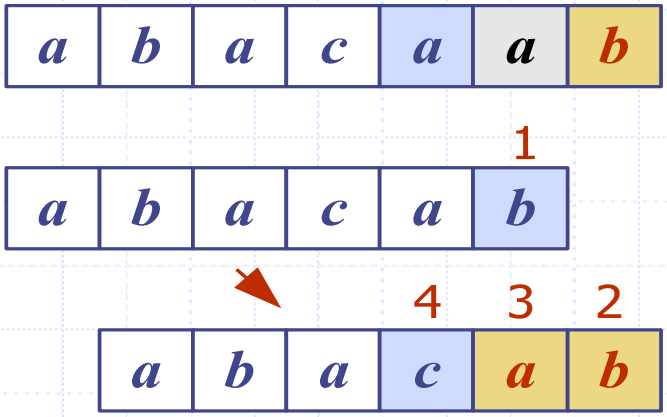
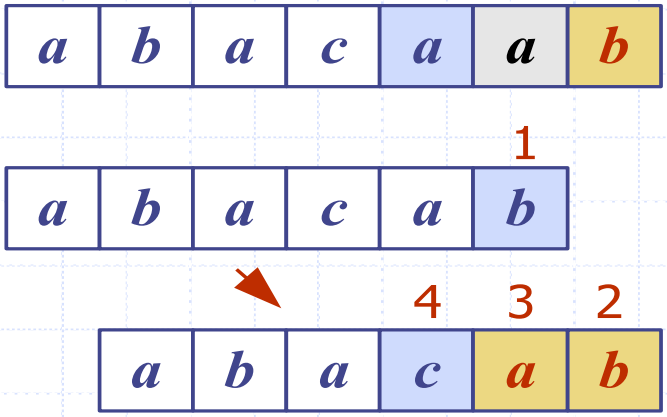
Processamento de Textos
Capítulo 11



Agenda

- Reconhecimento de Padrões
  - O método da força bruta (FB)
  - O método de Boyer-Moore (BM)
  - O método de Knuth-Morris-Pratt (KMP)
- Busca de Palavras em Textos
  - Tries
- Compressão de textos
  - Código de Huffman

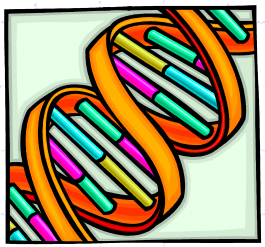
Reconhecimento de Padrões (Pattern Matching)



Strings (§ 11.1)

- Uma string é uma sequência ou cadeia de caracteres
- Exemplos de strings:
  - Um programa Java
  - Um documento HTML
  - Uma sequência de DNA
  - Uma imagem digitalizada
- Um alfabeto Σ é um conjunto de caracteres possíveis para uma família de strings
- Exemplos de alfabetos:
  - ASCII
  - Unicode
  - {0, 1}
  - {A, C, G, T}
- Considere uma string P de comprimento m
  - Uma substring $P[i..j]$ de P é a subsequência de P formada pelos caracteres nas posições de i até j inclusive
  - Um prefixo de P é uma substring do tipo $P[0..i]$, $i \leq m-1$
  - Um sufixo de P é uma substring do tipo $P[i..m-1]$, $i \geq 0$.
- Sejam as strings T (texto de tamanho n) e P (padrão), o problema do reconhecimento de padrões consiste em encontrar uma sub sequência em T, que seja igual a P. **Suponha que n >>> m.**
- Aplicações:
  - Editores de textos
  - Programas de busca (Search engines): Google, Altavista,etc
  - Pesquisa em biologia

Método da Força Bruta (§ 11.2.1)



- O algoritmo da força bruta (FB) para reconhecimento de padrões procura o padrão P, iniciando na primeira posição de T, e então sucessivamente testando P em todas as posições subsequentes de T, até que uma das condições ocorra:
  - o padrão P seja encontrado, ou
  - todas as posições de T onde P poderia estar tenham sido tentadas.
- O algoritmo FB executa em tempo O ((n-m+1)m) = $O(nm)$ passos
- Exemplo do pior caso:
  - $T = aaa \ldots ah$
  - $P = aaah$
  - pode ocorrer em imagens, ou em seqüências de DNA
  - não é comum em textos da língua Inglesa (ou Portuguesa)

Algoritmo *BruteForceMatch*(T, P)

Entrada: texto T de comprimento n e padrão P de comprimento m $(m \leq n)$

Saida: Índice de T onde começa uma substring igual a P, ou -1 caso não a encontre

para $i \leftarrow 0$ **até** $n - m$

{testa o deslocamento i do padrão}

$j \leftarrow 0$

enquanto $((j < m)$ *e* $(T[i + j] = P[j]))$

$j \leftarrow j + 1$

se $j = m$

retorne i {achou P na posição i}

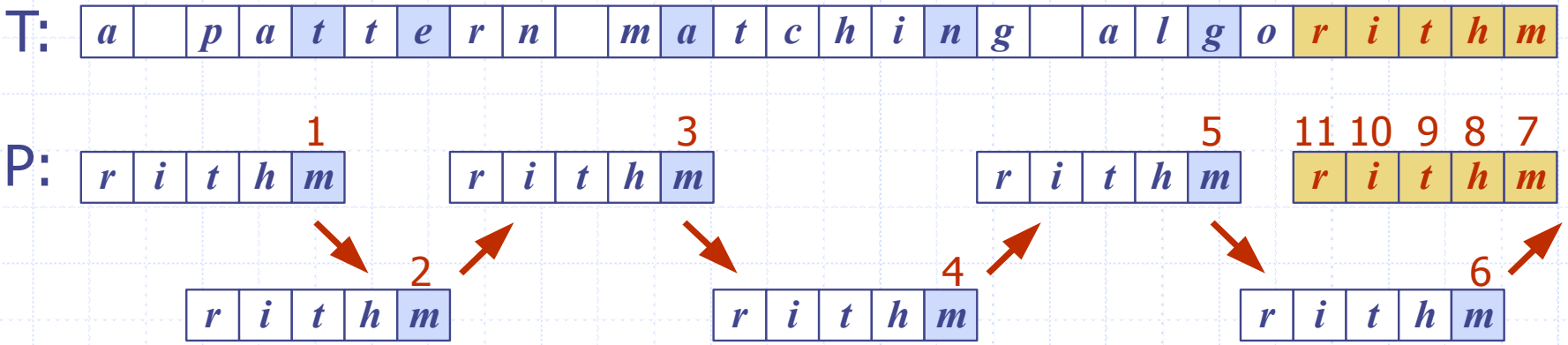
return **-1** {não achou uma substring igual a P em T }

O Método Boyer-Moore (§ 11.2.2)

- O algoritmo de Boyer-Moore (BM) para reconhecimento de padrões se baseia em duas heurísticas

  Heurística do espelho: compara P com uma subsequência de T do fim para o começo

  Heurística do salto de caracteres: quando um erro ocorre em $T[i] = c$
  - se P contém c, desloque P para frente, de modo a alinhar $T[i]$ com a última ocorrência de c em P
  - senão, desloque P para frente, de modo a alinhar $P[0]$ com $T[i + 1]$
- Exemplo



A Função "Last-Occurrence"

- O algoritmo BM efetua um pré-processamento sobre o padrão P e o alfabeto Σ para construir a função "*last-occurrence*" L
- Essa função mapeia caracteres de Σ para números inteiros, de modo que $L(c)$ é definido como
  - o maior índice i onde ocorre o caracter c (isto é, $P[i] = c$), ou
  - -1 se esse índice não existir
- Exemplo:
  - $\Sigma = \{a, b, c, d\}$
  - $P = abacab$

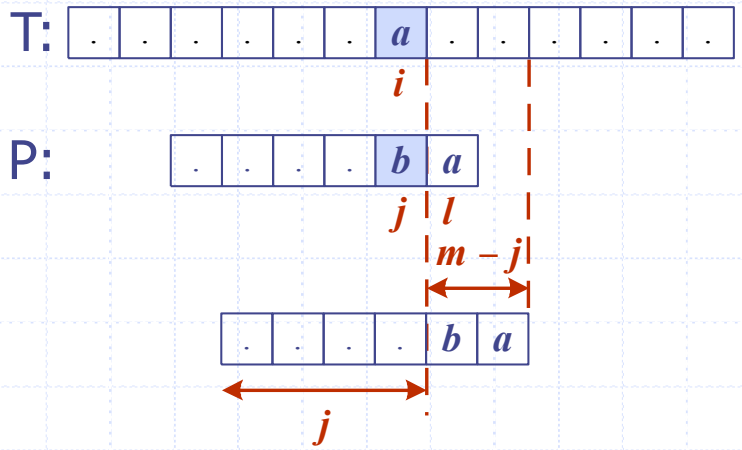
c	a	b	c	d
$L(c)$	4	5	3	-1

- A função "*last-occurrence*" pode ser representada por um vetor indexado pelo código numérico dos caracteres
- A função "last-occurrence" pode ser computada em tempo $O(m + s)$, onde m é o comprimento de P e s é o tamanho do alfabeto Σ

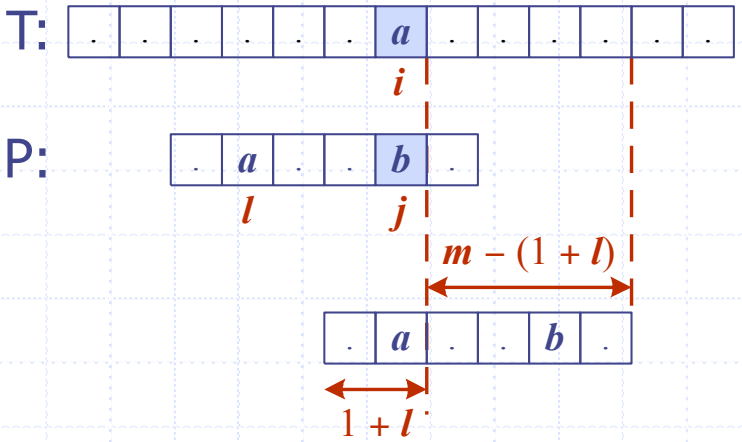
O algoritmo Boyer-Moore

```
Algoritmo BoyerMooreMatch(T, P, Σ)
    L ← lastOccurenceFunction(P, Σ )
    i ← m − 1
    j ← m − 1
    repita
      se T[i] = P[j]
          se j = 0
                retorne i {achou P em i }
          senão { heuristica do espelho}
                i ← i − 1
                j ← j − 1
      senão
         { heuristica do salto de caracteres}
          l ← L[T[i]]
          i ← i + m − min(j, 1 + l)
          j ← m − 1
    até que ( i > n − 1)
    retorne −1 { não achou P em T}
```

Caso 1: $j < 1 + l$



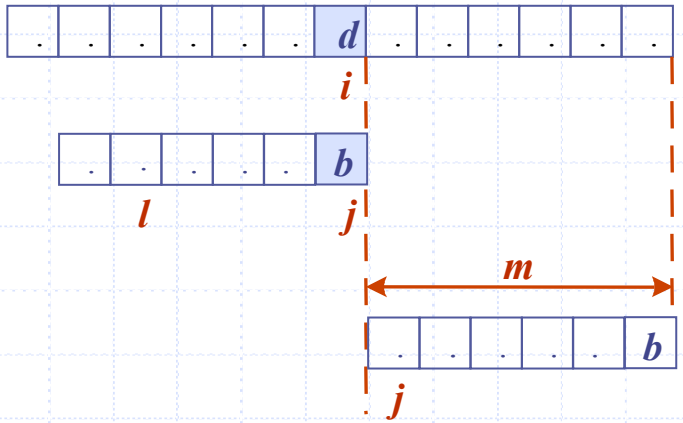
Caso 2: $1 + l \leq j$



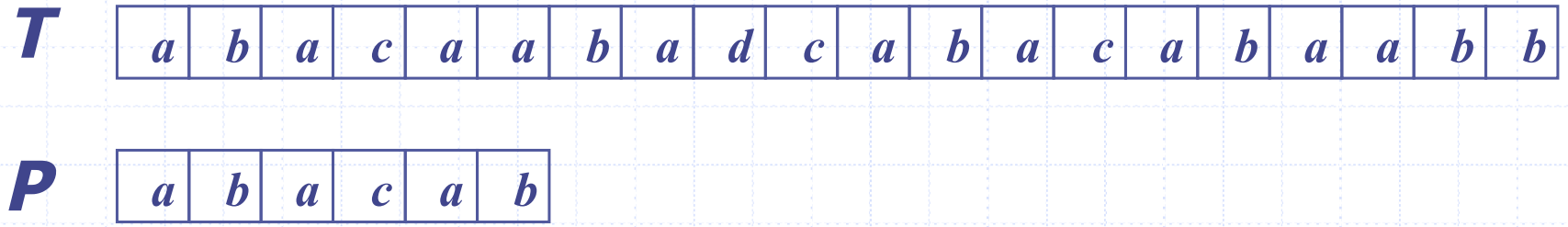
O algoritmo Boyer-Moore (cont.)

