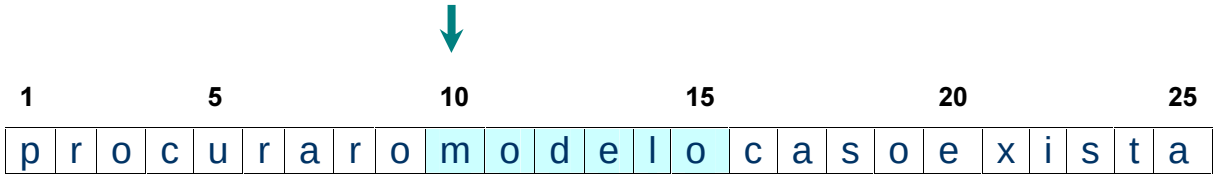
Capítulo 2 : Concordância de Modelos
(pesquisa de padrões) (*pattern matching*)

- Em grande parte dos casos de pesquisa, o que se pretende encontrar não é um elemento simples mas sim uma determinada sequência.

❒ **Problema:** Numa **cadeia** de caracteres, procurar ocorrências **(concordâncias)** de um dado **padrão (modelo)**.



(uma **concordância** ocorre a partir de **i = 10**)

▪ **Aplicações:** Todo o tipo de pesquisas (em estruturas sequenciais):

palavras (ou frases) em textos;
sub-listas em listas;

processadores de texto; motores de busca;
biologia molecular; bibliotecas digitais; criminologia;
filtros de *spam*; imagens digitais; ...

❒ **Especificação:**

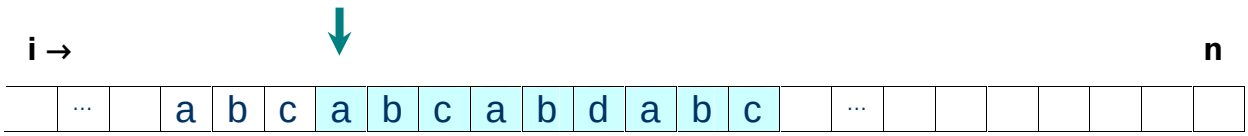
Dados: **cadeia:** s[1..n]
padrão: p[1..m] (onde n >> m)

Resultado: **localização** do início do padrão na cadeia:

$\{ i \in [1..n] : \forall\ k \in [1..m] \Rightarrow s[i + k - 1] = p[k] \}$

❒ Solução Trivial:

Ir deslocando o modelo ao longo da cadeia, comparando caracter a caracter.



k → m

a b c a b d a b c	(falhou para **k = 6**)
a b c a b d a b c	(falhou para **k = 1**)
a b c a b d a b c	(falhou para **k = 1**)
a b c a b d a b c	(**concordância**)

▪ Complexidade da Solução Trivial:

- No **pior caso** são realizadas **m** × **n** comparações.
- Qual a instância que gera o pior caso?
- Quais são os modelos "difíceis"?
- Necessidade de um algoritmo **linear** em **n**.

{ solução trivial }

```
procedure   PesquisarModelo  ( s, p : cadeia; m, n : integer;
                                  var indice : integer;  var achou : boolean);
var    i, j, k : integer;
       iguais : boolean;

begin  i := 0;
       achou:= false;

       while  not achou and ( i < n–m+1) do
              begin  { não começa em [1..i–1]   e  achou = (começa em i) }
                     { não começa em [1..i] }
                     { avançar modelo na cadeia }
                     i = i + 1;
                     k := 0;
                     iguais := true;

                     while  iguais and ( k < m ) do
                            begin  { concordância até k–1
                                                e iguais = (s[i + k – 1] = p[k]) }
                                   { concordância até k }
                                   { avançar no modelo }
                                   k := k + 1 ;
                                   iguais := s[i + k –1] = p[k]
                                   { concordância até k–1
                                                e iguais = (s[i + k – 1] = p[k]) }
                            end;
                     { não iguais ou (k ≥ m) } { iguais = (s[i + k – 1] = p[k]) }

                     achou := iguais
                     { não começa em [1..i–1] e achou = (começa em i) }
              end;
       { achou ou (i ≥ n–m+1) } { achou = (começa em i) }

       { achou = ( ∃ i ∈ [1..n] : ∀ k ∈ [1..m] ⇒ s[i + k – 1] = p[k])  }

       if achou
       then indice := i    { indice ∈ [1..n] : ∀ k ∈ [1..m] ⇒ s[indice + k – 1] = p[k]) }
       else indice := 0    { indice ∉ [1..n] }

end; { PesquisarModelo }
```

