# Processamento de Cadeias de Caracteres*

Última alteração: 10 de Outubro de 2006

*Transparências elaboradas por Charles Ornelas, Fabiano Botelho, Leonardo Rocha, Leonardo Mata e Nivio Ziviani



## Definição e Motivação

- **Cadeia de caracteres**: seqüência de elementos denominados caracteres.
- Os caracteres são escolhidos de um conjunto denominado **alfabeto**.

  Ex.: em uma cadeia de *bits* o alfabeto é $\{0, 1\}$.
- **Casamento de cadeias de caracteres** ou **casamento de padrão**: encontrar todas as ocorrências de um padrão em um texto.
- Exemplos de aplicação:
  - edição de texto;
  - recuperação de informação;
  - estudo de seqüências de DNA em biologia computacional.



## Notação

- Texto: arranjo $T[0..n-1]$ de tamanho $n$;
- Padrão: arranjo $P[0..m-1]$ de tamanho $m \leq n$.
- Os elementos de $P$ e $T$ são escolhidos de um alfabeto finito $\Sigma$ de tamanho $c$.

  Ex.: $\Sigma = \{0, 1\}$ ou $\Sigma = \{a, b, \ldots, z\}$.
- **Casamento de cadeias** ou **casamento de padrão**: dados duas cadeias $P$ (padrão) de comprimento $|P| = m$ e $T$ (texto) de comprimento $|T| = n$, onde $n \gg m$, deseja-se saber as ocorrências de $P$ em $T$.



## Categorias de Algoritmos

- $P$ e $T$ não são pré-processados:
  - algoritmo seqüencial, on-line e de tempo-real;
  - padrão e texto não são conhecidos *a priori*.
  - complexidade de tempo $O(mn)$ e de espaço $O(1)$, para pior caso.
- $P$ pré-processado:
  - algoritmo seqüencial;
  - padrão conhecido anteriormente permitindo seu pré-processamento.
  - complexidade de tempo $O(n)$ e de espaço $O(m + c)$, no pior caso.
  - ex.: programas para edição de textos.

## Categorias de Algoritmos

- $P$ e $T$ são pré-processados:
  - algoritmo constrói índice.
  - complexidade de tempo $O(\log n)$ e de espaço é $O(n)$.
  - tempo para obter o índice é grande, $O(n)$ ou $O(n \log n)$.
  - compensado por muitas operações de pesquisa no texto.
  - Tipos de índices mais conhecidos:
    * Arquivos invertidos
    * Árvores *trie* e árvores Patricia
    * Arranjos de sufixos



## Exemplos: $P$ e $T$ são pré-processados

- Diversos tipos de índices: arquivos invertidos, árvores *trie* e Patricia, e arranjos de sufixos.
- Um **arquivo invertido** possui duas partes: **vocabulário** e **ocorrências**.
- O vocabulário é o conjunto de todas as palavras distintas no texto.
- Para cada palavra distinta, uma lista de posições onde ela ocorre no texto é armazenada.
- O conjunto das listas é chamado de ocorrências.
- As posições podem referir-se a palavras ou caracteres.



## Exemplo de Arquivo Invertido

0 6 15 21 25 35 44 52

Texto exemplo. Texto tem palavras. Palavras exercem fascínio.

| | |
|---|---|
| exemplo | 6 |
| exercem | 44 |
| fascínio | 52 |
| palavras | 25 35 |
| tem | 21 |
| texto | 0 15 |



## Arquivo Invertido - Tamanho

- O vocabulário ocupa pouco espaço.
- A previsão sobre o crescimento do tamanho do vocabulário é dada pela lei de Heaps.
- **Lei de Heaps:** o vocabulário de um texto em linguagem natural contendo $n$ palavras tem tamanho $V = Kn^{\beta} = O(n^{\beta})$, em que $K$ e $\beta$ dependem das características de cada texto.
- $K$ geralmente assume valores entre 10 e 100, e $\beta$ é uma constante entre 0 e 1, na prática ficando entre 0,4 e 0,6.
- O vocabulário cresce sublinearmente com o tamanho do texto, em uma proporção perto de sua raiz quadrada.
- As ocorrências ocupam muito mais espaço.
- Como cada palavra é referenciada uma vez na lista de ocorrências, o espaço necessário é $O(n)$.
- Na prática, o espaço para a lista de ocorrências fica entre 30% e 40% do tamanho do texto.

