

CTC-12



Projeto e Análise de Algoritmos

Carlos Alberto Alonso Sanches

CTC-12



6) Busca de padrões

Knuth-Morris-Pratt, Boyer-Moore, Karp-Rabin

Padrões e alfabetos

- Padrões (*patterns* ou *strings*) são sequências de caracteres.
 - Exemplos: documentos, programas, páginas *web*, sequências de DNA, imagens digitalizadas, etc.
- Um alfabeto Σ é um conjunto de possíveis caracteres para uma família de padrões.
 - Exemplos: ASCII, Unicode, {A, C, G, T}, {0, 1}.
- Dados os padrões T e P, de tamanhos n e m, o problema da busca de padrões (*pattern matching*), também chamado de *correspondência de cadeias*, consiste em encontrar subsequências de T iguais a P.
- Sem perda de generalidade, vamos supor $n > m$.

Prefixos e sufixos

- Consideraremos um padrão P de tamanho m como um vetor $P[0..m-1]$ de caracteres.
- Os prefixos de P são as subsequências $P_i = P[0..i]$, onde $0 \leq i < m$.
- Os sufixos de P são as subsequências $S_i = P[i..m-1]$, onde $0 \leq i < m$.
- Seja ε o caractere vazio. Por definição, $P_{-1} = S_m = \varepsilon$.
- Se w é um padrão, então $\varepsilon w = w\varepsilon = w$.

Solução por "força bruta"

- Há dois problemas: encontrar todas as ocorrências de P em T ou apenas a primeira delas. Vamos apresentar algoritmos que encontram somente a primeira ocorrência.
 - Caso se deseje encontrar as demais, bastaria armazenar a posição inicial de cada ocorrência e continuar a busca. Os algoritmos apresentados consideram a possibilidade de sobreposição.
- O algoritmo de busca de padrões através da "força bruta" compara P com T para cada possível deslocamento:
 - até que a primeira ocorrência do padrão P seja encontrada em T ;
 - ou até que todos os possíveis deslocamentos sejam testados.
- No pior caso, (praticamente) todos os caracteres de P serão comparados com todos os caracteres de T .
- Portanto, este algoritmo gasta tempo $O(n.m)$.

Algoritmo "força bruta"

```
int BruteForceMatch() {
    for (i=0; i<=n-m; i++) {
        j = 0;
        while (j<m && T[i+j]==P[j])
            j++;
        if (j == m) return i; // P encontrado em T[i]
    }
    return -1; // P não foi encontrado
}
```

Tempo: $O((n-m).m) = O(n.m)$

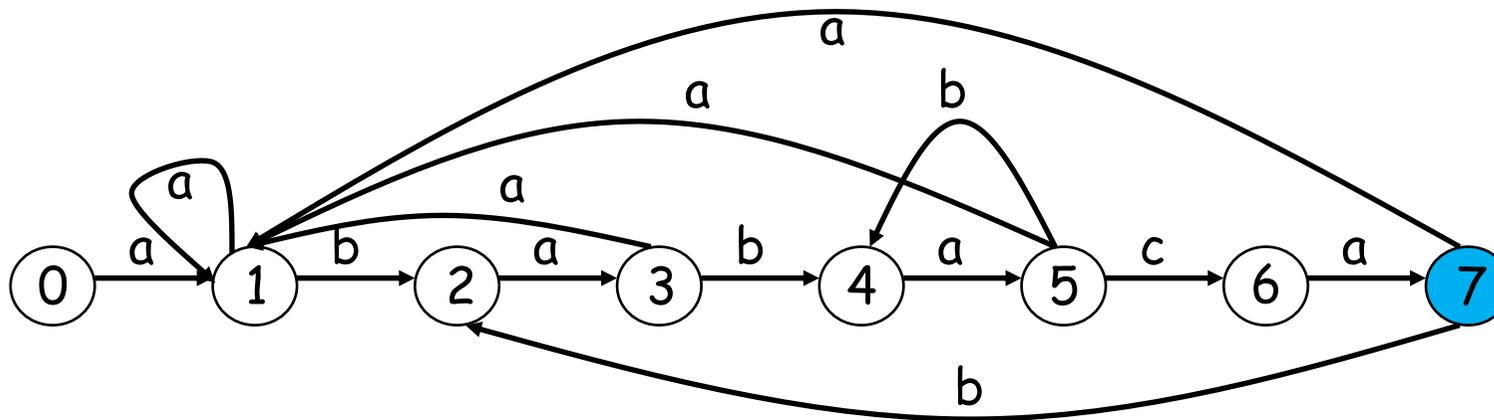
Busca através de autômato

- Um método mais eficiente é montar uma máquina de estados, chamada *autômato finito*.
- Ideias:
 - Cada estado desse autômato significa o reconhecimento de um prefixo do padrão.
 - A leitura de cada caractere de T provoca uma mudança de estado.
 - Há um único estado final, que somente é atingido após a leitura de uma subsequência igual a P .
- O autômato pode ser representado por uma matriz AF de duas dimensões: considerando que esteja no estado s e que o próximo caractere é x , $AF[s,x]$ será o próximo estado.
- A principal dificuldade não é o uso desse autômato, mas a sua construção.