❒ Algoritmo de Knuth – Morris – Pratt (*KMP*) (1976)

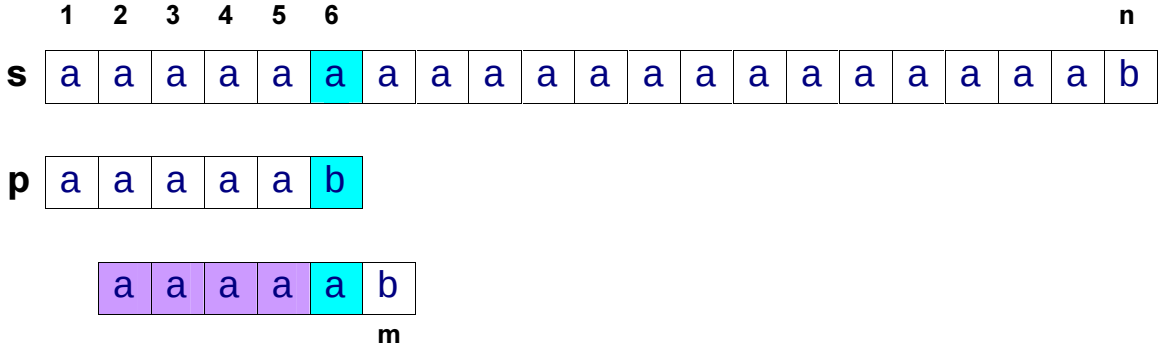

Don Knuth


Jim Morris


Vaughan Pratt

■ A ideia:

- Analisemos um exemplo do pior caso:



- Seriam efectuadas (n−m+1) × m comparações.

  Após a comparação s[6] ≠ p[6],
  o algoritmo trivial avançaria o padrão e voltaria a comparar:
  s[2] = p[1]; s[3] = p[2]; s[4] = p[3]; s[5] = p[4]; s[6] = p[5]; ...

  Deveria comparar apenas s[6] = p[5] !

- Pretendemos um algoritmo onde concordâncias anteriores não voltem a ser testadas. Só assim poderá ser linear em n.

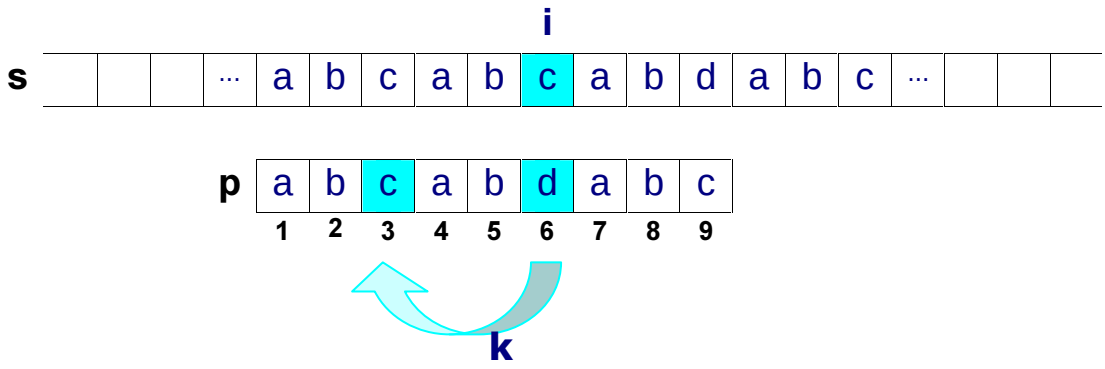
- Os casos "maus" são provocados por concordâncias parciais.

- As concordâncias parciais são provocadas por repetições dentro do modelo.

- Pretendemos um algoritmo que "conheça" as repetições existentes no modelo.

- Como registar as repetições?