- Se **c** não for encontrado em **P**, então deslocamos **P** para frente em **m** posições

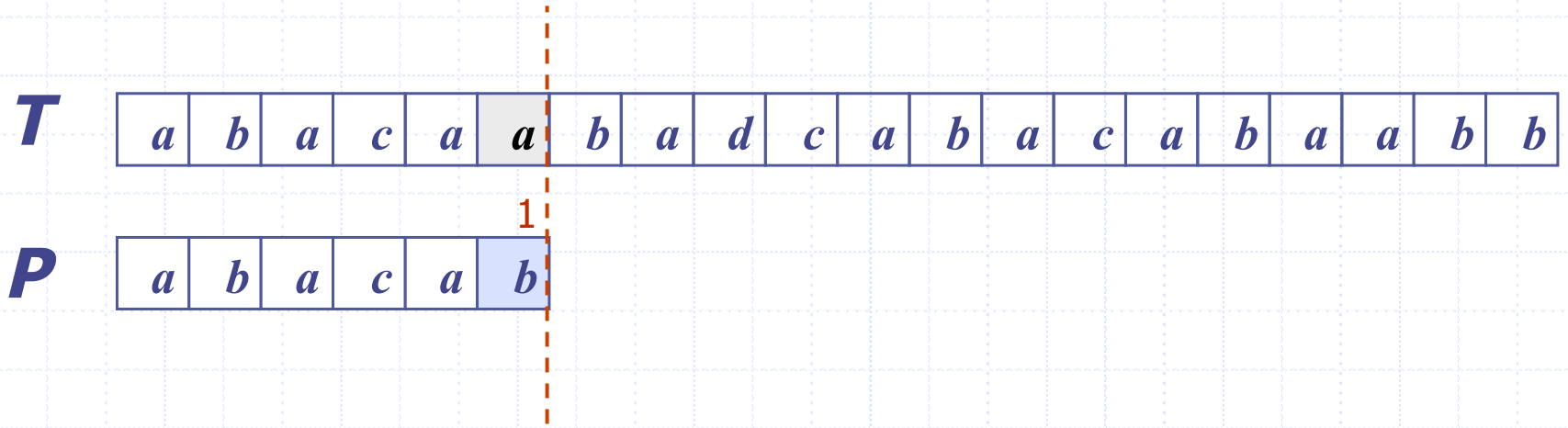
Caso 3:



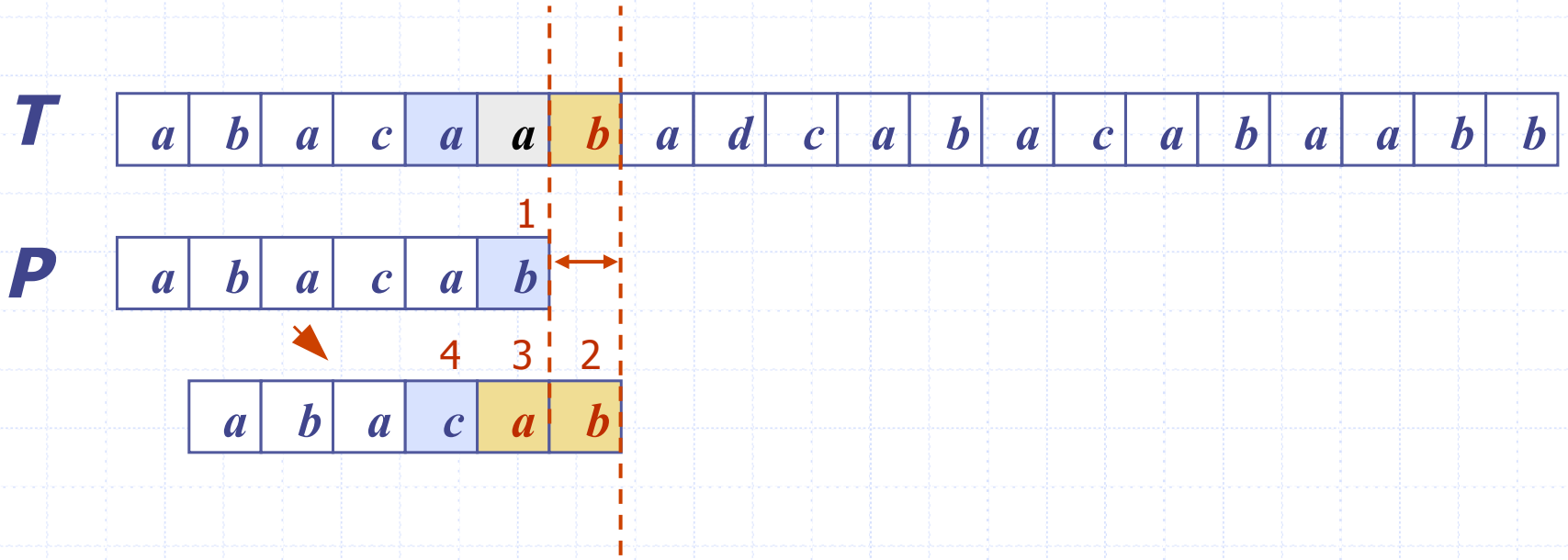
Exemplo



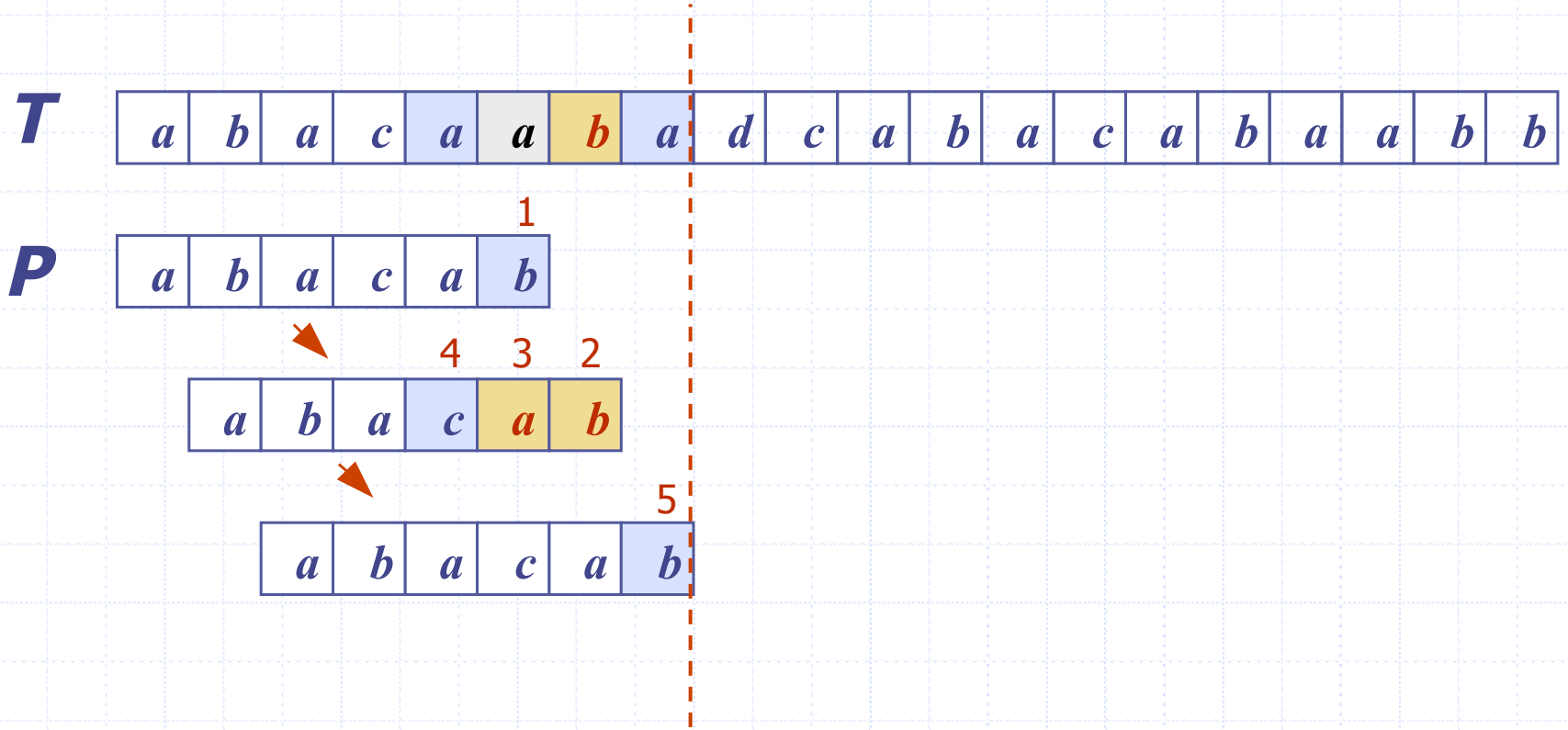
Exemplo



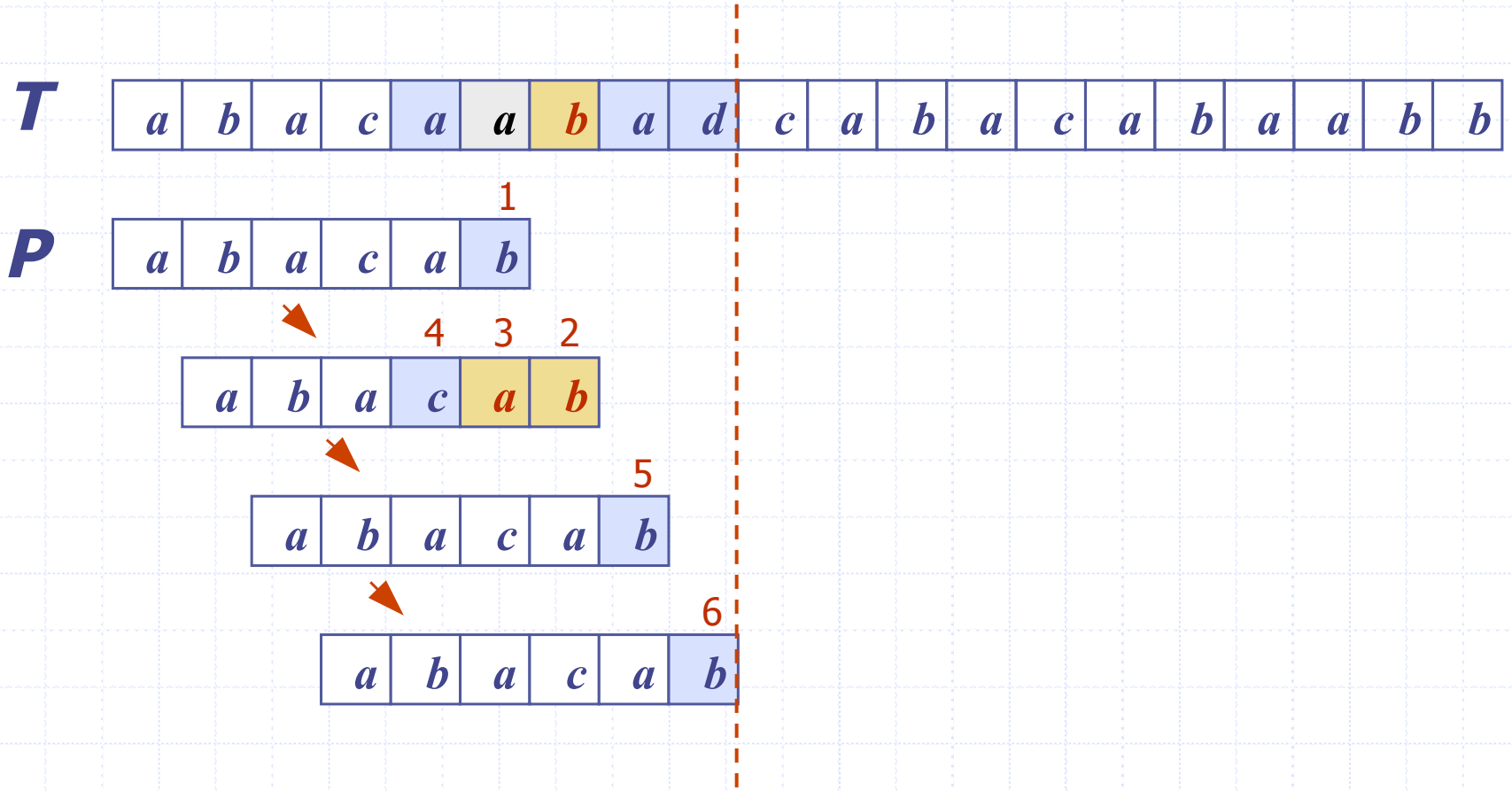
Exemplo



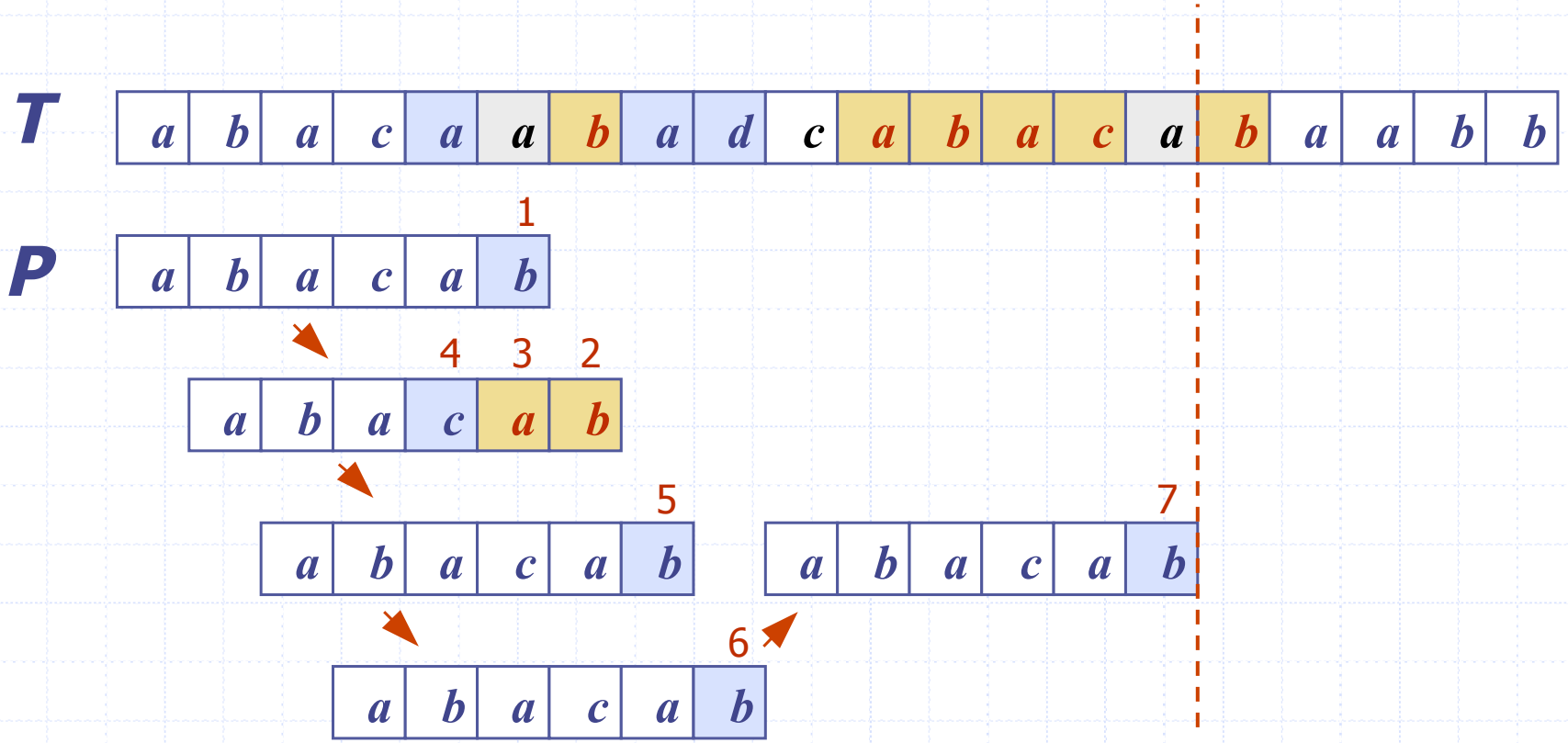
Exemplo



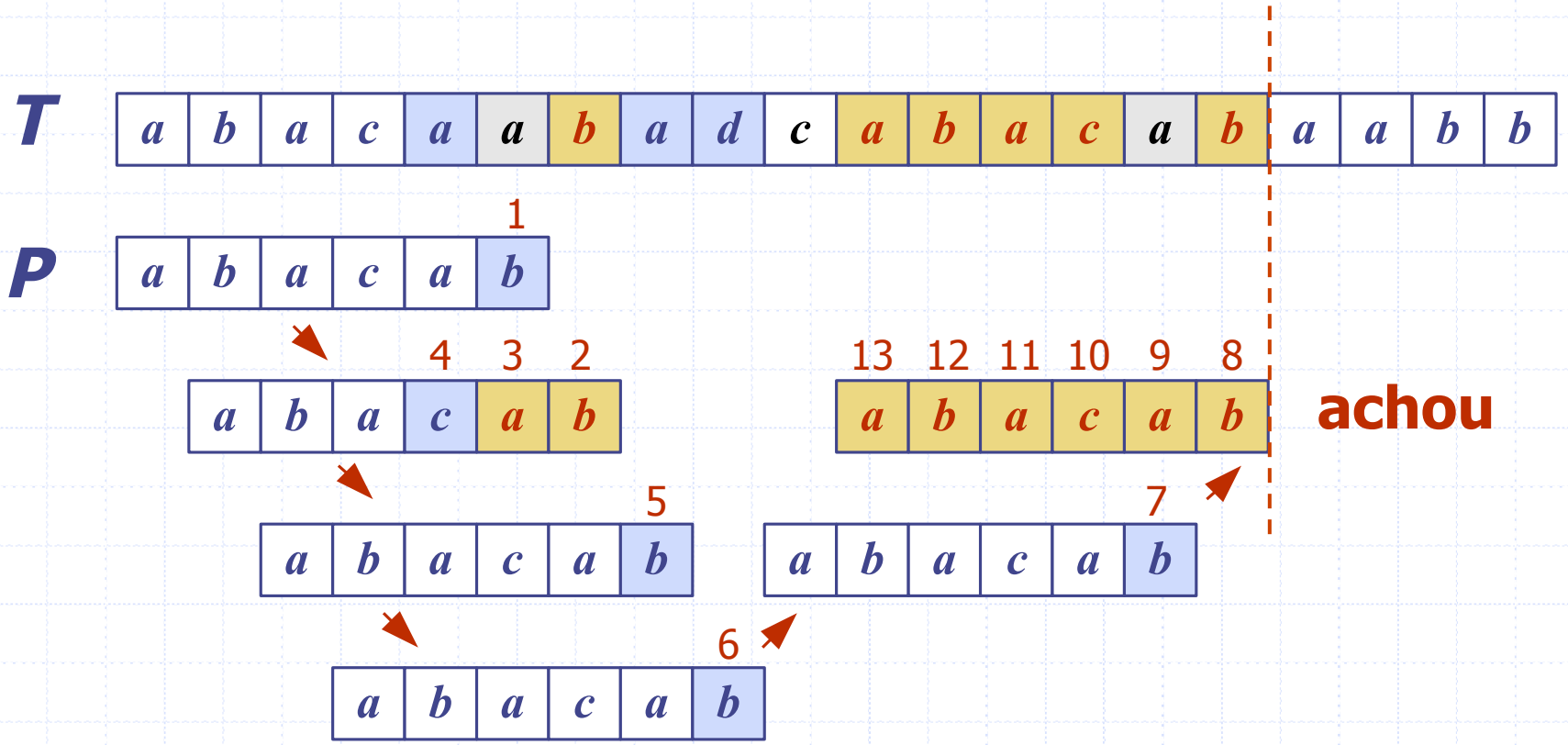
Exemplo


T
a b a c a a b a d c a b a c a b a a b b
P
1
a b a c a b
4 3 2
a b a c a b
5
a b a c a b
6
a b a c a b

Exemplo

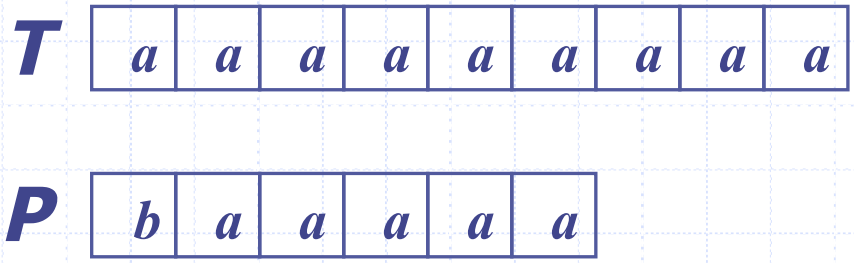

T
a b a c a a b a d c a b a c a b a a b b
P
1
a b a c a b
4 3 2
a b a c a b
5
a b a c a b
7
a b a c a b
6
a b a c a b

Exemplo


T
a b a c a a b a d c a b a c a b a a b b
P
1
a b a c a b
4 3 2
a b a c a b
5
a b a c a b
6
a b a c a b
7
a b a c a b
13 12 11 10 9 8
a b a c a b
achou

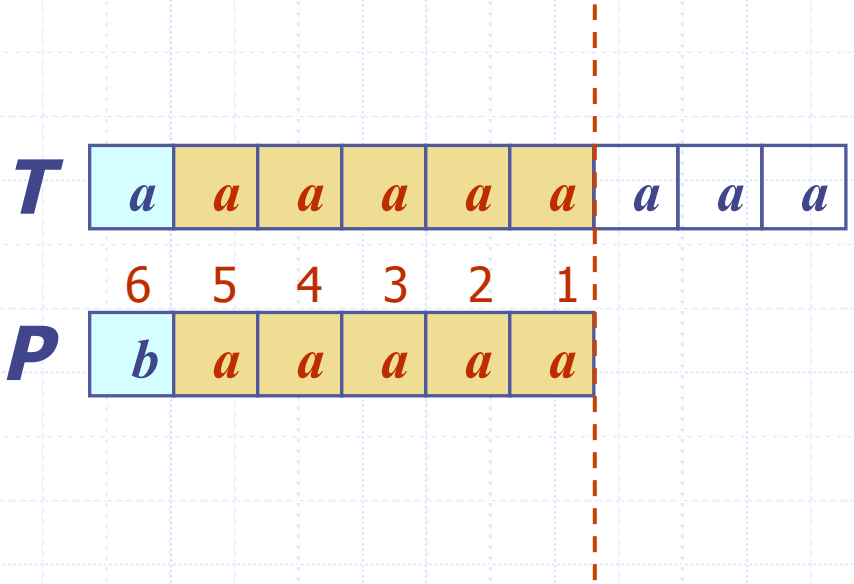
Análise

- O algoritmo BM executa em tempo $O(nm)$ no pior caso
- Exemplo do pior caso:
  - $T = aaa \ldots a$
  - $P = baaa$
- O pior caso pode ocorrer em imagens e sequências de DNA, mas é incomum em textos em Inglês, Português, etc.
- O algoritmo de Boyer-Moore é significativamente mais rápido do que o algoritmo força-bruta para textos em Inglês.