Exemplo

$\Sigma = \{a, b, c\}$

$P = \text{"ababaca"}$



Estado	a	b	c
0	1	0	0
1	1	2	0
2	3	0	0
3	1	4	0
4	5	0	0
5	1	4	6
6	7	0	0
7	1	2	0

As transições não representadas são retornos para o estado 0

$T = \text{"abababacaba"}$

Busca de
P em T:

i	-	0	1	2	3	4	5	6	7	8	9	10
T[i]	-	a	b	a	b	a	b	a	c	a	b	a
Estado	0	1	2	3	4	5	4	5	6	7	2	3

Algoritmo

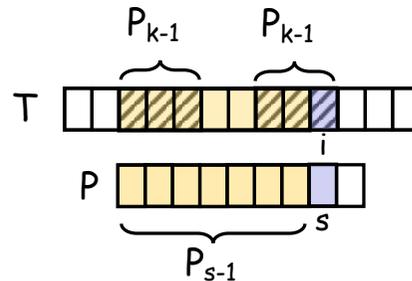
Busca de P em T através de um autômato:

```
int AFMatch() {
    s = 0; // 0 é o estado inicial
    for (i=0; i<n; i++) {
        s = AF[s,T[i]];
        if (s == m) // último estado foi atingido
            return i-m; // posição de P em T
    }
    return -1; // P não foi encontrado
}
```

Tempo: $O(n)$

Construção do autômato

- O que caracteriza o estado s , $0 \leq s \leq m$, é que o prefixo P_{s-1} acabou de ser reconhecido. Portanto, quando o estado final m é atingido, P foi reconhecido completamente.
- Transições quando não ocorre casamento:
 - Se o autômato está no estado s (ou seja, P_{s-1} foi reconhecido) e, após ler o caractere $T[i]$, deve voltar para o maior prefixo ainda válido, será preciso encontrar o maior estado $0 \leq k \leq s$ tal que P_{k-1} seja sufixo de $P_{s-1}T[i]$. Isso pode ser feito através de uma busca exaustiva em P .



- Como $P_{-1} = \varepsilon$ sempre é sufixo de $P_{s-1}T[i]$, haverá volta para o estado inicial quando $k = 0$ for a única solução.
- Por outro lado, transições para estados maiores correspondem aos casos em que P_s é sufixo de $P_{s-1}T[i]$, ou seja, se houver solução para $k = s+1$.

Algoritmo

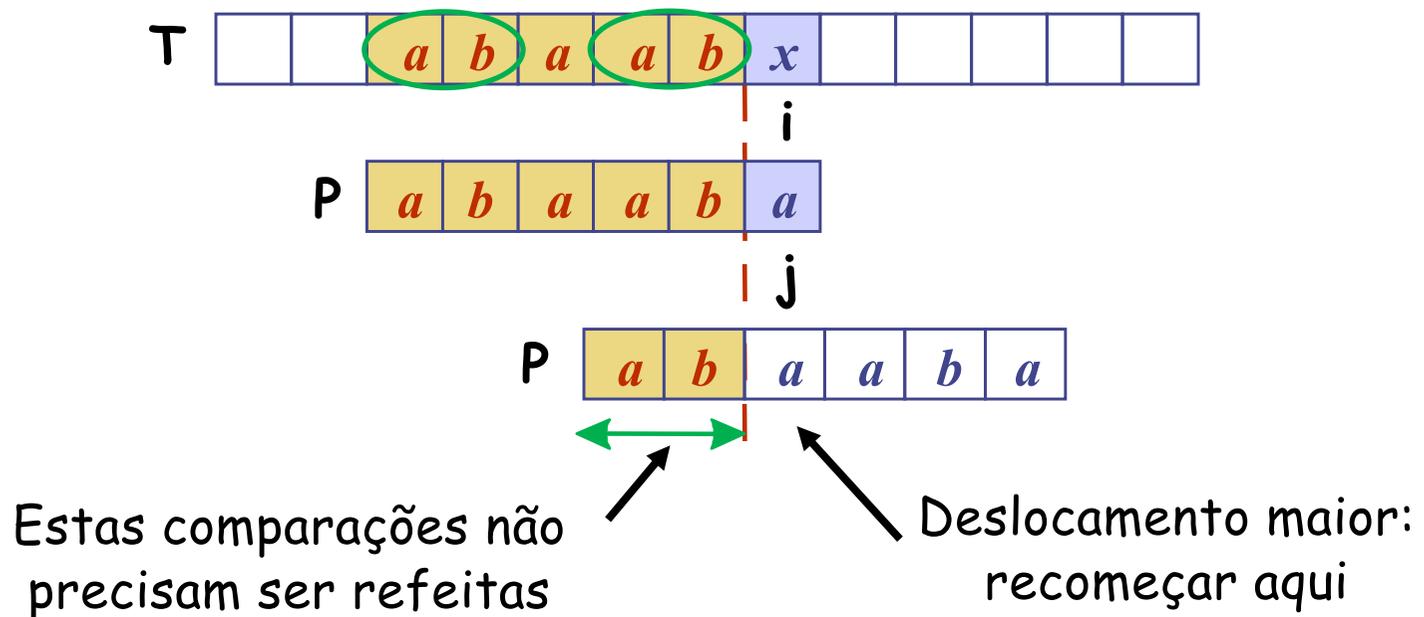
```
AFConstruct() {
  for (s=0; s<=m; s++)
    for x ∈ Σ {
      k = min{s+2, m+1}; // primeiro estado a ser testado
      repeat
        k--; // testes em ordem decrescente
      until (Pk-1 seja sufixo de Ps-1x); // tempo Θ(m)
      AF[s,x] = k;
    }
}
```