## Arquivo Invertido - Pesquisa

- A pesquisa tem geralmente três passos:
  - *Pesquisa no vocabulário*: palavras e padrões da consulta são isoladas e pesquisadas no vocabulário.
  - *Recuperação das ocorrências*: as listas de ocorrências das palavras encontradas no vocabulário são recuperadas.
  - *Manipulação das ocorrências*: as listas de ocorrências são processadas para tratar frases, proximidade, ou operações booleanas.
- Como a pesquisa em um arquivo invertido sempre começa pelo vocabulário, é interessante mantê-lo em um arquivo separado.
- Na maioria das vezes, esse arquivo cabe na memória principal.



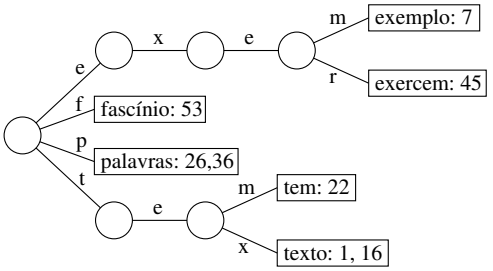
## Arquivo Invertido - Pesquisa

- A pesquisa de palavras simples pode ser realizada usando qualquer estrutura de dados que torne a busca eficiente, como *hashing*, árvore *trie* ou árvore B.
- As duas primeiras têm custo $O(m)$, onde $m$ é o tamanho da consulta (independentemente do tamanho do texto).
- Guardar as palavras na ordem lexicográfica é barato em termos de espaço e competitivo em desempenho, já que a pesquisa binária pode ser empregada com custo $O(\log n)$, sendo $n$ o número de palavras.
- A pesquisa por frases usando índices é mais difícil de resolver.
- Cada elemento da frase tem de ser pesquisado separadamente e suas listas de ocorrências recuperadas.
- A seguir, as listas têm de ser percorridas de forma sicronizada para encontrar as posições nas quais todas as palavras aparecem em seqüência.



## Arquivo Invertido Usando Trie

- Arquivo invertido usando uma **árvore *trie*** para o texto: `Texto exemplo. Texto tem palavras. Palavras exercem fascínio.`




- O vocabulário lido até o momento é colocado em uma árvore *trie*, armazenando uma lista de ocorrências para cada palavra.
- Cada nova palavra lida é pesquisada na *trie*:
  - Se a pesquisa é sem sucesso, então a palavra é inserida na árvore e uma lista de ocorrências é inicializada com a posição da nova palavra no texto.
  - Senão, uma vez que a palavra já se encontra na árvore, a nova posição é inserida ao final da lista de ocorrências.



## Casamento Exato

- Consiste em obter todas as ocorrências **exatas** do padrão no texto.

  Ex.: ocorrência exata do padrão `teste`.

```
   teste
os testes testam estes alunos . . .
```

- Dois enfoques:

1. leitura dos caracteres do texto um a um: algoritmos força bruta, Knuth-Morris-Pratt e Shift-And.
2. pesquisa de $P$ em uma janela que desliza ao longo de $T$, pesquisando por um sufixo da janela que casa com um sufixo de $P$, por comparações da direita para a esquerda: algoritmos Boyer-Moore-Horspool e Boyer-Moore.