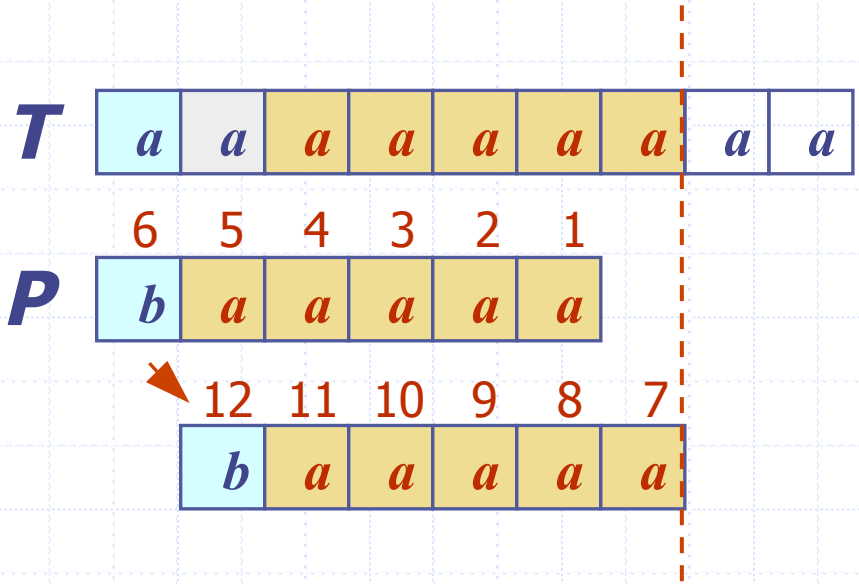
Análise

- O pior caso



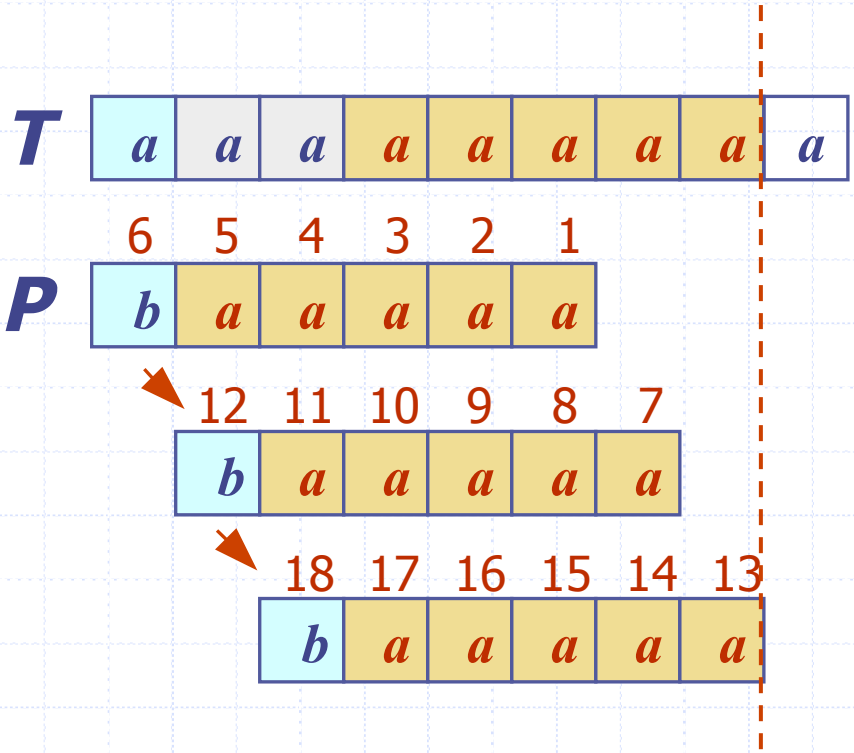
Análise

- O pior caso



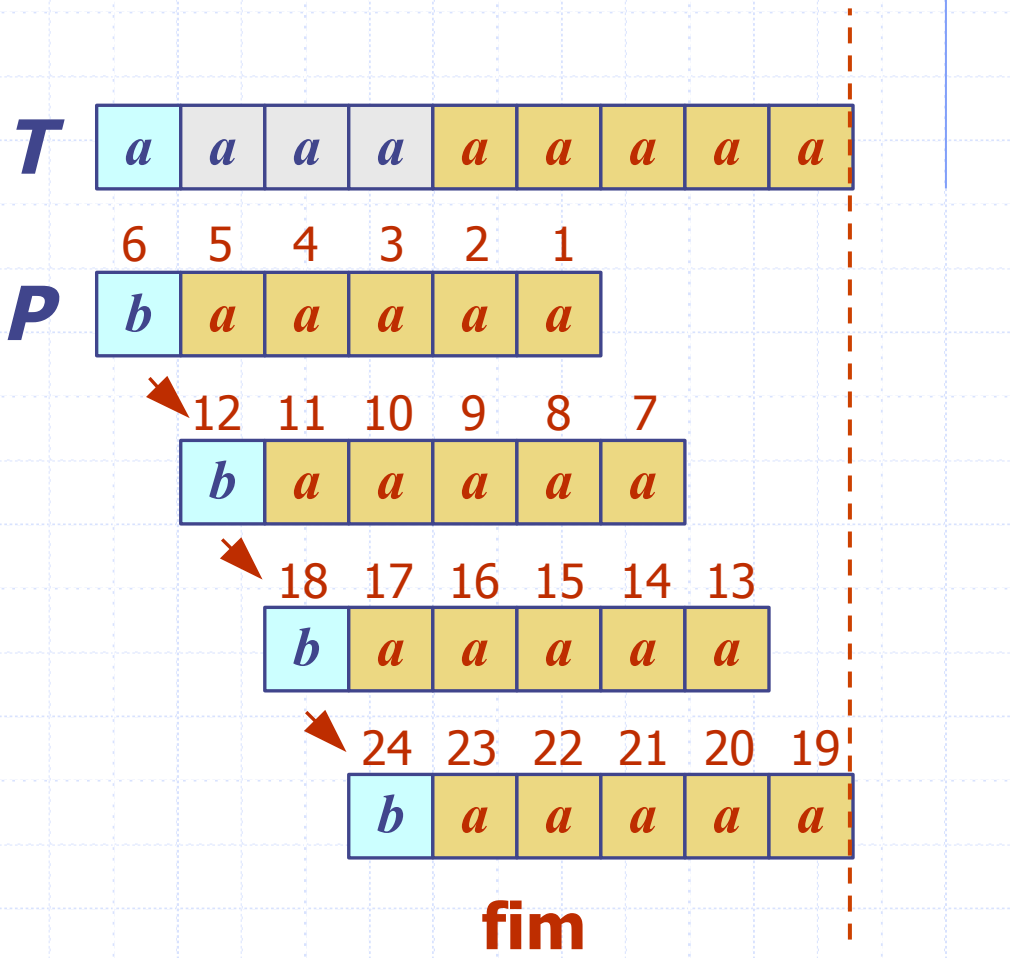
Análise

- O pior caso



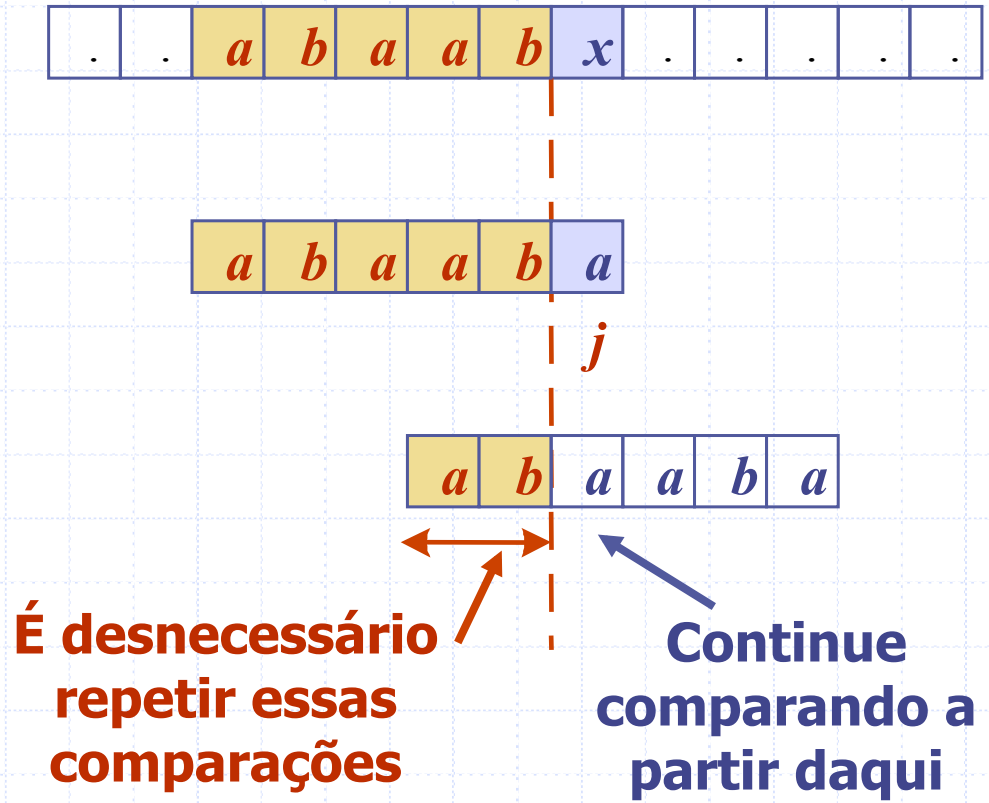
Análise

- O pior caso

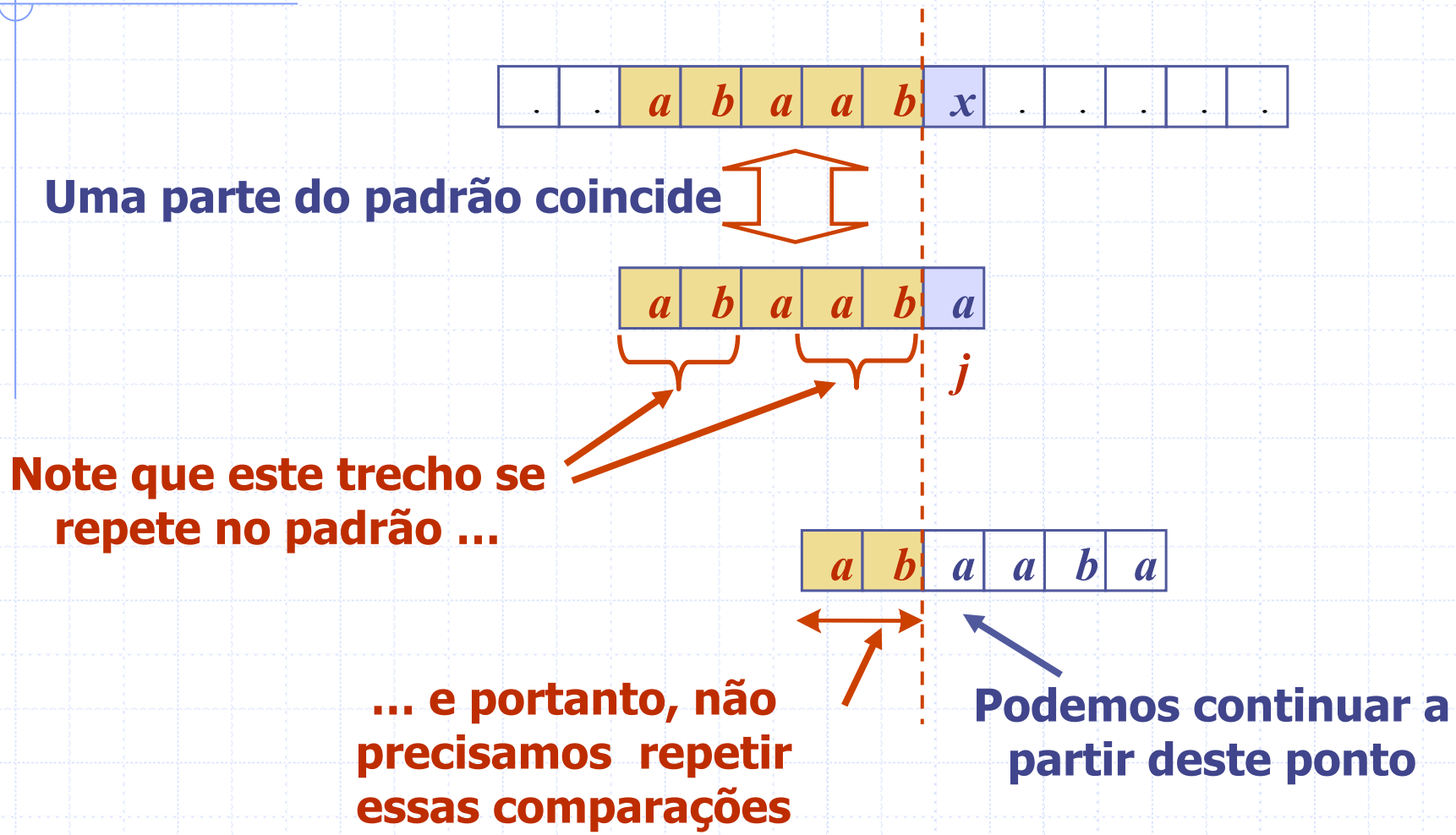


O algoritmo KMP (§ 11.2.3)

- O algoritmo de Knuth-Morris-Pratt compara o padrão com o texto da **esquerda-para-direita**, mas desloca o padrão de uma forma mais inteligente do que o algoritmo da força-bruta (FB)
- Quando ocorre uma desigualdade, qual seria o deslocamento **máximo** possível, de modo a evitar comparações redundantes ?
- Resposta: o maior prefixo de $P[0..j]$ que seja também um sufixo de $P[1..j]$



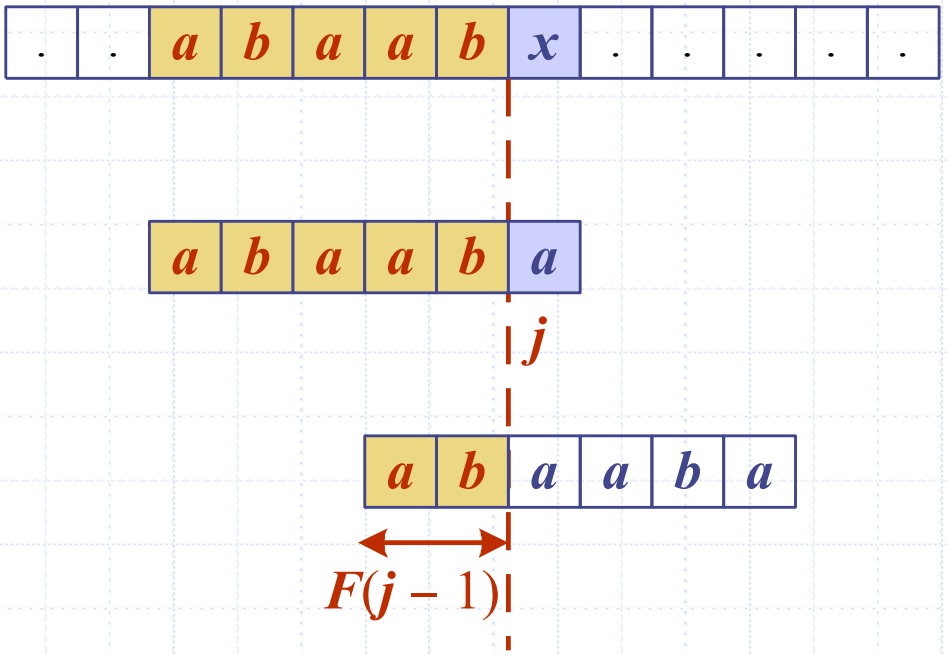
O algoritmo KMP - princípio



A Função de Falha do KMP

- O algoritmo KMP efetua um pré-processamento inicial no padrão, para saber se existe um prefixo que se repete dentro do próprio padrão mais adiante
- A **função de falha** $F(j)$ é definida como o comprimento do maior prefixo $P[0..j]$ que seja também um sufixo de $P[1..j]$
- O algoritmo KMP é uma evolução do algoritmo da força-bruta (FB): quando ocorre uma falha na comparação (i.e., $P[j] \neq T[i]$) ele desloca o padrão para frente, fazendo $j \leftarrow F(j-1)$