Tempo: $O(m^3 \cdot |\Sigma|)$

Pode ser melhorado para $\Theta(m \cdot |\Sigma|)$

Knuth-Morris-Pratt (1970-1976)

- Ideia: considerando o algoritmo "força bruta", quando ocorre uma diferença entre $T[i]$ e $P[j]$, não seria possível fazer um *deslocamento maior* de P para a direita, *evitando comparações redundantes*?
- Exemplo:

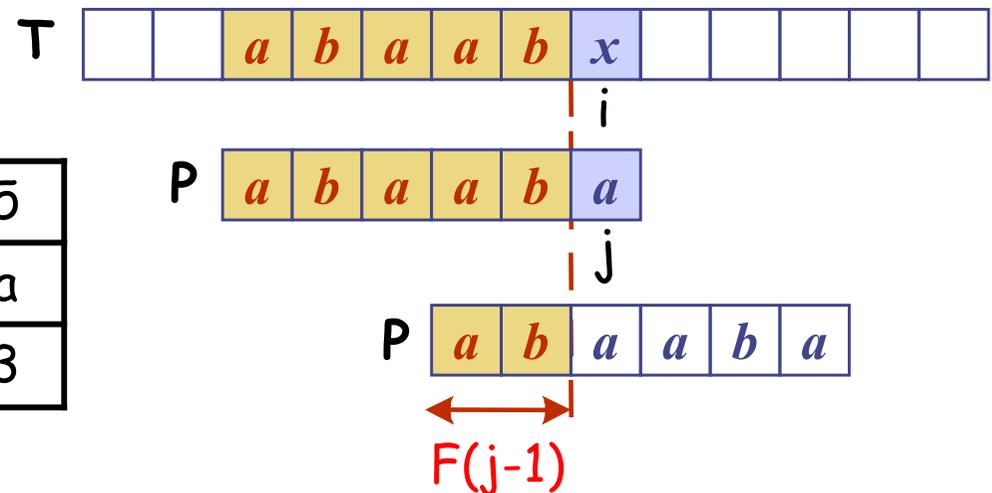


Função de falha

- Pré-processamento em P: determina se seus prefixos aparecem como subsequências dele mesmo.
- A função de falha $F(k)$ será definida como o tamanho do maior prefixo de $P[0..k]$ que é sufixo de $P[1..k]$.
- Informalmente, é o tamanho do maior "começo" de P_k que também aparece no seu "fim", sem considerar ele mesmo.

- Exemplo:

k	0	1	2	3	4	5
P[k]	a	b	a	a	b	a
F(k)	0	0	1	1	2	3



- Se $P[j] \neq T[i]$, então j receberá o valor $F(j-1)$.

Algoritmo KMP

```
int KMPMatch() {
    FailureFunction(); // Veremos que gasta tempo  $\Theta(m)$ 
    i = 0;
    j = 0;
    while (i < n)
        if (T[i] == P[j])
            if (j == m-1)
                return i-j;
            else
                { i++; j++; }
        else
            if (j != 0)
                j = F[j-1];
            else
                i++;
    return -1;
}
```