- Voltando ao primeiro exemplo:



Após a comparação s[i] ≠ p[6] o algoritmo deveria comparar s[i] = p[3]

Porque no padrão existem as repetições p[1] = p[4] e p[2] = p[5]

- As **Repetições** em p vão ser registadas num vector auxiliar.

p	a	b	c	a	b	d	a	b	c
	1	2	3	4	5	6	7	8	9
R	0	0	0	1	2	0	1	2	3

p[4] = p[1] ⇒ R[4] = 1
p[5] = p[2] ⇒ R[5] = 2
p[6] ≠ p[3] ⇒ R[6] = 0

p[7] = p[1] ⇒ R[7] = 1
p[8] = p[2] ⇒ R[8] = 2
p[9] = p[3] ⇒ R[9] = 3

p	a	a	a	a	a	b

	1	2	3	4	5	6
R	0	1	2	3	4	0

p[5] = p[4] = p[3] = p[2] = p[1]

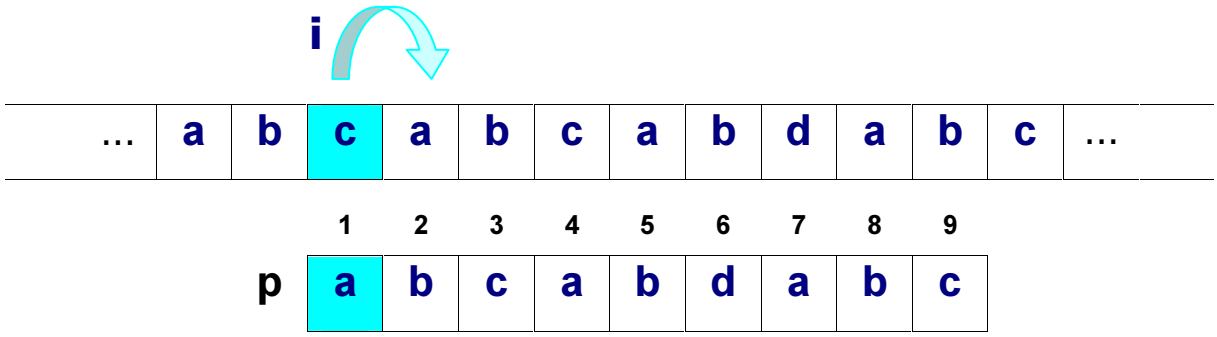
- **Algoritmo para a construção do vector R:**

```
R[1] ← 0
k ← 1
para   i = 2 .. m
       se  p[i] = p [k]
       então   R[i] ← k
               k ← k + 1
       senão   R[i] ← 0
               k ← 1
```

```
procedure   construirR ( p : cadeia; m : integer; var R : vector);
            var  i, k : integer;

            begin  for  i := 1 to m do
                        R[i] := 0;

                   k := 1;
                   for  i := 2 to m do
                        if p[i] = p[k]
                        then  begin  R[i] := k;
                                     k := k + 1
                              end
                        else  k := 1

            end; { construirR }
```

O algoritmo KMP:

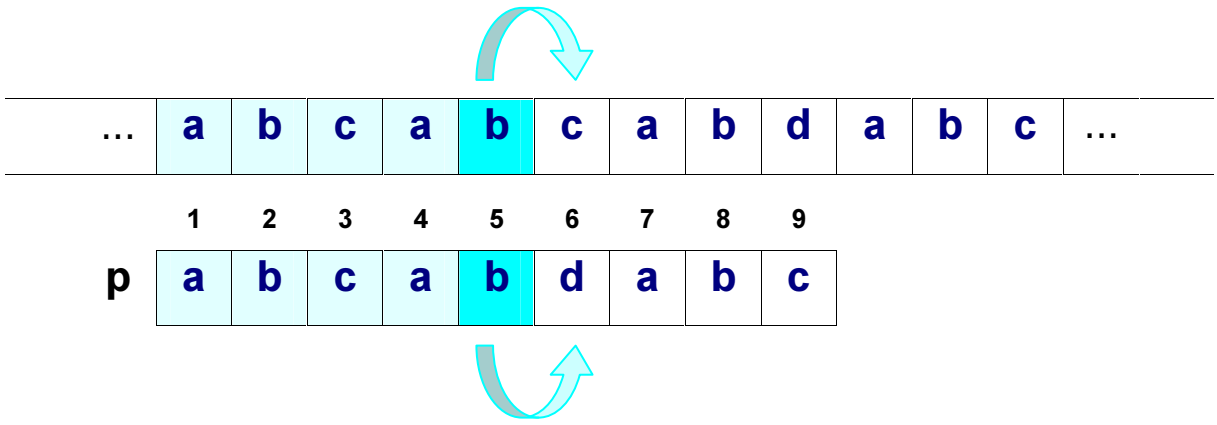
- Se foi verificada uma **desigualdade** para **k = 1** :