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

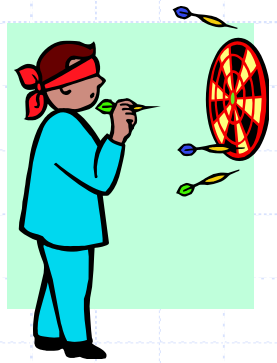


O algoritmo KMP

- A função de falha pode ser representada como um vetor e pode ser calculada em tempo $O(m)$
- A cada iteração do laço temos
  - um incremento de i em uma unidade, ou
  - o tamanho do deslocamento (que é de $i - j$ posições) aumenta em pelo menos uma unidade (observe que $F(j-1) < j$)
- Assim, não poderá haver mais do que $2n$ iterações do laço ***enquanto***
- Portanto, o algoritmo KMP executa em tempo ótimo $O(m+n)$

```
Algoritmo KMPMatch(T, P)
    F ← failureFunction(P)
    i ← 0
    j ← 0
    enquanto i < n
        se T[i] = P[j]
            se j = m − 1
                retorne i − j { achou}
            senão
                i ← i + 1
                j ← j + 1
        senão
            if j > 0
                j ← F[j − 1]
            else
                i ← i + 1
    return −1 { não achou}
```

Computando a Função de Falha do KMP

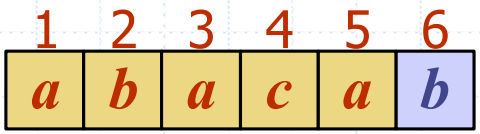


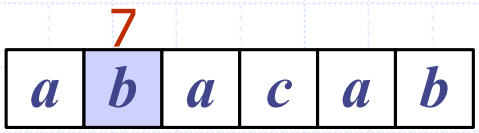
- A função de falha pode ser representada como um vetor e pode ser computada em tempo $O(m)$
- Sua construção é similar à construção do próprio KMP
- A cada iteração do laço temos que
  - i é incrementado em uma unidade, ou
  - o tamanho do deslocamento, dado por $i - j$, aumenta em pelo menos uma unidade (observe que $F(j-1) < j$)
- Portanto, não pode haver mais do que $2m$ iterações do laço "enquanto"

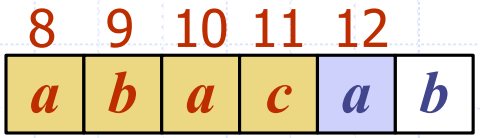
Algoritmo *failureFunction*(P)

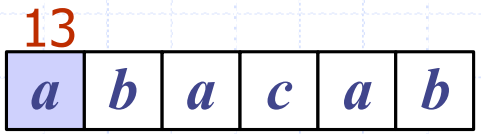
```
F[0] ← 0
i ← 1
j ← 0
enquanto i < m
    se P[i] = P[j]
        {já encontramos j + 1 caracteres}
        F[i] ← j + 1
        i ← i + 1
        j ← j + 1
    senão se j > 0 então
    {j indexa logo após um prefixo de P }
        j ← F[j − 1]
    senão
        F[i] ← 0 { não achou }
        i ← i + 1
```

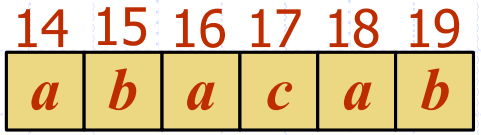
Exemplo











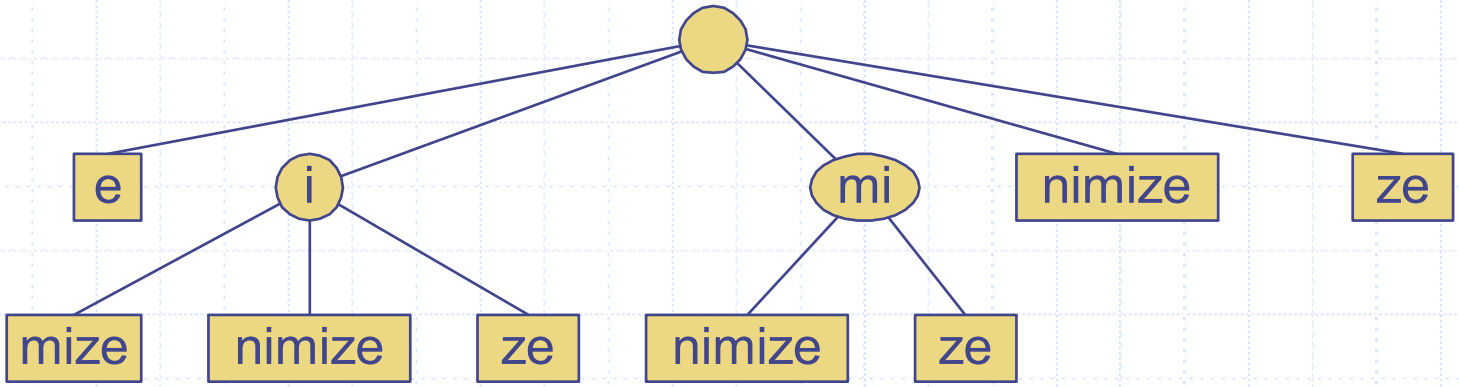
j	0	1	2	3	4	5
$P[j]$	a	b	a	c	a	b
$F(j)$	0	0	1	0	1	2

Processamento de Textos

Roteiro

- Busca de Palavras em Textos
  - Tries
- Compressão de textos
  - Código de Huffman

Trie ("retrieval")

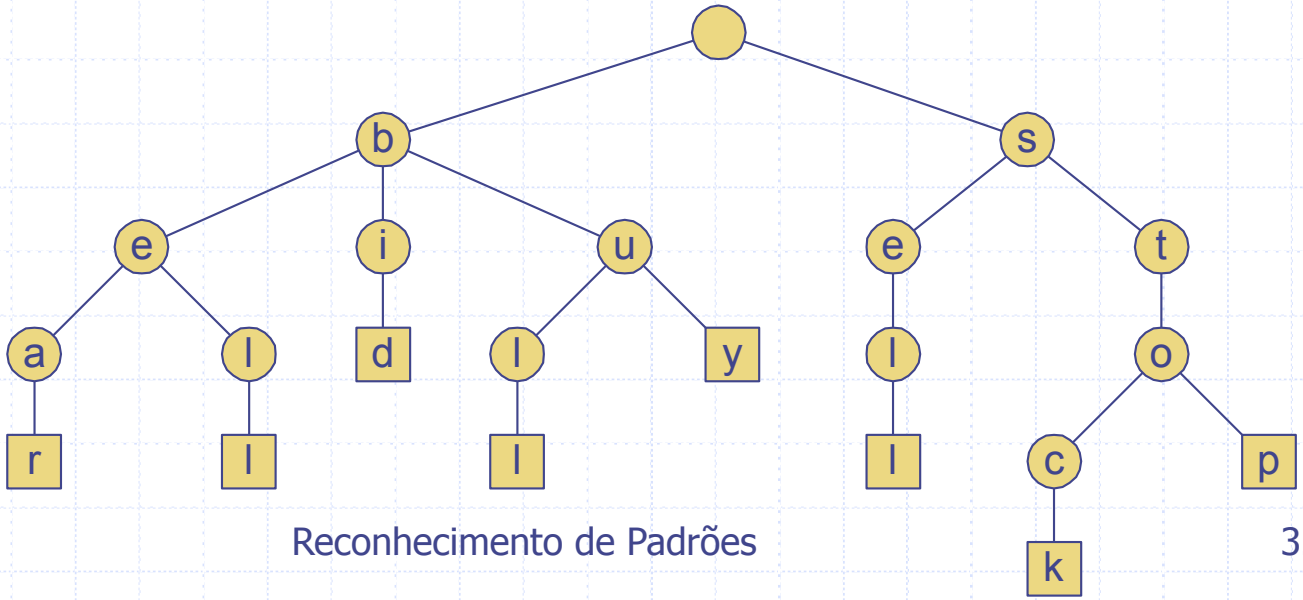


Pré-processando Strings

- **Pré-processar** o **padrão**, antes de fazer a busca, pode reduzir o tempo de execução
  - Depois de pré-processar o padrão, o algoritmo KMP tenta fazer a busca em tempo linear, proporcional ao tamanho do texto.
- Se o texto for grande, tiver um conteúdo fixo (i.e. não é modificado), e for consultado freqüentemente, então podemos **pré-processar** o **texto**, em vez do padrão.
- Um "trie" é uma estrutura de dados compacta, que permite representar um conjunto de strings (por exemplo, todas as palavras de um texto).
  - Com um "trie", podemos processar buscas (de padrões ou prefixos) em tempo proporcional ao tamanho do padrão (complexidade linear).

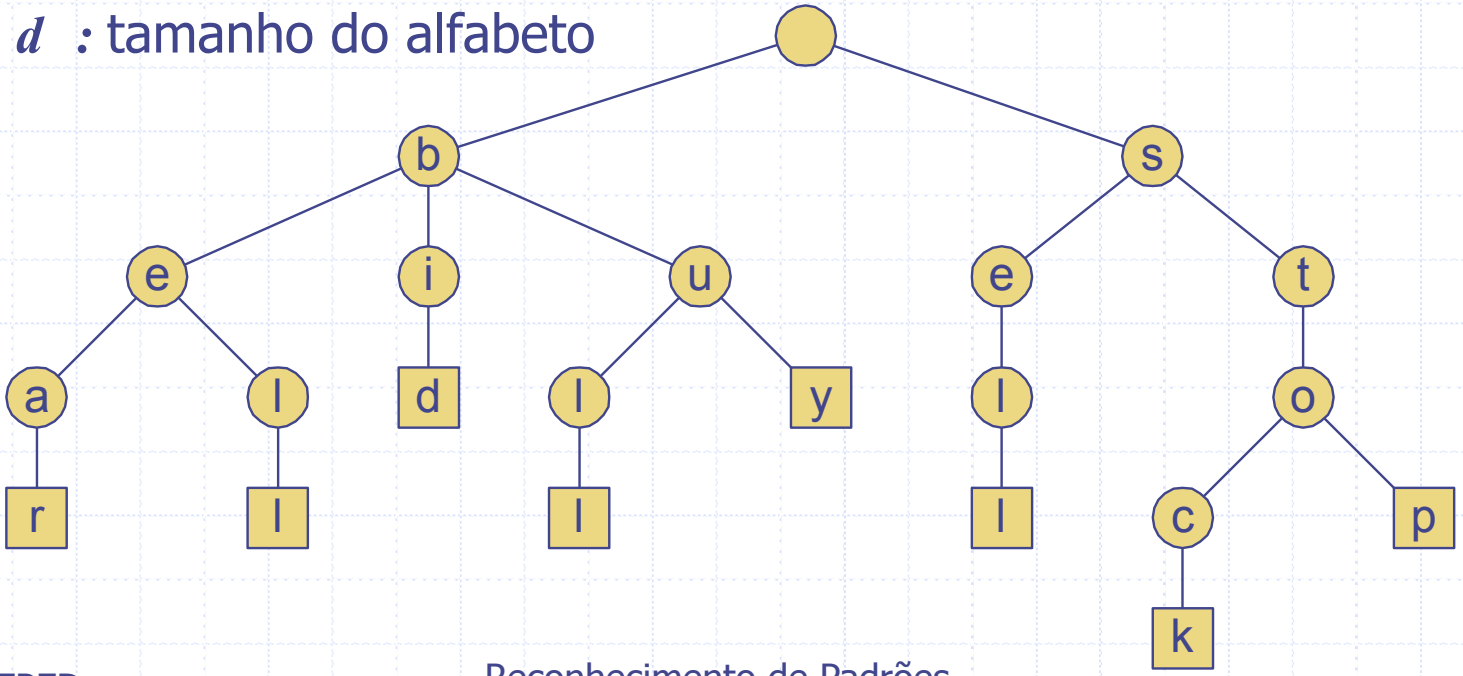
Trie Padrão (§ 11.3.1)