j é incrementado n vezes no máximo

Como é um decremento, será executado até n vezes

Tempo do laço while: $O(n)$

Exemplo

a b a c a a b a c c a b a c a b a a b b

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

7
a b a c a b

8 9 10 11 12
a b a c a b

13
a b a c a b

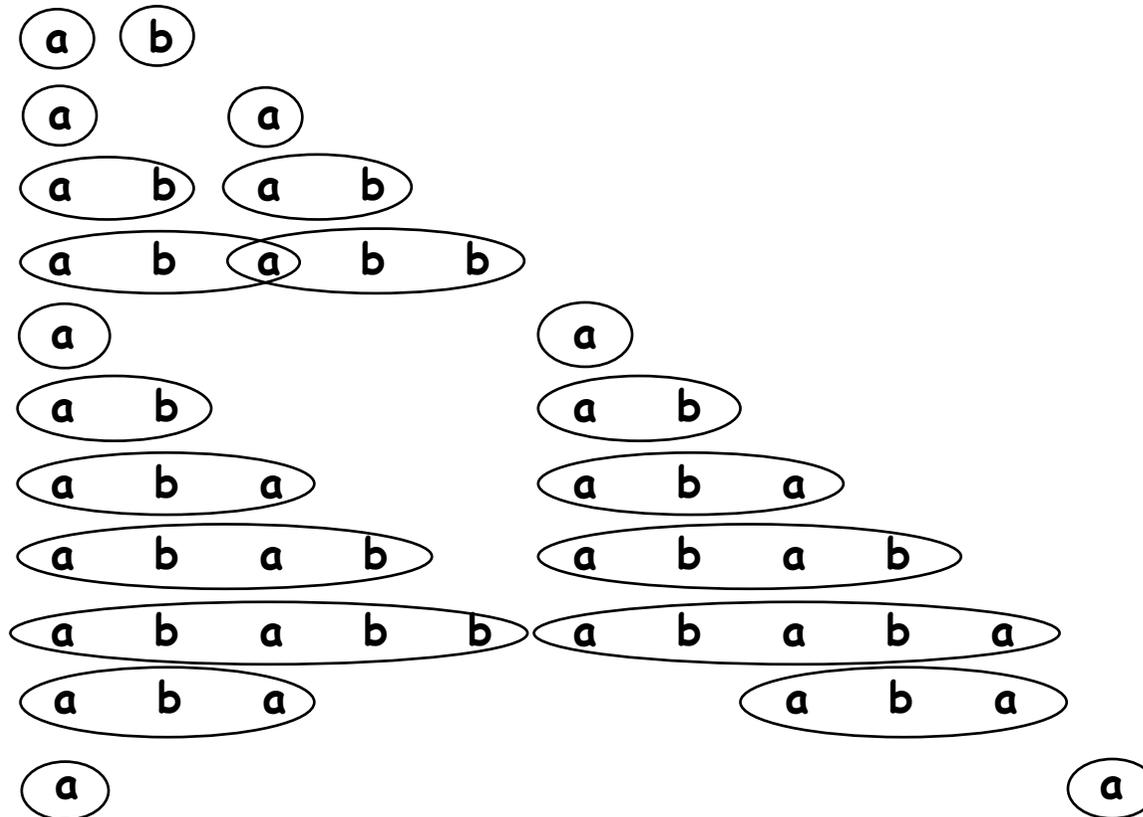
14 15 16 17 18 19
a b a c a b

j	0	1	2	3	4	5
P[j]	a	b	a	c	a	b
F(j)	0	0	1	0	1	2

Exemplo: cálculo da função de falha



j	0	1	2	3	4	5	6	7	8	9	10
$P[j]$	a	b	a	b	b	a	b	a	b	a	a
$F[j]$	0	0	1	2	0	1	2	3	4	3	1



Uso "recursivo" de F

Cálculo da função de falha

Subsequências de P são procuradas dentro dele mesmo:

```
FailureFunction() {
  F[0] = 0;
  j = 0;      // índice que percorre os prefixos
  i = 1;      // índice que percorre os sufixos
  while (i < m)
    if (P[i] == P[j]) // já combinaram j+1 caracteres
      F[i++] = ++j;
    else if (j == 0)
      F[i++] = 0;
    else
      j = F[j-1]; // uso "recursivo" de F
}
```

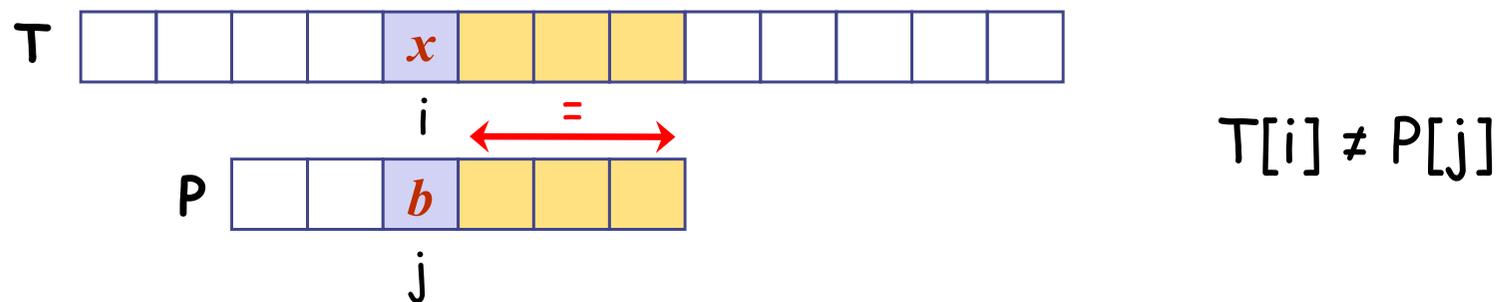
 j é incrementado m-1 vezes no máximo

 Como é um decremento, será executado até m-1 vezes

Tempo: $\Theta(m)$

Boyer-Moore (1976)

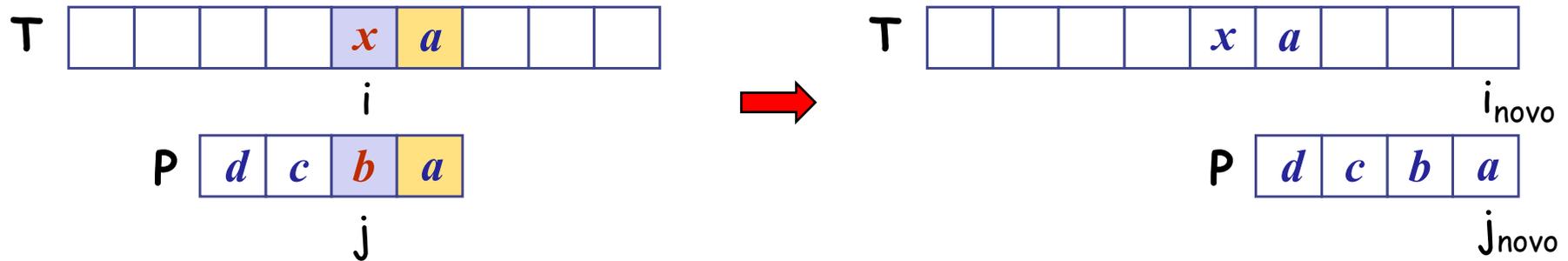
- Ideias do algoritmo de Boyer-Moore:
 - Baseia-se na alta probabilidade de encontrar diferenças quando os padrões são pequenos e os alfabetos grandes.
 - Por isso, P é comparado com T de trás para frente.
 - Quando se encontra uma diferença em $T[i]$, o padrão P dará um salto à frente, considerando-se as comparações já realizadas.