## Casamento Exato - Métodos Considerados

- A classe $CasamentoExato$ apresenta a assinatura dos métodos de casamento exato implementados a seguir.
- $maxChar$ é utilizada para representar o tamanho do alfabeto considerado (caracteres ASCII).
- As cadeias de caracteres $T$ e $P$ são representadas por meio da classe ***String***.

```
package cap8;
public class CasamentoExato {
  private static final int maxChar = 256;
  // Assinatura dos métodos para casamento exato considerados
  public static void forcaBruta (String T, int n, String P, int m)
  public static void shiftAndExato (String T, int n, String P, int m)
  public static void bmh (String T, int n, String P, int m)
  public static void bmhs (String T, int n, String P, int m)
}
```



## Força Bruta - Implementação

- É o algoritmo mais simples para casamento de cadeias.
- A idéia é tentar casar qualquer subcadeia no texto de comprimento $m$ com o padrão.

```
public static void forcaBruta (String T, int n, String P, int m) {
  // Pesquisa P[0..m-1] em T[0..n-1]
  for (int i = 0; i < (n - m + 1); i ++) {
    int k = i; int j = 0;
    while ((j < m) && (T.charAt (k) == P.charAt (j))) { j++; k++; }
    if (j == m) System.out.println ("Casamento na posicao: " + i);
  }
}
```



## Força Bruta - Análise

- Pior caso: $C_n = m \times n$.
- O pior caso ocorre, por exemplo, quando $P =$ `aab` e $T =$`aaaaaaaaaa`.
- Caso esperado: $\overline{C_n} = \frac{c}{c-1}\left(1 - \frac{1}{c^m}\right)(n - m + 1) + O(1)$
- O caso esperado é muito melhor do que o pior caso.
- Em experimento com texto randômico e alfabeto de tamanho $c = 4$, o número esperado de comparações é aproximadamente igual a 1,3.



## Autômatos

- Um autômato é um modelo de computação muito simples.
- Um **autômato finito** é definido por uma tupla $(Q, I, F, \Sigma, \mathcal{T})$, onde $Q$ é um conjunto finito de estados, entre os quais existe um estado inicial $I \in Q$, e alguns são estados finais ou estados de término $F \subseteq Q$.
- Transições entre estados são rotuladas por elementos de $\Sigma \cup \{\epsilon\}$, em que $\Sigma$ é o alfabeto finito de entrada e $\epsilon$ é a transição vazia.
- As transições são formalmente definidas por uma função de transição $\mathcal{T}$.
- $\mathcal{T}$ associa a cada estado $q \in Q$ um conjunto $\{q_1, q_2, \ldots, q_k\}$ de estados de $Q$ para cada $\alpha \in \Sigma \cup \{\epsilon\}$.

## Tipos de Autômatos

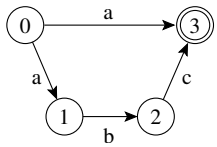
- **Autômato finito não-determinista**:
  - Quando $\mathcal{T}$ é tal que existe um estado $q$ associado a um dado caractere $\alpha$ para mais de um estado, digamos $\mathcal{T}(q, \alpha) = \{q_1, q_2, \ldots, q_k\}$, $k > 1$, ou existe alguma transição rotulada por $\epsilon$.
  - Neste caso, a função de transição $\mathcal{T}$ é definida pelo conjunto de triplas $\Delta = \{(q, \alpha, q')$, onde $q \in Q$, $\alpha \in \Sigma \cup \{\epsilon\}$, e $q' \in \mathcal{T}(q, \alpha)$.
- **Autômato finito determinista**:
  - Quando a função de transição $\mathcal{T}$ é definida pela função $\delta = Q \times \Sigma \cup \epsilon \rightarrow Q$.
  - Neste caso, se $\mathcal{T}(q, \alpha) = \{q'\}$, então $\delta(q, \alpha) = q'$.