- Um "trie" padrão para um conjunto S de seqüências é uma **árvore ordenada** que satisfaz às seguintes propriedades:
  - Todo nó exceto a raiz é rotulado com um caracter do alfabeto.
  - Os filhos de um nó são ordenados.
  - Os caminhos partindo da raiz até os nós externos reproduzem todas as seqüência de S (*nenhuma seqüência é prefixo de outra seqüência de S*). A altura do "trie" é o comprimento da maior seqüência em S.
- Exemplo: trie padrão para o conjunto de seqüências
  S = { bear, bell, bid, bull, buy, sell, stock, stop }



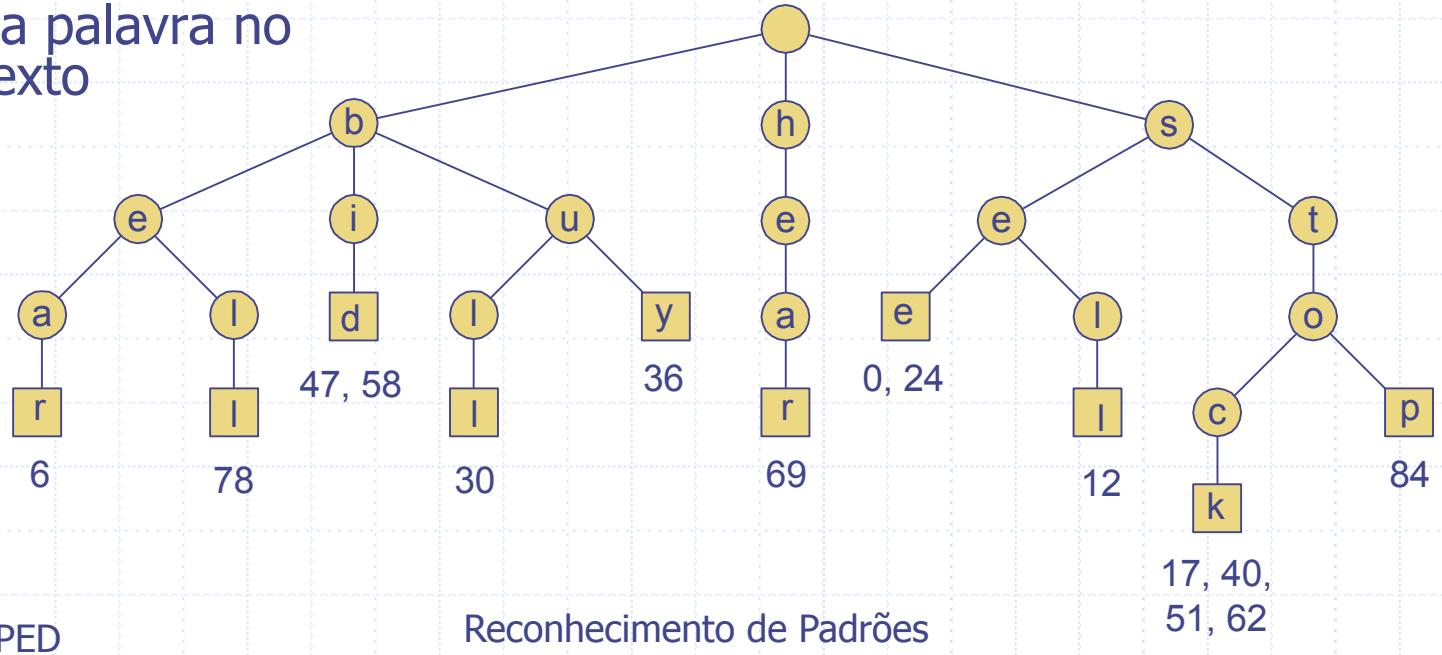
Trie Padrão

- Um "trie" padrão utiliza espaço de memória $O(n)$ e permite buscas, inserções e remoções em tempo $O(dm)$, onde:
  - n : número de strings no conjunto S
  - m : comprimento da string utilizada como argumento de busca
  - d : tamanho do alfabeto



Reconhecendo palavras com Tries

- Armazenar todas as palavras do texto em um trie
- Cada folha armazena as ocorrências da palavra no texto



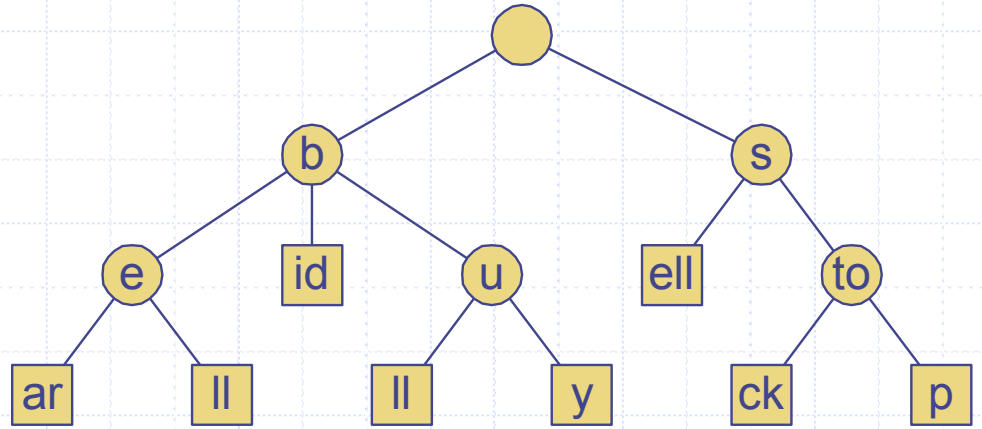
Tries

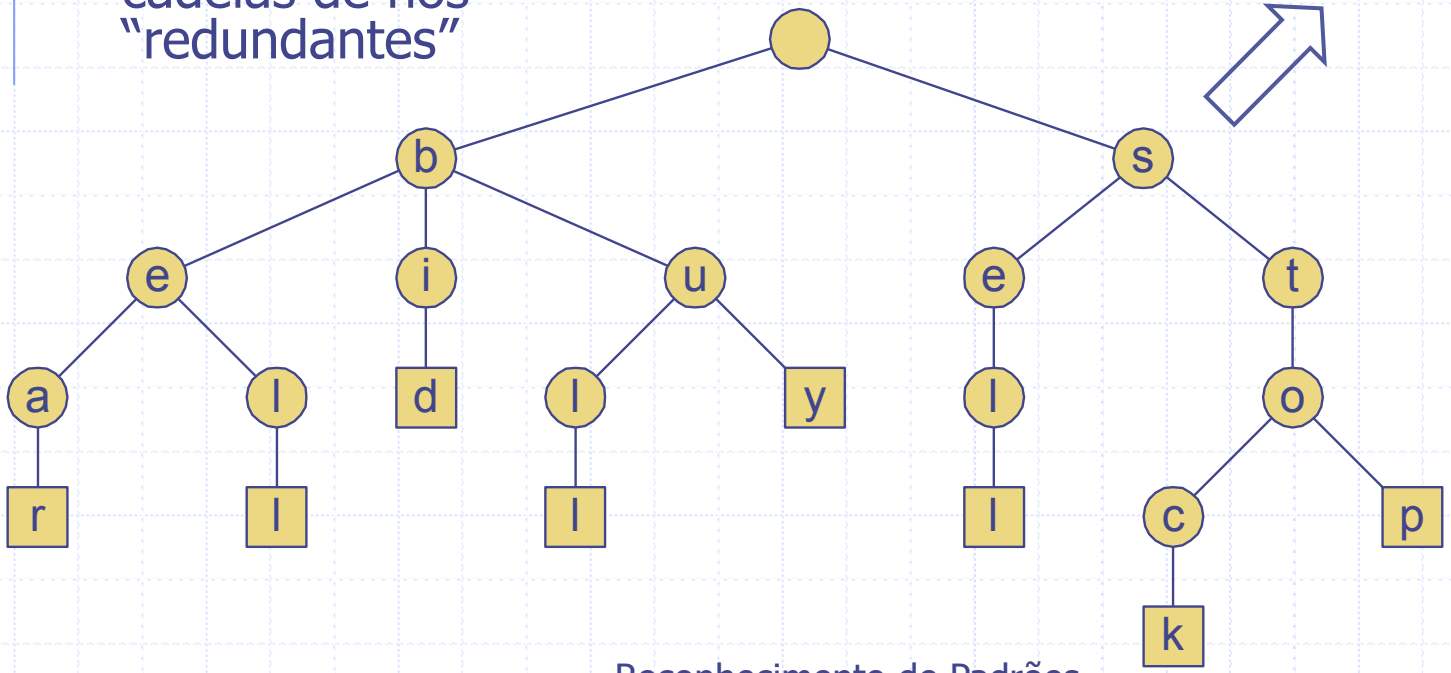
- O pior caso do número de nós de um "trie" acontece quando nenhum par de sequências compartilha de um prefixo, ou seja, exceto pela raiz, todos os nós internos têm apenas um filho.

- Um "trie" T para um conjunto S de sequências pode ser usado para implementar um **dicionário** cujas chaves são as sequências de S.

Tries Comprimidos (§ 11.3.2)

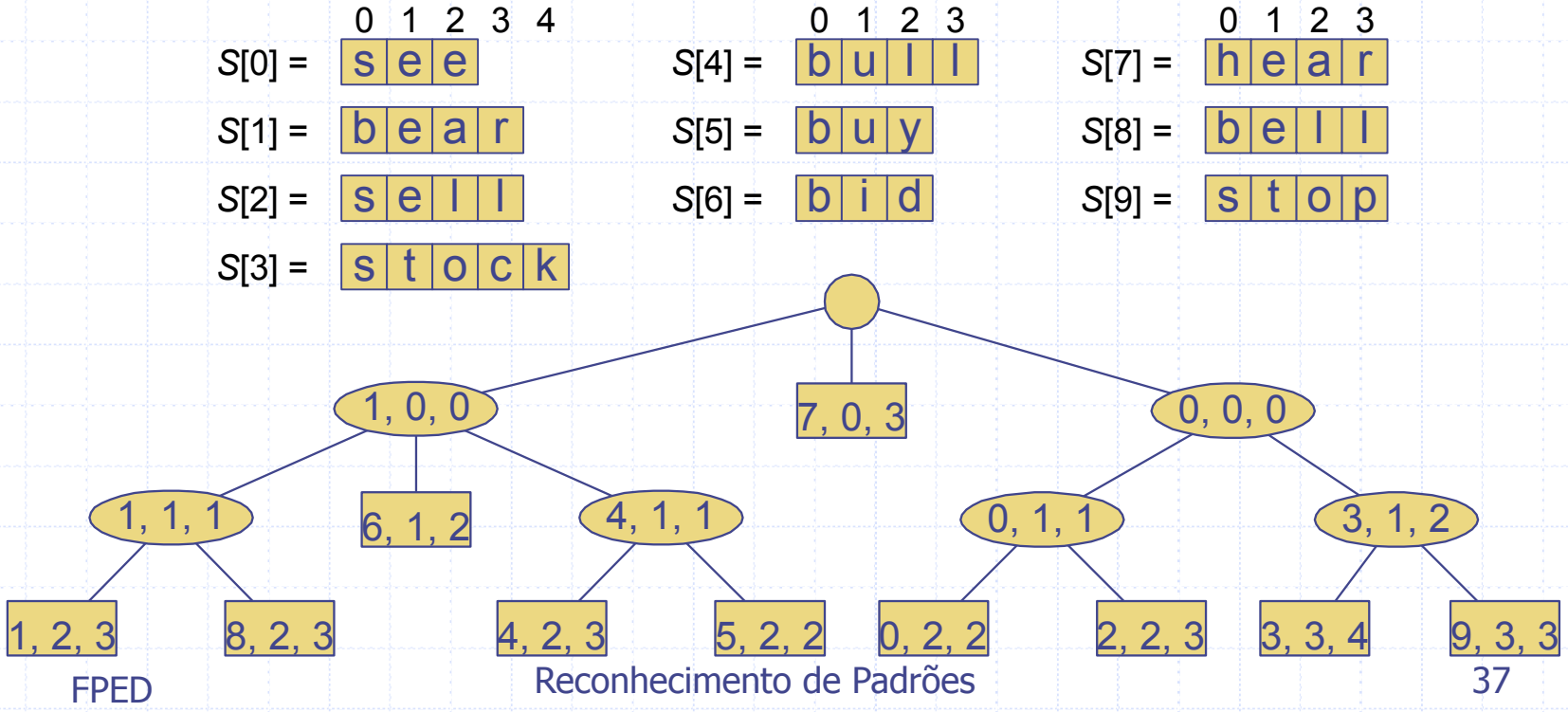
- Um "trie" comprimido possui nós internos de grau no mínimo igual a dois
- É obtido a partir do "trie" padrão, comprimindo as cadeias de nós "redundantes"