o modelo avança na cadeia, continuando k = 1,

```
i ← i + 1
```

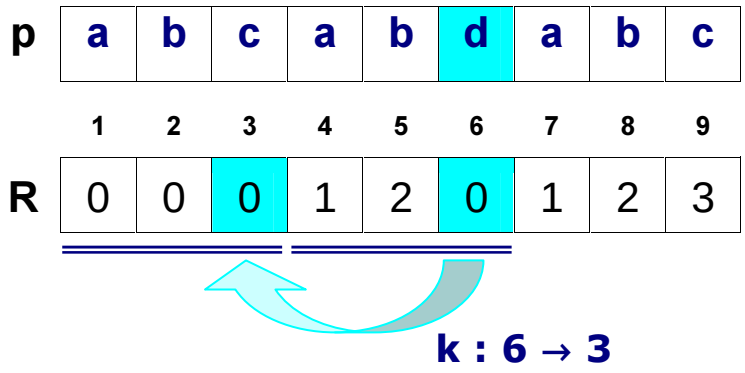
- Como continuar o processo após uma **concordância parcial**?
- Se foi verificada uma **igualdade** s[i] = p[k], como para **k = 5** :



deverão ser comparados os dois caracteres seguintes,

```
i ← i + 1
k ← k + 1
```

- Se foi verificada uma **desigualdade** s[i] ≠ p[k], tal como para **k = 6** :

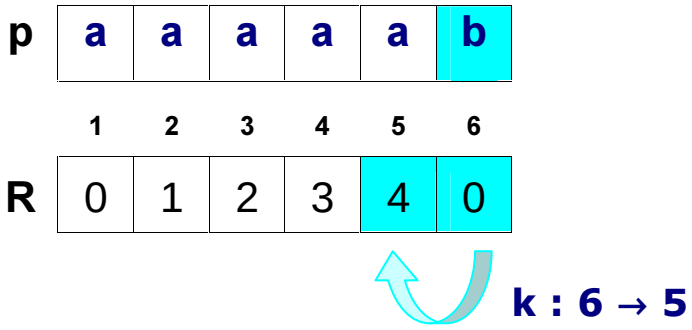


É necessário regressar ao caracter "correspondente" no sub-padrão anterior. Ou seja,

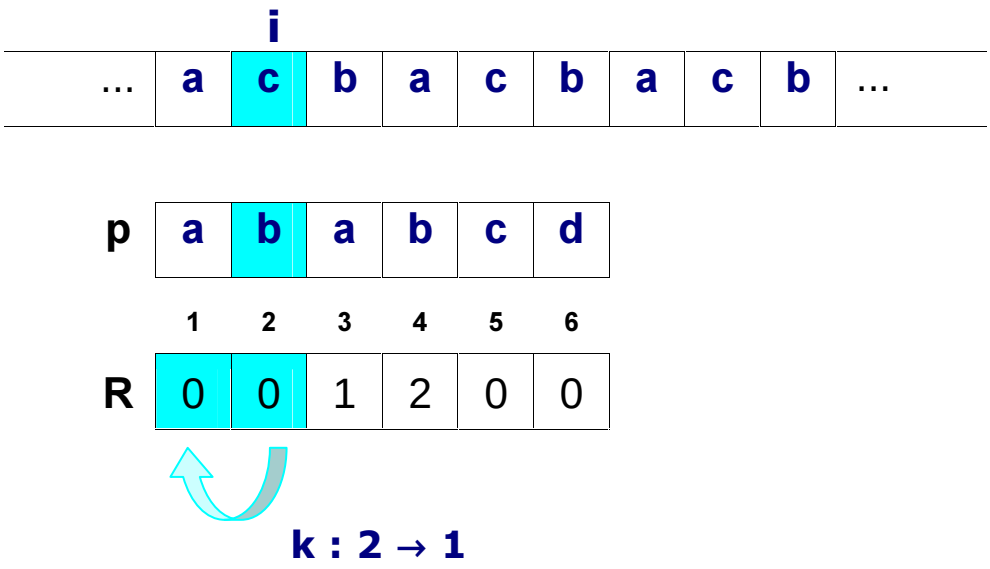
$$k \leftarrow R[k-1] + 1$$

- E processo continuaria, para o mesmo **i**, voltando a comparar s[i] com p[k].