## Exemplo de Autômatos

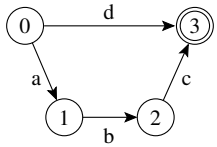
- Autômato finito não-determinista.

  A partir do estado 0, através do caractere de transição $a$ é possível atingir os estados 2 e 3.




- Autômato finito determinista.

  Para cada caractere de transição todos os estados levam a um único estado.

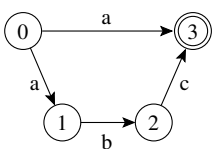





## Reconhecimento por Autômato

- Uma cadeia é **reconhecida** por $(Q, I, F, \Sigma, \Delta)$ ou $(Q, I, F, \Sigma, \delta)$ se qualquer um dos autômatos rotula um caminho que vai de um estado inicial até um estado final.
- A **linguagem reconhecida** por um autômato é o conjunto de cadeias que o autômato é capaz de reconher.

Ex.: a linguagem reconhecida pelo autômato abaixo é o conjunto de cadeias $\{a\}$ e $\{abc\}$ no estado 3.






## Transições Vazias

- São transições rotulada com uma cadeia vazia $\epsilon$, também chamadas de **transições-$\epsilon$**, em autômatos não-deterministas
- Não há necessidade de se ler um caractere para caminhar através de uma transição vazia.
- Simplificam a construção do autômato.
- Sempre existe um autômato equivalente que reconhece a mesma linguagem sem transições-$\epsilon$.

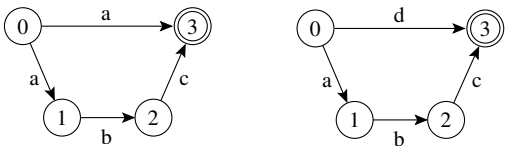
## Estados Ativos

- Se uma cadeia $x$ rotula um caminho de $I$ até um estado $q$ então o estado $q$ é considerado ativo depois de ler $x$.
- Um autômato finito determinista tem no máximo um estado ativo em um determinado instante.
- Um autômato finito não-determinista pode ter vários estados ativos.
- Casamento aproximado de cadeias pode ser resolvido por meio de autômatos finitos não-deterministas.



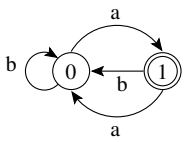
## Ciclos em Autômatos

- Os autômatos abaixo são **acíclicos** pois as transições não formam ciclos.




- **Autômatos finitos cíclicos**, deterministas ou não-deterministas, são úteis para **casamento de expressões regulares**
- A linguagem reconhecida por um autômato cíclico pode ser infinita.

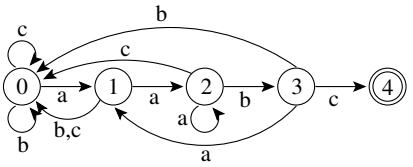
Ex: o autômato abaixo reconhece `ba`, `bba`, `bbba`, `bbbba`, e assim por diante.






## Exemplo de Uso de Autômato

- O autômato abaixo reconhece $P$ ={`aabc`}.




- A pesquisa de $P$ sobre um texto $T$ com alfabeto $\Sigma$ ={`a, b, c`} pode ser vista como a simulação do autômato na pesquisa de $P$ sobre $T$.
- No início, o estado inicial ativa o estado 1.
- Para cada caractere lido do texto, a aresta correspondente é seguida, ativando o estado destino.
- Se o estado 4 estiver ativo e um caractere `c` é lido o estado final se torna ativo, resultando em um casamento de `aabc` com o texto.
- Como cada caractere do texto é lido uma vez, a complexidade de tempo é $O(n)$, e de espaço é $m + 2$ para vértices e $|\Sigma| \times m$ para arestas.