- Será preciso averiguar 3 casos diferentes.

Caso 1

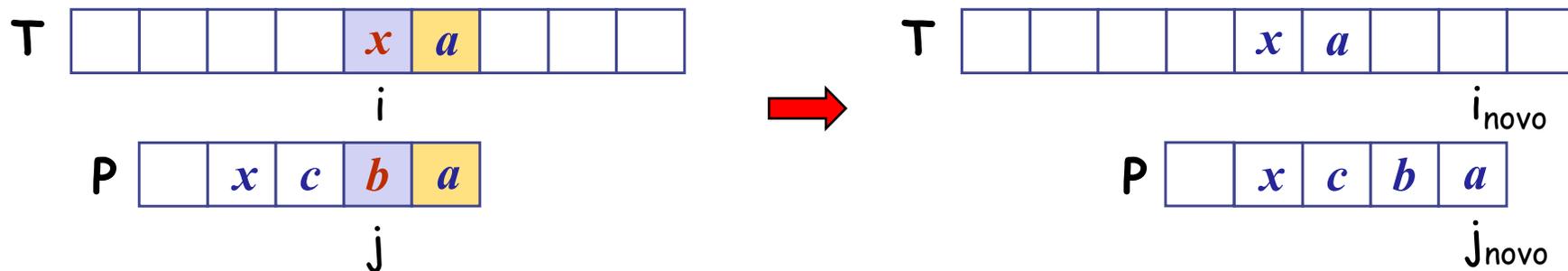
- P não contém x



- Deslocar P para a direita, alinhando P[0] com T[i+1]

Caso 2

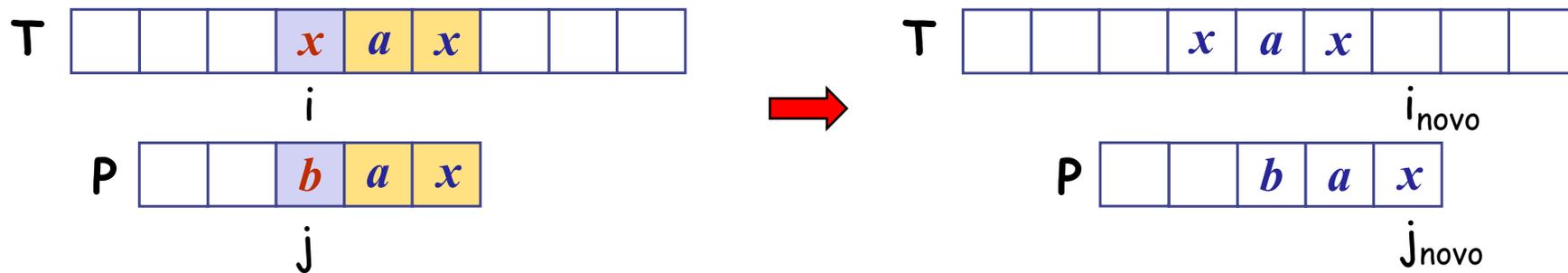
- A última ocorrência de x em P está algum índice menor do que j .



- Deslocar P para a direita, até que a última ocorrência de x fique alinhada com $T[i]$.

