, \pi[q], \pi^2[q], ...\pi^t[q]\}$ where t is the first natural number such that $\pi^t[q] = 0$.

Lemma

Let P be a pattern of length m and having the prefix function π. Then, for $q = 1, 2, ...m$, one has $\pi^*[q] = \{k \mid P_k \text{ suffix of } P_q\}$

Proof :

2. let us prove that : $\{k \mid P_k \text{ suffix of } P_q\} \subseteq \pi^*[q]$ Proof by contradiction :
   let us assume that there is an integer in the set $\{k \mid P_k \text{ suffix of } P_q\} \setminus \pi^*[q]$.
   We denote j the largest of the values.
   As $q \in \pi^*[q] \cup \{k \mid P_k \text{ suffix of } P_q\} \Rightarrow j < q$.
   Let j' be the smallest integer in $\pi^*[q]$ that is greater than j.
   - P_j is a suffix of P_q since it belongs to $\{k \mid P_k \text{ suffix of } P_q\}$
   - $P_{j'}$ is a suffix of P_q since it belongs to $\pi^*[q]$

   Thus P_j suffix of $P_{j'}$ (trivial because $j' > j$)
   Moreover j is the largest value of $\{k \mid P_k \text{ suffixe de } P_q\} \setminus \pi^*[q]$
   We should then have $\pi[j'] = j$ and then $j \in \pi^*[q]$.
   Contradiction.

□

CÔTE D'AZUR
Algorithmics for Biology
Jean-Paul Comet
Complexity
Pat. Matching
Rabin-Karp
Automata
KMP
BM
Graphs
Dyn.Prog.
Sequences

Validity of the prefix function

Lemma

Let P be a pattern of length m and prefix function π. For $q = 1, 2, ..., m$,
if $\pi[q] > 0$ then $\pi[q] - 1 \in \pi^*[q-1]$.

Proof :
If $k = \pi[q] > 0$ then P_k suffix of P_q.
Thus P_{k-1} suffix of P_{q-1} (by deleting the last character of P_k and P_q)
According to the previous lemma : $k - 1 \in \pi^*[q-1]$. □

For $q = 2, 3, ..., m$, we define the subset $E_{q-1} \subseteq \pi^*[q-1]$ by :

$$E_{q-1} = \{k \mid k \in \pi^*[q-1] \text{ by } P[k+1] = P[q]\}$$

Intuitively, E_{q-1} is made up of values $k \in \pi^*[q-1]$ such that it is possible to extend P_k to P_{k+1} and obtain a suffix of P_q.

Validity of the prefix function

corrolary

Let P be a pattern of length m and prefix function pi. For $q = 2, 3, \ldots, m$,

$$\pi[q] = 0 \text{ if } E_{q-1} = \{\}$$
$$\pi[q] = 1 + \max\{k \in E_{q-1}\} \text{ if } E_{q-1} \neq \{\}$$

Proof :
If $r = \pi[q] > 0$ then P_r suffix of P_q.
And $r \geq 1 \Rightarrow P[r] = P[q]$
According to the previous lemma, if $r \geq 1$, we have :

$$r = 1 + \max\underbrace{\{k \in \pi^*[q-1] \mid P[k+1] = P_q]\}}_{E_{q-1}}$$

If $r = 0$, there is no $k \in \pi^*[q-1]$ for which we can extend P_k to P_{k+1} to obtain a suffix of P_q, since we would then have $\pi[q] > 0$.
Thus $E_{q-1} = \{\}$ □

CÔTE D'AZUR
Algorithmics for Biology
Jean-Paul Comet
Complexity
Pat. Matching
Rabin-Karp
Automata
KMP
BM
Graphs
Dyn.Prog.
Sequences

Algorithm validity :

1. $\pi[1] = 0$ correct because $\pi[q] < q$ for all q
2. At the start of each loop iteration, we have $k = \pi[q-1]$
   - for the first loop : imposed by $\pi[1] = 0$ and $k = 0$
   - for the others : imposed by $\pi[q] = k$
3. `while` loop : we run through all the values of $\pi^*[q-1]$ until we find one for which $P[k+1] = P[q]$.
   At this point, we know that k is the largest value of E_{q-1} ; and from the corollary, we can give to $\pi[q]$ the value $k+1$
   If no such k is found, $k = 0$
   □

CÔTE D'AZUR
Algorithmics for Biology
Jean-Paul Comet
Complexity
Pat. Matching
Rabin-Karp
Automata
KMP
BM
Graphs
Dyn.Prog.
Sequences

Global procedure

```
KMP1(T,P)
  n = long[T];
  m = long[P];
  PI = Calcul_fonct_prefixe(P);
  q = 0 ;

  Pour i=1 à n faire
    tant que q>0 & P[q+1]!=T[i]
        q = PI[q];
    si P[q+1]=T[i]:
        q = q+1
    si q = m alors
        print(''hit at '', i-m);
        q = PI[q];
```

```
KMP2(T,P)
  n = long[T];
  m = long[P];
  PI = Compute_prefix_Function(P)
        ;
  i = q = 0 ;

  while (i<n):
    if T[i] == P[q]:
      i++; q++;}
    else
      if q==0 :
        i++;
      else:
        q = PI[q-1];
    if q==m
      print(''hit at '', i-m);
      q = PI[q-1];
```

The first version is based on the same idea as the prefix function.
The second version manages two indices in a single loop : one to indicate progress in the text and another to indicate progress in the pattern.

Complexity analysis : requires amortized analysis...