- Para o outro exemplo:



- E mais um exemplo:



s[i] ≠ p[2] : k ← R[k-1]+1 = R[1]+1 = 1

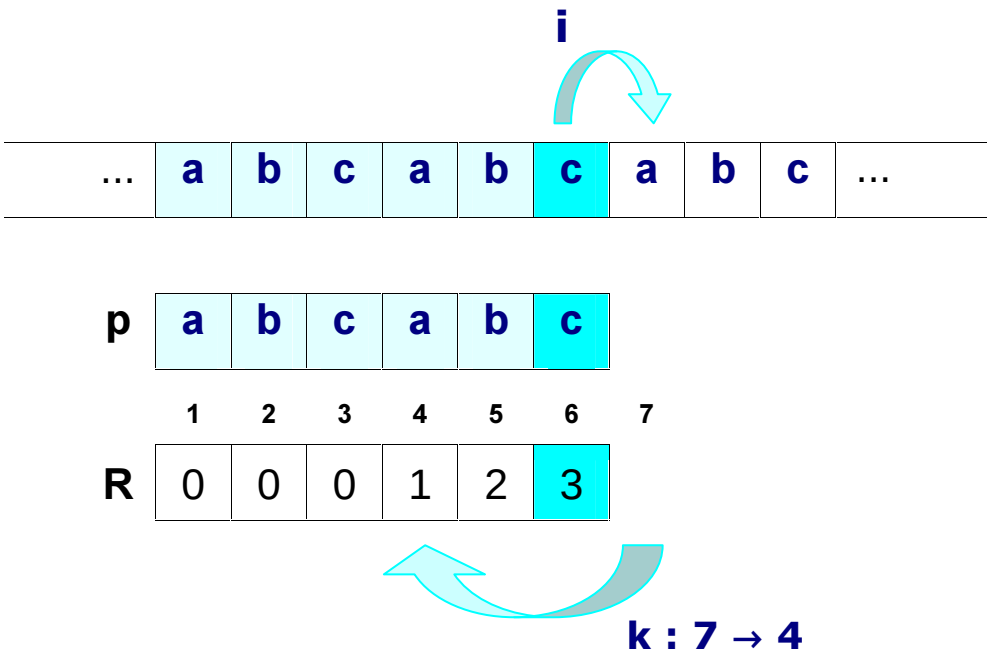
s[i] ≠ p[1] : i ← i + 1

- O **i** avança quando a desigualdade é verificada para k = 1.

- Portanto:

```
se  k = 1
então   i ← i + 1
senão   k ← R[k-1] + 1
```

- E após uma **concordância total**?



s[i] = p[6] : i ← i + 1
k ← k + 1 = 7

k > m : detectada uma concordância.

é necessário recuar no modelo,
utilizando a mesma fórmula.

- Note-se que, neste caso, existem duas concordâncias totais.

- Para uma cadeia de comprimento **n**, o algoritmo KMP nunca efectua mais de duas comparações por elemento. Assim, o número total de comparações varia entre n e 2n, sendo portanto um **algoritmo linear**.

{ Algoritmo KMP }

```
procedure   ContarConcordancias  ( s, p : cadeia; m, n : integer;
                                   var R : vector;  var conc : integer );
var    i, k : integer;

begin  construirR(p, m, R);

        conc := 0;
        i := 1;
        k := 1;

        while  i <= n  do

                if s[i] = p[k]
                then  begin  { concordância parcial }
                             i := i + 1;
                             k := k + 1;

                             if  k > m
                             then { concordância total }
                                  begin  conc := conc + 1;
                                         k := R[k-1] + 1
                                  end
                      end

                else  { desigualdade }
                      if  k = 1
                      then  i := i + 1
                      else  k := R[k-1] + 1

end; { ContarConcordancias }
```

exercício: Adaptar este módulo para a utilização em ficheiros de texto. Por exemplo: procurar uma frase, contar as ocorrências de uma palavra, ...

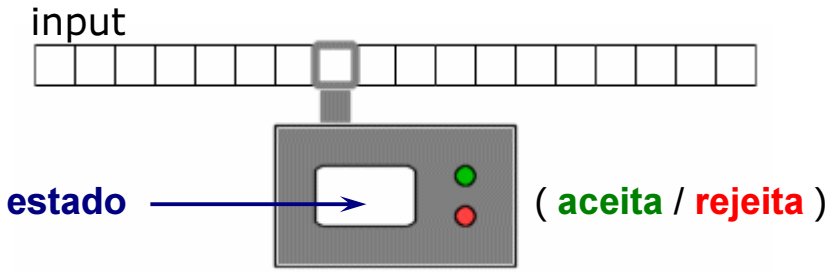
❒ O Modelo Matemático do Algoritmo KMP:

Definição: Um **Autómato Finito Determinista** (**AFD**) é um quíntuplo ordenado,

$$A = (Q, \Sigma, \delta, q_0, F)$$

onde:

- Q é um conjunto finito, não vazio, de **estados**,
- Σ é um conjunto finito (**alfabeto**) de **símbolos de entrada**,
- $\delta : Q \times \Sigma \longrightarrow Q$ é a função (parcial) de **transição** ou de **mudança de estado**,
- $q_0 \in Q$ é o **estado inicial**,
- $F \subseteq Q$ é o conjunto dos **estados finais** ou de **aceitação**.