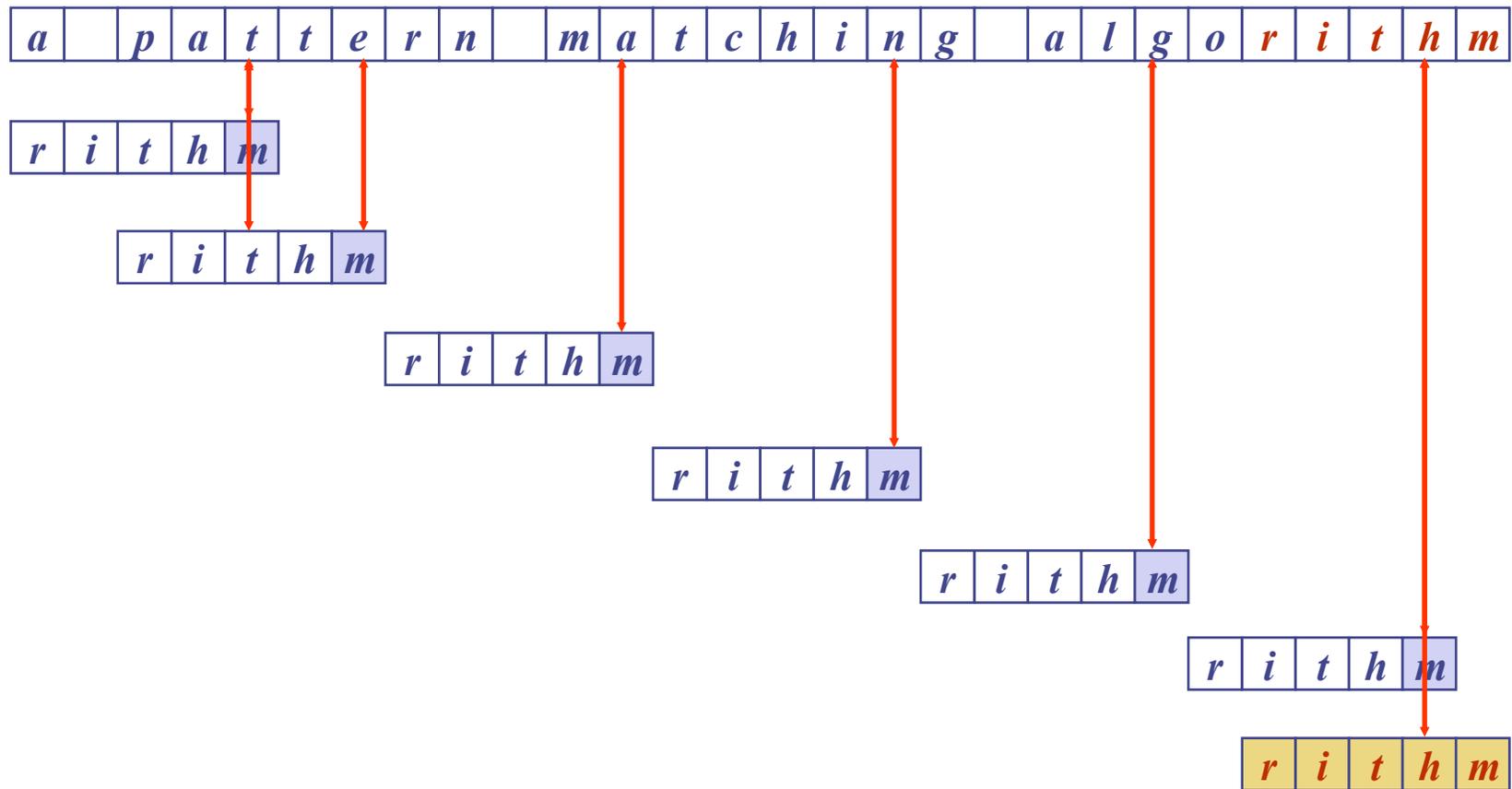
Caso 3

- A última ocorrência de x em P está em algum índice maior do que j .



- Deslocar P apenas uma posição para a direita.

Exemplo



Outro exemplo

<i>a</i>	<i>b</i>	<i>a</i>	<i>c</i>	<i>a</i>	<i>a</i>	<i>b</i>	<i>a</i>	<i>d</i>	<i>c</i>	<i>a</i>	<i>b</i>	<i>a</i>	<i>c</i>	<i>a</i>	<i>b</i>	<i>a</i>	<i>a</i>	<i>b</i>	<i>b</i>
----------	----------	----------	----------	----------	----------	----------	----------	----------	----------	----------	----------	----------	----------	----------	----------	----------	----------	----------	----------

1

<i>a</i>	<i>b</i>	<i>a</i>	<i>c</i>	<i>a</i>	<i>b</i>
----------	----------	----------	----------	----------	----------

4 3 2

<i>a</i>	<i>b</i>	<i>a</i>	<i>c</i>	<i>a</i>	<i>b</i>
----------	----------	----------	----------	----------	----------

5

<i>a</i>	<i>b</i>	<i>a</i>	<i>c</i>	<i>a</i>	<i>b</i>
----------	----------	----------	----------	----------	----------

6

<i>a</i>	<i>b</i>	<i>a</i>	<i>c</i>	<i>a</i>	<i>b</i>
----------	----------	----------	----------	----------	----------

7

<i>a</i>	<i>b</i>	<i>a</i>	<i>c</i>	<i>a</i>	<i>b</i>
----------	----------	----------	----------	----------	----------

13 12 11 10 9 8

<i>a</i>	<i>b</i>	<i>a</i>	<i>c</i>	<i>a</i>	<i>b</i>
----------	----------	----------	----------	----------	----------

Função de última ocorrência

- Através de um pré-processamento, o algoritmo de Boyer-Moore calcula uma função $L: \Sigma \rightarrow \mathbf{I}$, onde $L(x)$ é definida como:

- o maior índice i tal que $P[i] = x$;
- -1, caso este índice não exista.