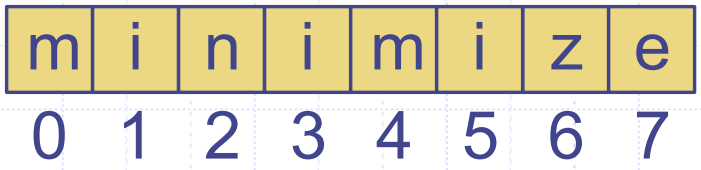
Representação Compacta

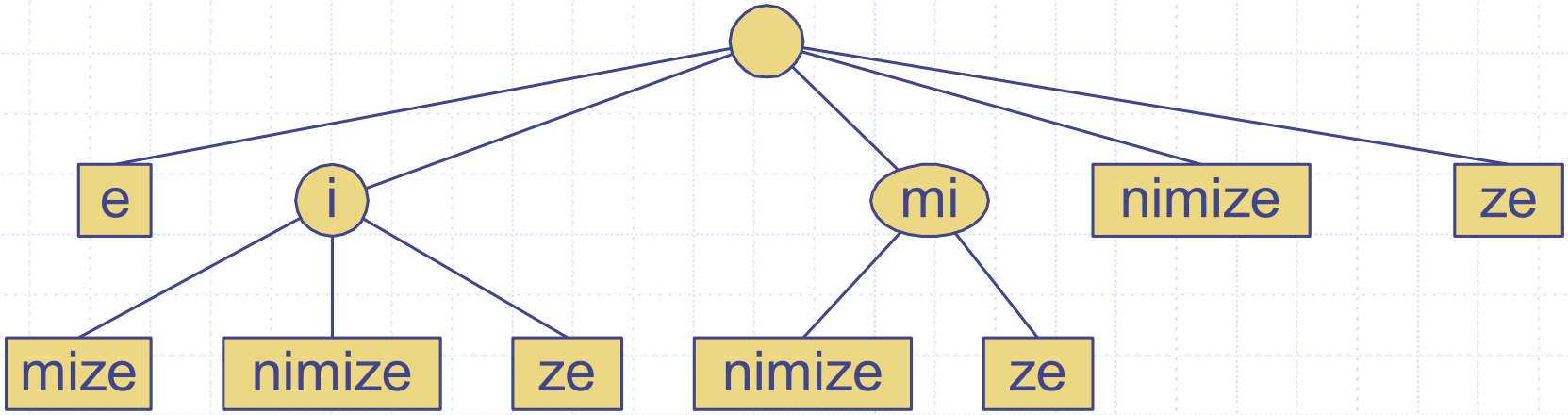
- A representação compacta de um "trie" comprimido para um vetor de strings:
  - Armazena intervalos de índices, em vez de substrings, em seus nós
  - Consome espaço de memória $O(s)$, onde s é a quantidade de strings no vetor
  - Serve como uma estrutura de índices auxiliar: rótulo X(i,j,k) = S[i][j..k]



Trie de Sufixos (§ 11.3.3)

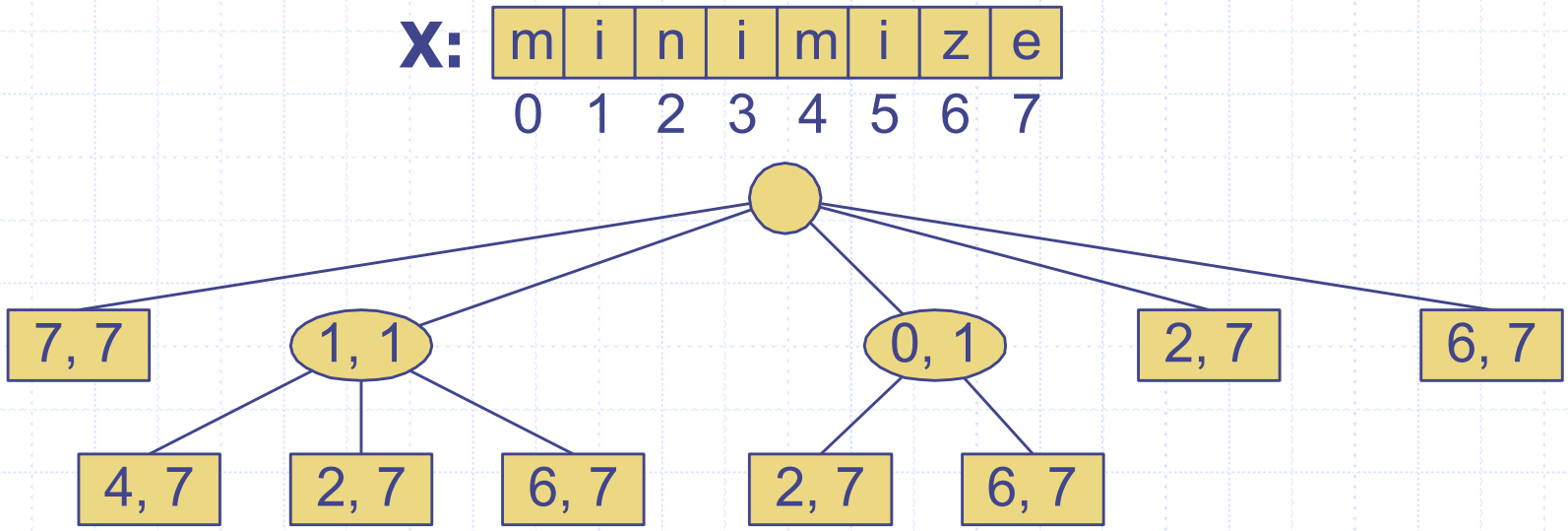
- Uma aplicaçao dos "tries" ocorre quando as sequências de S são sufixos de X.
- O trie de **sufixos** de uma string X é o "trie" comprimido de todos os sufixos de X



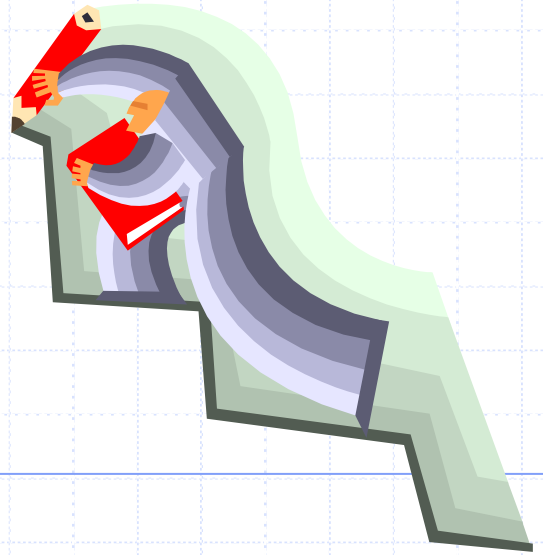


Eficiência de Tries de Sufixo

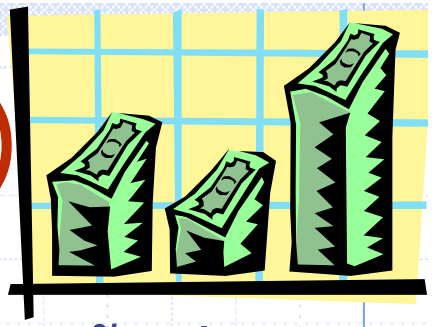
- A representação compacta do trie de sufixo para a string X de comprimento n de um alfabeto de tamanho d
  - Usa espaço em memória que é $O(n)$.
  - Permite buscas de padrões arbitrários em X em tempo $O(dm)$, onde m é o comprimento do padrão.
  - Pode ser construído em tempo $O(n)$.



Compressão de Textos
O Método Guloso



O Método Guloso (§ 11.4.2)



- **O Método Guloso** é um paradigma de construção de algoritmos de otimização, baseado em:
  - **configurações**: diferentes escolhas, coleções ou valores a serem encontrados.
  - **função objetivo**: pontuações associadas a cada configuração, fazer escolhas para maximizar ou minimizar algum aspecto.
- Em geral, parte-se de uma configuração inicial, cujo custo pode ser calculado. Em seguida, segue-se iterativamente fazendo uma seqüência de escolhas, com base no melhor custo disponível no momento, de modo a atingir o menor custo global
- Funciona bem quando aplicado a problemas que satisfazem a propriedade de **escolha-gulosa**:
  - uma solução global ótima pode sempre ser atingida através de uma série de escolhas feitas a partir da configuração inicial.

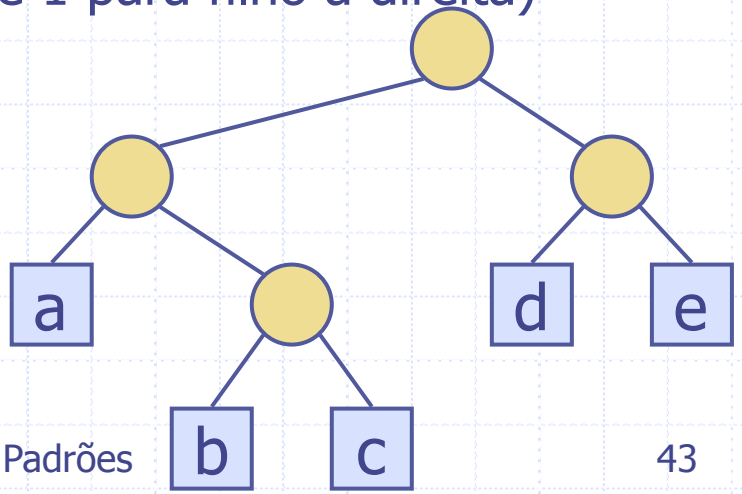
Compressão de Textos (§ 11.4)

- Dada uma string X, codificar de modo eficiente X como uma string Y de menor tamanho
  - Economia de memória ou largura de banda
- Uma boa abordagem: **códigos de Huffman**
  - Calcular a freqüência relativa f(c) de cada caracter c.
  - Codifique os caracteres mais freqüentes com as palavras de menor tamanho
  - Nenhuma palavra do código deve ser prefixo de outra
  - Use uma árvore de codificação para determinar as palavras de código

Árvore de Codificação - Exemplo

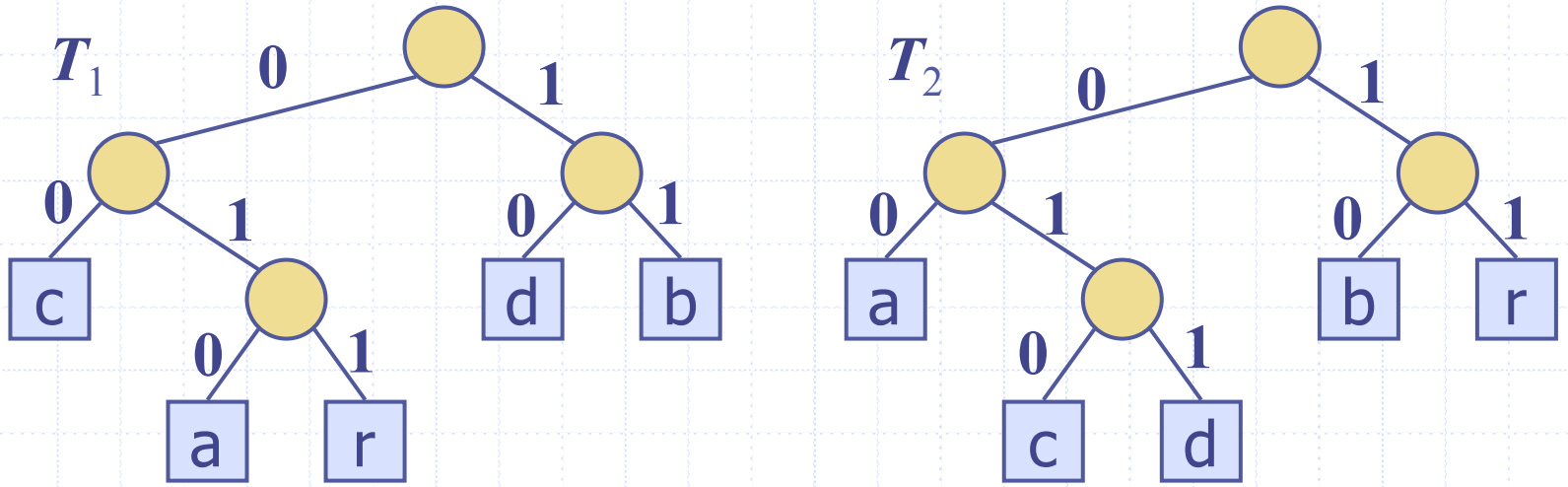
- Um **código** é um mapeamento de cada caracter de um alfabeto para uma palavra de códigos binários
- Um **código de prefixos** é um código binário tal que nenhuma palavra desse código é prefixo de outra
- Uma **árvore de codificação** representa um código de prefixos
  - Cada nó externo armazena um caracter
  - A palavra de código de um caracter é dada pelo caminho partindo da raiz até o nó externo que armazena esse caracter (sendo 0 para filho à esquerda, e 1 para filho à direita)