- Para cada modelo a pesquisar, o algoritmo KMP consiste na **construção** de um AFD e na **simulação** do seu funcionamento ao longo da cadeia (input) de pesquisa.

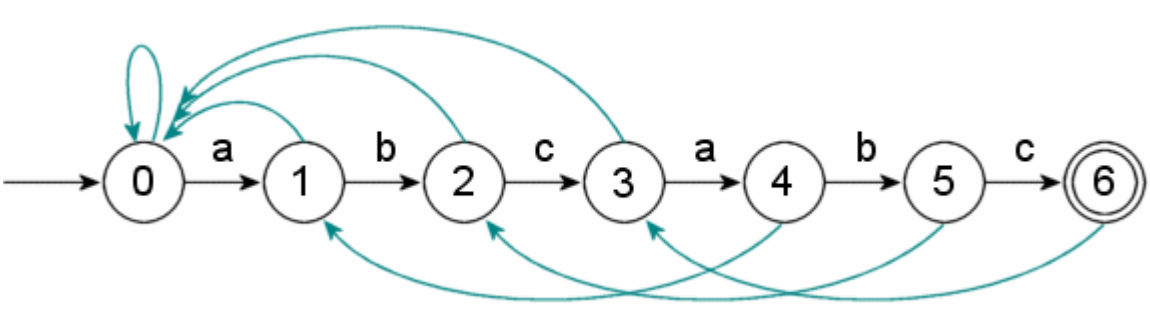
- Um **estado de aceitação** do autómato corresponde a uma **concordância total**.

- A cada modelo corresponde um AFD, cuja **função de transição** foi representada pelo **vector R**.

  Por exemplo:

p	a	b	c	a	b	c
	1	2	3	4	5	6
R	0	0	0	1	2	3

  ou, na representação habitual de autómatos:



  por exemplo, $R[5] = 2 \Leftrightarrow \begin{cases} \delta(q_5, \mathbf{c}) = q_6 \\ \delta(q_5, x) = q_2 \ , \forall\ x \neq \mathbf{c} \end{cases}$

- Partindo do **estado inicial** (0), a leitura da sequência **abcabc** conduz ao **estado de aceitação** ((6)).

- Após uma **desigualdade** (ou uma concordância total) o autómato **regressa** a um dos estados anteriores.

- Esse "regresso" era representado pela fórmula k := R[k-1] + 1

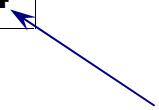
❒ Uma variante do Algoritmo KMP:

- No algoritmo KMP base, o **vector R** é sempre utilizado através da fórmula k := R[k-1] + 1.

- Uma alteração simples (para evitar a subtracção e a soma) consiste em registar num **vector S** o resultado dessa fórmula.

  Por exemplo:

p	a	b	c	a	b	c	
	1	2	3	4	5	6	7
R	0	0	0	1	2	3	
S	1	1	1	1	2	3	4



Utilização de mais um elemento (k = m+1) para o caso da **concordância total**.

```
procedure  construirS ( p : cadeia; m : integer; var S : vector);
           var  i, k : integer;

           begin  S[1] := 1;
                  k := 1;

                  for  i := 2 to m do
                       begin  S[i] := k;
                              if p[i] = p[k]
                              then  k := k + 1
                              else  k := 1
                       end;

                  S[m+1] := k

           end; { construirS }
```

- No exemplo seguinte é utilizada uma **função**.
  A pesquisa é realizada num **ficheiro F** de texto, considerado global à função, e o **vector S** é local.

```
function  concordancias ( p : cadeia; m : integer ) : integer;
var    k, c : integer;
       car : char;
       S : vector;

begin  construirS(p, m, S);

       c := 0;
       k := 1;

       reset(F);
       read(F, car);

       while  not  eof(F)  do

              if car = p[k]
              then  begin  { concordância parcial }
                           read(F, car);
                           k := k + 1;

                           if  k > m
                           then { concordância total }
                                begin  c := c + 1;
                                       k := S[k]
                                end
                    end

              else  { desigualdade }
                    if  k = 1
                    then read(F, car)
                    else  k := S[k];

       close(F);

       concordancias := c

end; { concordancias }
```