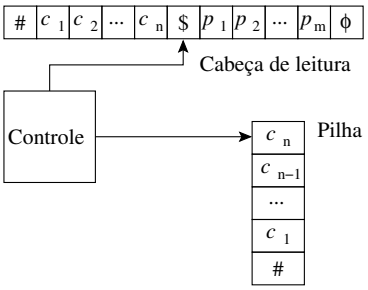


## Knuth-Morris-Pratt (KMP)

- O KMP é o primeiro algoritmo (1977) cujo pior caso tem complexidade de tempo linear no tamanho do texto, $O(n)$.
- É um dos algoritmos mais famosos para resolver o problema de casamento de cadeias.
- Tem implementação complicada e na prática perde em eficiência para o Shift-And e o Boyer-Moore-Horspool.
- Até 1971, o limite inferior conhecido para busca exata de padrões era $O(mn)$.

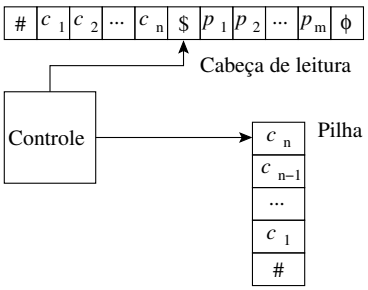
## KMP - 2DPDA

- Em 1971, Cook provou que qualquer problema que puder ser resolvido por um autômato determinista de dois caminhos com memória de pilha (*Two-way Deterministic Pushdown Store Automaton*, 2DPDA) pode ser resolvido em tempo linear por uma máquina RAM.
- O 2DPDA é constituído de:
  - uma fita apenas para leitura;
  - uma pilha de dados (memória temporária);
  - um controle de estado que permite mover a fita para esquerda ou direita, empilhar ou desempilhar símbolos, e mudar de estado.






## KMP - Casamento de cadeias no 2DPDA




- No autômato da acima, a entrada é constituída da cadeia $\#c_1c_2 \cdots c_n\$p_1p_2 \cdots p_m\phi$.
- A partir de $\#$ todos os caracteres são empilhados até encontrar o caractere $\$$.
- A leitura cotinua até encontrar o caractere $\phi$.
- A seguir a leitura é realizada no sentido contrário, iniciando por $p_n$, comparado-o com o último caractere empilhado, no caso $c_n$.
- Esta operação é repetida para os caracteres seguintes, e se o caractere $\$$ for atingido então as duas cadeias são iguais.



## KMP - Algoritmo

- Primeira versão do KMP é uma simulação linear do 2DPDA
- O algoritmo computa o sufixo mais longo no texto que é também o prefixo de $P$.
- Quando o comprimento do sufixo no texto é igual a $|P|$ ocorre um casamento.
- O pré-processamento de $P$ permite que nenhum caractere seja reexaminado.
- O apontador para o texto nunca é decrementado.
- O pré-processamento de $P$ pode ser visto como a construção econômica de um autômato determinista que depois é usado para pesquisar pelo padrão no texto.



## Shift-And

- O Shift-And é vezes mais rápido e muito mais simples do que o KMP.
- Pode ser estendido para permitir casamento aproximado de cadeias de caracteres.
- Usa o conceito de **paralelismo de bit**:
  - técnica que tira proveito do paralelismo intrínseco das operações sobre *bits* dentro de uma palavra de computador.
  - É possível empacotar muitos valores em uma única palavra e atualizar todos eles em uma única operação.
- Tirando proveito do paralelismo de *bit*, o número de operações que um algoritmo realiza pode ser reduzido por um fator de até $w$, onde $w$ é o número de *bits* da palavra do computador.

## Shift-And - Notação para Operações Usando Paralelismo de *bit*

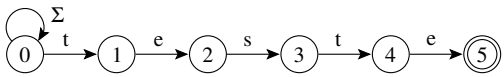
- Para denotar **repetição de *bit*** é usado exponenciação: $01^3 = 0111$.
- Uma seqüência de *bits* $b_0 \dots b_{c-1}$ é chamada de **máscara de bits** de comprimento $c$, e é armazenada em alguma posição de uma palavra $w$ do computador.
- Operações sobre os *bits* da palavra do computador:
  - "|": operação *or*;
  - "&": operação *and*;
  - "$\sim$": complementa todos os *bits*;
  - "$>>$": move os *bits* para a direita e entra com zeros à esquerda (por exemplo, $b_0, b_1, \dots, b_{c-2}, b_{c-1} >> 2 = 00b_0, b_1, \dots, b_{c-3}$).