00	010	011	10	11
a	b	c	d	e



Otimizando a Árvore

- Dada um texto X, queremos encontrar um código de prefixos para os seus caracteres de tal forma a produzir uma codificação compacta de X
  - Caracteres mais frequentes devem ter códigos mais curtos
  - Caracteres mais raros devem ter códigos longos
- Exemplo
  - X = abracadabra
  - T_1 codifica X com 29 bits
  - T_2 codifica X com 24 bits



Algoritmo de Huffman

- Dada uma string X, o algoritmo de Huffman constrói um código de prefixo que minimiza a codificação de X
- Executa em tempo $O(n + d \log d)$, onde n é o comprimento de X e d é o número de caracteres distintos de X
- Uma fila de prioridades baseada em um heap é utilizada como estrutura de dados auxiliar

Algoritmo *HuffmanEncoding*(X)
  Entrada string X de comprimento n
  Saída árvore de codificação ótima para X
  $C \leftarrow$ *distinctCharacters*(X)
  computeFrequencies(C, X)
  $Q \leftarrow$ novo heap vazio
  para todo $c \in C$
    $T \leftarrow$ nova árvore c/ um único nó contendo c
    Q.*insert*(*getFrequency*(c), T)
  enquanto Q.*size*() > 1
    $f_1 \leftarrow Q$.*minKey*()
    $T_1 \leftarrow Q$.*removeMin*()
    $f_2 \leftarrow Q$.*minKey*()
    $T_2 \leftarrow Q$.*removeMin*()
    $T \leftarrow$ *join*(T_1, T_2)
    Q.*insert*($f_1 + f_2$, T)
  retorne Q.*removeMin*()

Exemplo

X = abracadabra

Freqüências

a	b	c	d	r
5	2	1	1	2

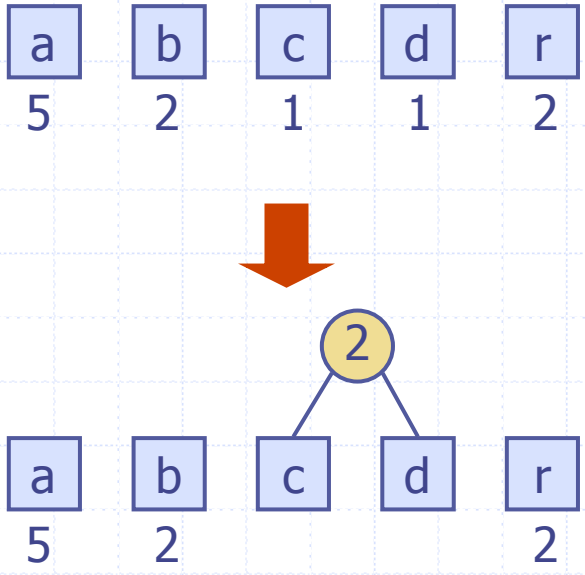
a	b	c	d	r
5	2	1	1	2

Exemplo

X = abracadabra

Freqüências

a	b	c	d	r
5	2	1	1	2

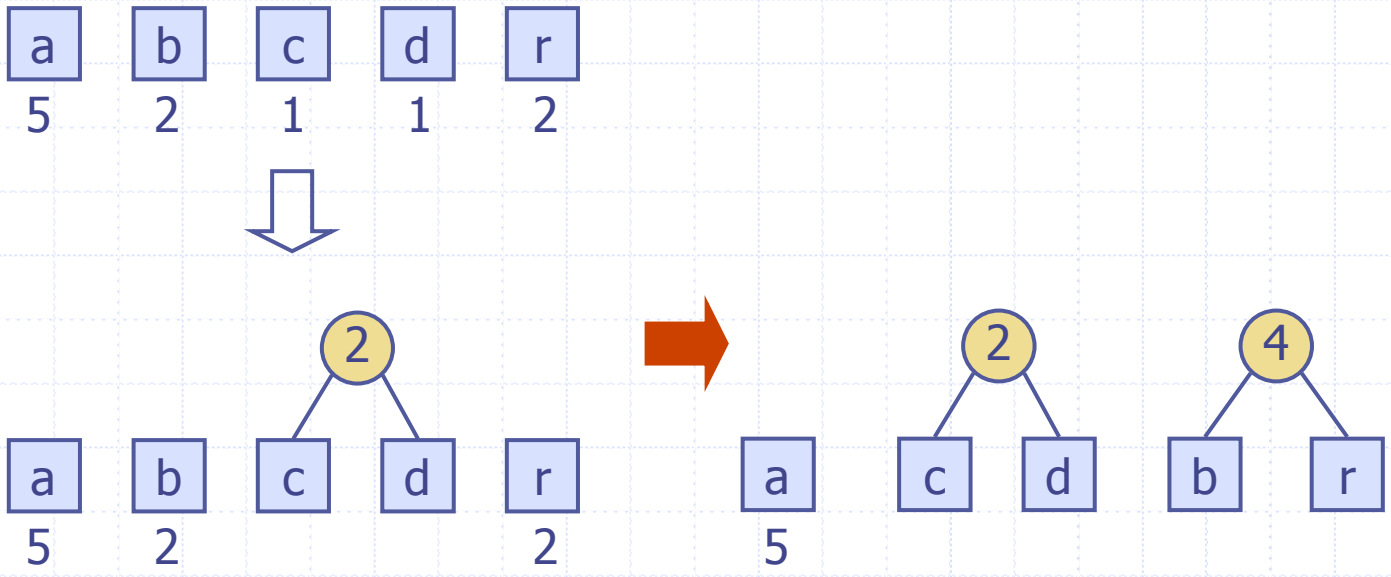


Exemplo

X = abracadabra

Freqüências

a	b	c	d	r
5	2	1	1	2

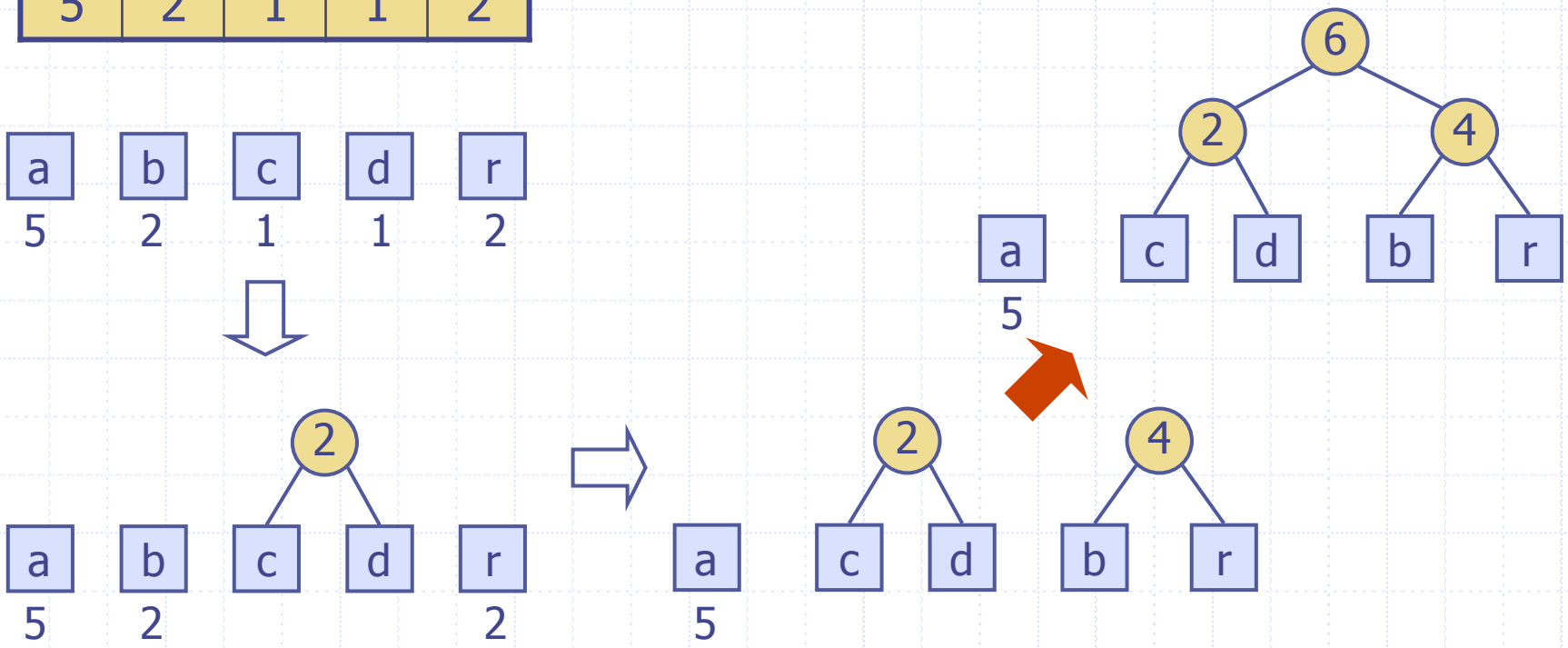


Exemplo

X = abracadabra

Freqüências

a	b	c	d	r
5	2	1	1	2

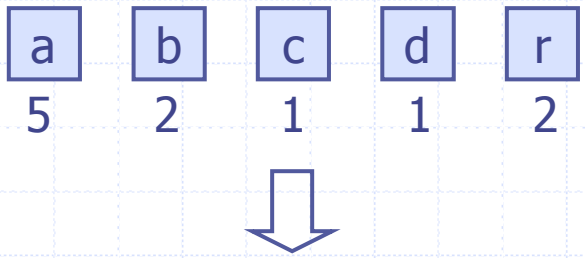


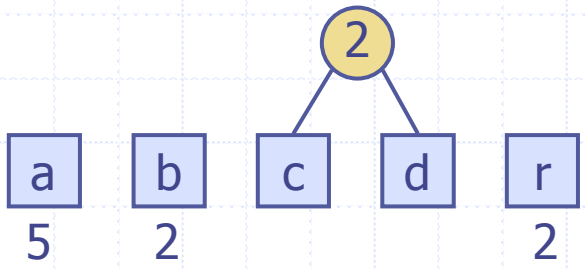
Exemplo

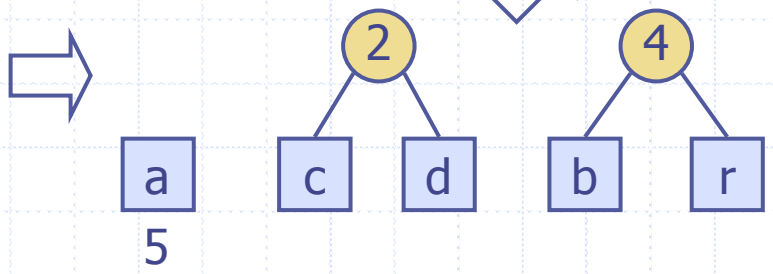
X = abracadabra

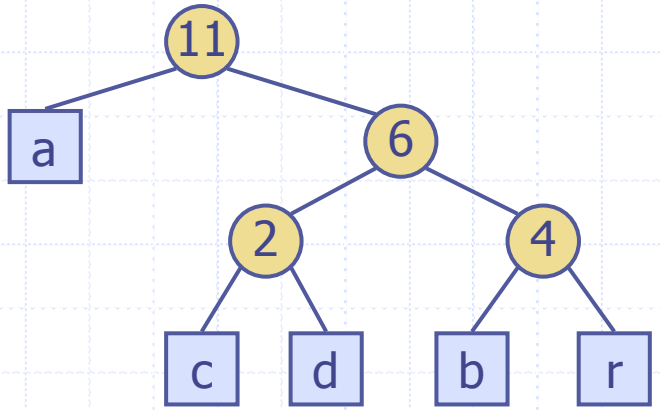
Freqüências

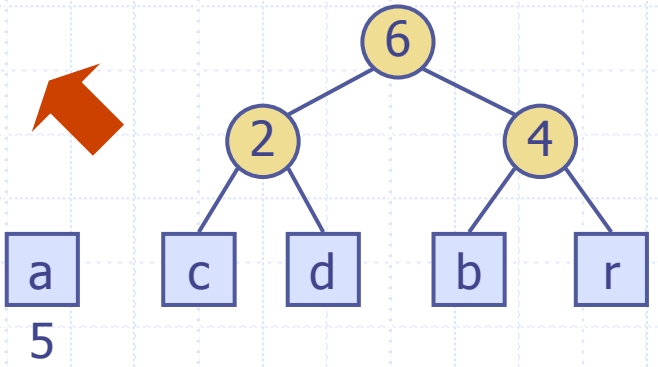
a	b	c	d	r
5	2	1	1	2











Árvore de Huffman Estendida - Exemplo

String: **a fast runner need never be afraid of the dark**

Character		a	b	d	e	f	h	i	k	n	o	r	s	t	u	v
Frequency	9	5	1	3	7	3	1	1	1	4	1	5	1	2	1	1