❒ Pesquisa numa Lista Linear Ligada:

- Consideremos agora que o domínio de pesquisa é uma lista linear ligada, definida por:

```
type  lista = ^elemento;
      elemento =  record  letra : char;
                          prox : lista
                  end;
```

```
function  concordancias ( ponta : lista; p : cadeia; m : integer ) : integer;
var   k, c : integer;
      S : vector;
      este : lista;

begin  construirS(p, m, S);
       c := 0;
       k := 1;
       este := ponta;

       while  este <> nil do
              if este^.letra = p[k]
              then  begin  { concordância parcial }
                           este := este^.prox;
                           k := k + 1;

                           if  k > m
                           then { concordância total }
                                begin  c := c + 1;
                                       k := S[k]
                                end
                    end
              else  { desigualdade }
                    if  k = 1
                    then  este := este^.prox
                    else  k := S[k];

       concordancias := c

end; { concordancias }
```

❒ Representação do Modelo por uma Lista Ligada:

- Para representar o modelo numa forma "semelhante" a um Autómato Finito Determinista, consideremos a seguinte **lista duplamente ligada**:

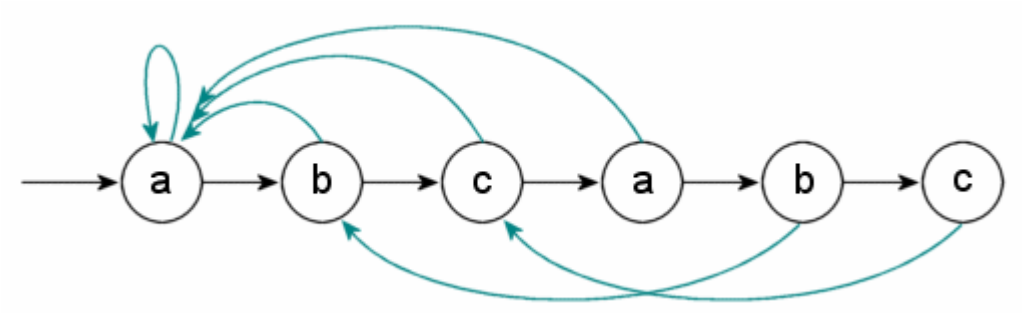
```
type  modelo = ^elemento;
      elemento =  record  letra : char;
                          prox, recua : modelo
                  end;
```

- O ponteiro **prox** é utilizado no caso de uma igualdade de caracteres (k ← k+1) e, no caso de uma desigualdade, o ponteiro **recua** segue as transições definidas pelo **vector S** (k ← S[k]).

  Por exemplo, ao modelo:

p	a	b	c	a	b	c
	1	2	3	4	5	6
S	1	1	1	1	2	3

  corresponde a lista:



- O deslocamento desta estrutura ao longo do domínio de pesquisa, simula o funcionamento de um Autómato Finito Determinista.

- Assumindo que esta representação do **modelo** foi já construída:

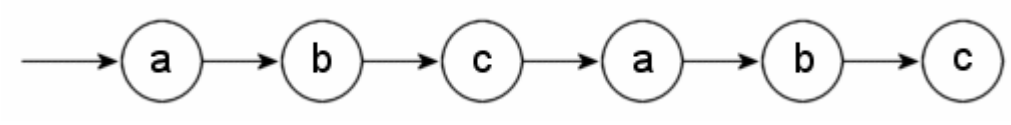
```
function  concordancias ( ponta : lista; inicio : modelo ) : integer;

var   c : integer;
      este : lista;
      p : modelo;

begin  c := 0;
       este := ponta;
       p := inicio;

       while  este <> nil do

              if este^.letra = p^.letra
              then  begin  { concordância parcial }
                           este := este^.prox;

                           if  p^.prox = nil
                           then  { concordância total }
                                 begin  c := c + 1;
                                        p := p^. recua
                                 end
                           else  p := p^. prox
                     end

              else  { desigualdade }
                    if  p = inicio
                    then  este := este^.prox
                    else  p := p^. recua;

       concordancias := c

end; { concordancias }
```

❒ Construção da Representação:

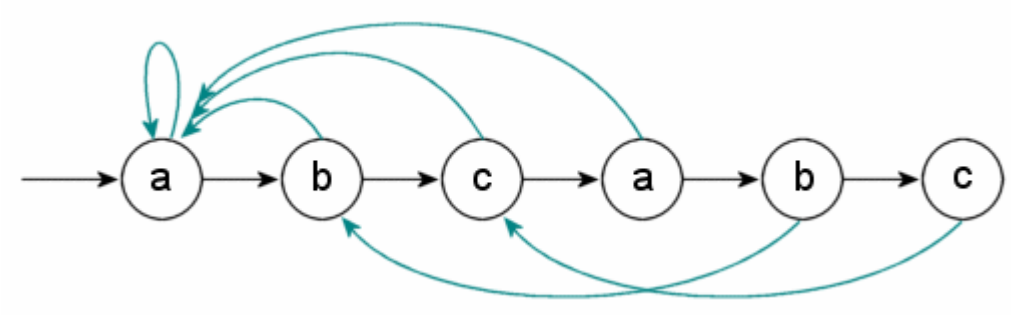
- A partir do padrão inicial e do vector S, começamos por construir a lista e definir os campos **letra** e **prox**:

  Por exemplo:

p	a	b	c	a	b	c
	1	2	3	4	5	6
S	1	1	1	1	2	3

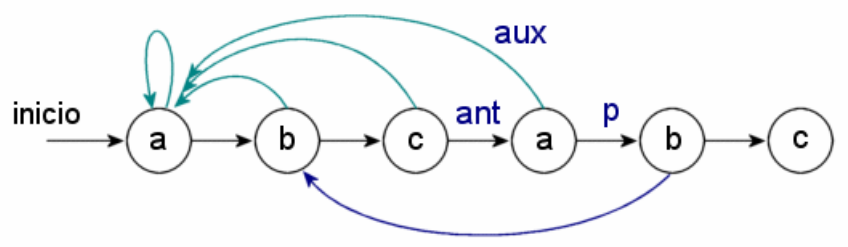


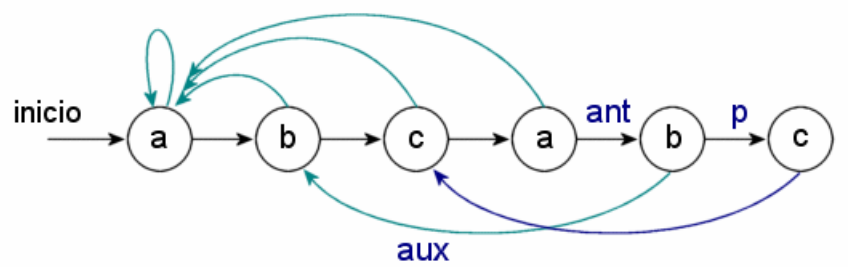
- Procuremos um algoritmo para a definição do campo **recua**:



Algoritmo 1:

```
var p, ant, aux : modelo;
…


p := inicio;
p^.recua := inicio;

while  p^.prox <> nil  do

        begin  ant := p;
               p := p^.prox;
               aux := ant^.recua;

               while  ( aux^.letra <> ant^.letra ) and ( aux <> inicio ) do
                      aux := aux^.recua;

               if ( aux^.letra = ant^.letra ) and (ant <> inicio)
               then  p^.recua := aux^.prox
               else  p^.recua := inicio

        end;
```





Algoritmo 2:

- Com base no algoritmo utilizado para a construção do **vector S**:

```
var  i, k : integer;
...

S[1] := 1;
k := 1;

for  i := 2 to m do
     begin  S[i] := k;
            if p[i] = p[k]
            then  k := k + 1
            else  k := 1
     end;
...
```

e adaptando directamente:

```
var p, aux : modelo;
...

p := inicio;
aux := inicio;
p^.recua := inicio;

while  p^.prox <> nil  do

       begin  p := p^.prox;
              p^.recua := aux;

              if  p^.letra = aux^.letra
              then  aux := aux^.prox
              else  aux := inicio

       end;
```

- Compare a eficiência dos dois algoritmos anteriores, na construção das listas associadas aos modelos:

p	a	b	c	a	b	c	
	1	2	3	4	5	6	7
R	0	0	0	1	2	3	
S	1	1	1	1	2	3	4

p	a	a	a	a	a	b	
	1	2	3	4	5	6	7
R	0	1	2	3	4	0	
S	1	1	2	3	4	5	1

p	a	a	a	a	b	a	
	1	2	3	4	5	6	7
R	0	1	2	3	0	1	
S	1	1	2	3	4	1	2