## Shift-And - Princípio de Funcionamento

- Mantém um conjunto de todos os prefixos de $P$ que casam com o texto já lido.
- Utiliza o paralelismo de *bit* para atualizar o conjunto a cada caractere lido do texto.
- Este conjunto é representado por uma máscara de *bits* $R = (b_0, b_1, \dots, b_{m-1})$.
- O algoritmo Shift-And pode ser visto como a simulação de um autômato que pesquisa pelo padrão no texto (não-determinista para simular o paralelismo de *bit*).

Ex.: Autômato não-determinista que reconhece todos os prefixos de $P =$`{teste}`






## Shift-And - Algoritmo

- O valor 1 é colocado na $j$-ésima posição de $R = (b_0, b_1, \dots, b_{m-1})$ se e somente se $p_0 \dots p_j$ é um sufixo de $t_0 \dots t_i$, onde $i$ corresponde à posição corrente no texto.
- A $j$-ésima posição de $R$ é dita estar *ativa*.
- $b_{m-1}$ ativo significa um casamento.
- $R'$, o novo valor do conjunto $R$, é calculado na leitura do próximo caractere $t_{i+1}$.
- A posição $j+1$ no conjunto $R'$ ficará ativa se e somente se a posição $j$ estiver ativa em $R$ e $t_{i+1}$ casa com $p_{i+1}$ ($p_0 \dots p_j$ era um sufixo de $t_0 \dots t_i$ e $t_{i+1}$ casa com $p_{j+1}$).
- Com o uso de paralelismo de *bit* é possível computar o novo conjunto com custo $O(1)$.



## Shift-And - Pré-processamento

- O primeiro passo é a construção de uma tabela $M$ para armazenar uma máscara de *bits* $b_0 \dots, b_{m-1}$ para cada caractere.

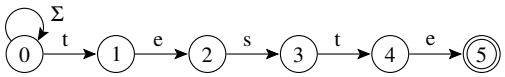
  Ex.: máscaras de *bits* para os caracteres presentes em $P =$`{teste}`.

| | 0 | 1 | 2 | 3 | 4 |
|---|---|---|---|---|---|
| M[t] | 1 | 0 | 0 | 1 | 0 |
| M[e] | 0 | 1 | 0 | 0 | 1 |
| M[s] | 0 | 0 | 1 | 0 | 0 |

- A máscara em $M[\mathtt{t}]$ é $10010$, pois o caractere `t` aparece nas posições 0 e 3 de $P$.

## Shift-And - Pesquisa

- O valor do conjunto é inicializado como $R = 0^m$ ($0^m$ significa 0 repetido $m$ vezes).
- Para cada novo caractere $t_{i+1}$ lido do texto o valor do conjunto $R'$ é atualizado: $R' = ((R >> 1) \mid 10^{m-1}) \,\&\, M[T[i]]$.
- A operação "$>>$" desloca todas as posições para a direita no passo $i + 1$ para marcar quais posições de $P$ eram sufixos no passo $i$.
- A cadeia vazia $\epsilon$ também é marcada como um sufixo, permitindo um casamento na posição corrente do texto (*self-loop* no início do autômato).




- Do conjunto obtido até o momento, são mantidas apenas as posições que $t_{i+1}$ casa com $p_{j+1}$, obtido com a operação *and* desse conjunto de posições com o conjunto $M[t_{i+1}]$ de posições de $t_{i+1}$ em $P$.