- Exemplo: $\Sigma = \{a, b, c, d\}$

P

a	b	a	c	a	b
0	1	2	3	4	5

x	a	b	c	d
$L(x)$	4	5	3	-1

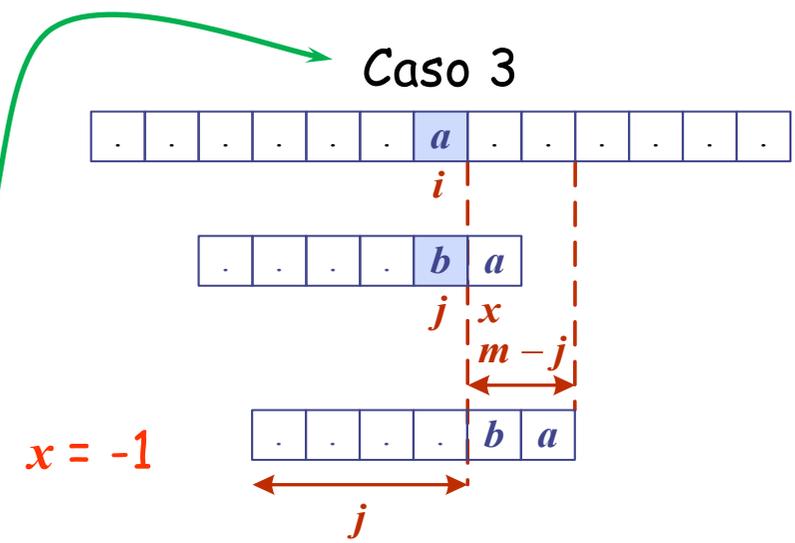
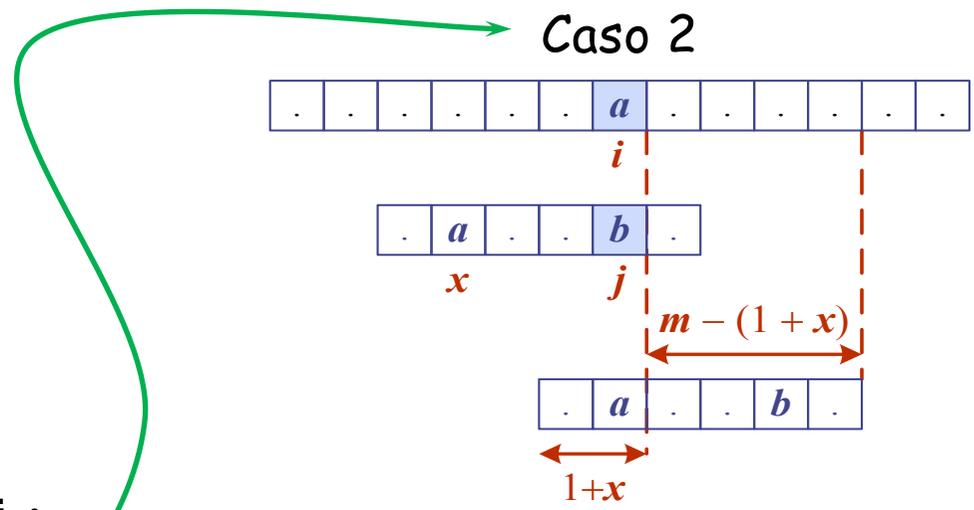
Algoritmo BM

```

int BoyerMooreMatch() {
  for (k=0; k<|Σ|; k++)
    L[k] = -1;
  for (k=0; k<m; k++)
    L[P[k]] = k;
  i = m-1;
  j = m-1;
  repeat
    if (T[i] == P[j])
      if (j == 0) return i;
      else { i--; j--; }
    else {
      x = L[T[i]];
      i += m - min{j, 1+x};
      j = m-1;
    }
  until (i > n-1);
  return -1;
}

```

Caso 1: $i += m$; pois $x = -1$



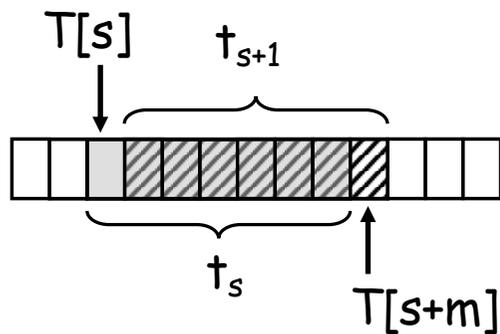
Karp-Rabin (1980)

- Se $\Sigma = \{0, 1\}$ e $m \leq 8$, a busca de P em T poderia ser realizada através de comparações entre *bytes*, que é muito rápida.
- Se $\Sigma = \{0, 1, 2, \dots, 9\}$, P seria um número de m dígitos. Poderíamos então aplicar essa mesma ideia, deslocando uma janela de m dígitos e comparando números inteiros?
- Se P eventualmente não couber em uma variável inteira, poderíamos calcular alguma função de dispersão (*hashing*) para cada subsequência de T com tamanho m e compará-la com o correspondente valor desta função para P :
 - Se forem iguais, um algoritmo de "força bruta" verificaria se o padrão foi mesmo encontrado.
 - Se forem diferentes, o algoritmo continuará a busca, calculando a mesma função para a próxima sequência de m dígitos em T .

Valor em P

- $\Sigma = \{0, 1, 2, \dots, 9\}$
- p : valor do número representado pelos m dígitos de P
- Através da Fórmula de Horner, p pode ser calculado em tempo $\Theta(m)$:
 - $p = (\dots(P[0].10 + P[1]).10 + \dots + P[m-3]).10 + P[m-2]).10 + P[m-1]$
 - Basta um comando `for` com multiplicações e adições.
- Exemplo:
 - $P = "1569"$, $m = 4$
 - $p = ((1.10 + 5)10 + 6)10 + 9$
 - 3 multiplicações e 3 adições

Valores em T



- t_s : valor do número representado pela subsequência $T[s..s+m-1]$, $0 \leq s < n-m$.
- Exemplo:
 - $T = "64152"$, $m = 4$, $s = 0$, $t_s = 6415$, $T[s+m] = 2$
 - $t_{s+1} = 10(6415 - 1000 \cdot 6) + 2 = 4152$
- t_{s+1} pode ser calculado a partir de t_s em tempo $\Theta(1)$:
 - $t_{s+1} = 10(t_s - 10^{m-1}T[s]) + T[s+m]$
 - A subtração de $10^{m-1}T[s]$ remove o dígito de mais alta ordem
 - $T[s+m]$ será o novo dígito de mais baixa ordem

Complexidade dos cálculos

- p e t_0 podem ser calculados em tempo $\Theta(m)$.
- t_{s+1} pode ser calculado a partir de t_s em tempo $\Theta(1)$.
- t_1, \dots, t_{n-m} podem ser calculados tempo $\Theta(n-m)$.
- Portanto, todas as ocorrências de P em T podem ser encontradas em tempo $\Theta(n)$.
- No entanto, para que as comparações entre p e cada t_s sejam feitas em tempo constante, esses números devem estar limitados ao valor máximo de um inteiro suportado pelo sistema (depende da quantidade de *bytes* utilizados).
- Como vimos, esse eventual problema pode ser resolvido com o uso de uma função de dispersão (*hashing*).

Uma solução

- *Função de dispersão* : todos os valores $(p, t_0, t_1, \dots, t_{n-m})$ serão calculados em módulo q , onde q é um número primo.
- Definindo $d = |\Sigma|$, q costuma ser escolhido de tal modo que o valor $d \cdot q$ possa ser armazenado em um número inteiro.
- Os cálculos passam a ser:
 - $t_{s+1} = (d(t_s - T[s] \cdot h) + T[s+m]) \bmod q$, onde $h = d^{m-1} \bmod q$.
- Evidentemente, $t_s \equiv p \bmod q$ não significa necessariamente que $P = T[s..s+m]$.
- Heurística:
 - Se $t_s \equiv p \bmod q$, verificar por "força bruta" se $P = T[s..s+m]$.
 - Caso contrário, continuar a busca.

Exemplo

- $t_{s+1} = (d(t_s - T[s].h) + T[s+m]) \bmod q$, onde $h = d^{m-1} \bmod q$.
- Sejam: $d=10$, $T="31526"$, $n=5$, $P="26"$, $m=2$, $q=11$.
- $p = 26 \bmod 11 = 4$
- $h = 10^1 \bmod 11 = 10$
- $t_0 = 31 \bmod 11 = 9$
- $t_1 = (10(9 - 3 \cdot 10) + 5) \bmod 11 = -205 \bmod 11 = 4$
 - Conferindo: $t_1 = 15 \bmod 11 = 4$
 - $t_1 \equiv p \bmod 11$, mas $T[1..2] = "15"$ e $P = "26"$: P não foi encontrado...
- $t_2 = (10(4 - 1 \cdot 10) + 2) \bmod 11 = -58 \bmod 11 = 8$
 - Conferindo: $t_2 = 52 \bmod 11 = 8$
- $t_3 = (10(8 - 5 \cdot 10) + 6) \bmod 11 = -414 \bmod 11 = 4$
 - $t_3 \equiv p \bmod 11$, e P realmente é encontrado em T

Algoritmo KR

```
int KarpRabinMatch() {
    d =  $|\Sigma|$ ;
    q = um primo maior que m;
    h =  $d^{m-1} \bmod q$ ;
    p = 0;
    t0 = 0;
    for (i=0; i<m; i++) { // cálculo de p e de t0
        p = (d.p + P[i]) mod q;
        t0 = (d.t0 + T[i]) mod q;
    }
    for (s=0; s<=n-m; s++) { // cálculo de t1, ..., tn-m
        if (p == ts) // fazer comparação "força bruta"
            if (P[1..m] == T[s..s+m]) return s;
        if (s < n-m) ts+1 = (d.(ts-T[s].h)+T[s+m]) mod q;
    }
    return -1;
}
```

Comentários

- Os cálculos de p e t_0 gastam tempo $\Theta(m)$.
- Os cálculos de t_1, \dots, t_{n-m} , mais a eventual comparação por "força bruta", gastam tempo $O((n-m).m)$.
- Portanto, o tempo total do algoritmo de Rabin-Karp pode chegar a $O(n.m)$.
- Na prática, este algoritmo tem bom desempenho.
- Importante: ele é válido para qualquer alfabeto! Basta interpretar cada caractere como um dígito...

Comparações

<i>Algoritmos</i>	<i>Pré-processamento</i>	<i>Tempo de busca</i>
"Força bruta"	-	$O(n.m), \Omega(n)$
Autômato finito	$\Theta(m \cdot \Sigma)$	$\Theta(n)$
KMP	$\Theta(m)$	$\Theta(n)$
Boyer-Moore	$\Theta(m + \Sigma)$	$O(n.m), \Omega(n/m)$
Karp-Rabin	-	$O(n.m), \Omega(n)$

- A tabela acima apresenta as complexidades de tempo na *busca de todas as ocorrências do padrão*.
- Teoricamente, KMP é o melhor.
- Na prática, Boyer-Moore é o mais usado. Além disso, possui algumas variações na literatura.