## Exemplo de funcionamento do Shif-And

Pesquisa do padrão $P =$`{teste}` no texto $T =$`{os testes ...}`.

| Texto | $(R >> 1) \mid 10^{m-1}$ | | | | | $R'$ | | | | |
|---|---|---|---|---|---|---|---|---|---|---|
| o | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| s | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| t | 1 | 0 | 0 | 0 | 0 | 1 | 0 | 0 | 0 | 0 |
| e | 1 | 1 | 0 | 0 | 0 | 0 | 1 | 0 | 0 | 0 |
| s | 1 | 0 | 1 | 0 | 0 | 0 | 0 | 1 | 0 | 0 |
| t | 1 | 0 | 0 | 1 | 0 | 1 | 0 | 0 | 1 | 0 |
| e | 1 | 1 | 0 | 0 | 1 | 0 | 1 | 0 | 0 | ***1*** |
| s | 1 | 0 | 1 | 0 | 0 | 0 | 0 | 1 | 0 | 0 |
| | 1 | 0 | 0 | 1 | 0 | 0 | 0 | 0 | 0 | 0 |



## Shift-And - Implementação

**void** Shift-And ($P = p_0p_1 \ldots p_{m-1}$, $T = t_0t_1 \ldots t_{n-1}$)
  *// Pré-processamento*
  **for** ($c \in \Sigma$) $M[c] = 0^m$;
  **for** (j = 0; j < m; j++) $M[p_j] = M[p_j] \mid 0^j10^{m-j-1}$;
  *// Pesquisa*
  $R = 0^m$;
  **for** (i = 0; i < n; i++)
    $R = ((R >> 1) \mid 10^{m-1}) \,\&\, M[T[i]]$;
    **if** ($R \,\&\, 0^{m-1}1 \neq 0^m$) "Casamento na posicao $i - m + 1$";

- **Análise**: O custo do algoritmo Shift-And é $O(n)$, desde que as operações possam ser realizadas em $O(1)$ e o padrão caiba em umas poucas palavras do computador.



## Boyer-Moore-Horspool (BMH)

- Em 1977, foi publicado o algoritmo Boyer-Moore (BM).
- A idéia é pesquisar no padrão no sentido da direita para a esquerda, o que torna o algoritmo muito rápido.
- Em 1980, Horspool apresentou uma simplificação no algoritmo original, tão eficiente quanto o algoritmo original, ficando conhecida como Boyer-Moore-Horspool (BMH).
- Pela extrema simplicidade de implementação e comprovada eficiência, o BMH deve ser escolhido em aplicações de uso geral que necessitam realizar casamento exato de cadeias.

## Funcionamento do BM e BMH

- O BM e BMH pesquisa o padrão $P$ em uma janela que desliza ao longo do texto $T$.
- Para cada posição desta janela, o algoritmo pesquisa por um sufixo da janela que casa com um sufixo de $P$, com comparações realizadas no sentido da direita para a esquerda.
- Se não ocorrer uma desigualdade, então uma ocorrência de $P$ em $T$ ocorreu.
- Senão, o algoritmo calcula um deslocamento que o padrão deve ser deslizado para a direita antes que uma nova tentativa de casamento se inicie.
- O BM original propõe duas heurísticas para calcular o deslocamento: ocorrência e casamento.



## BM - Heurística Ocorrência

- Alinha a posição no texto que causou a colisão com o primeiro caractere no padrão que casa com ele;

Ex.: $P$ ={cacbac}, $T$ ={aabcaccacbac}.

```
0 1 2 3 4 5 6 7 8 9 0 1
c a c b a c
a a b c a c c a c b a c
  c a c b a c
    c a c b a c
      c a c b a c
        c a c b a c
```

- A partir da posição 5, da direita para a esquerda, existe uma colisão na posição 3 de $T$, entre `b` do padrão e `c` do texto.
- Logo, o padrão deve ser deslocado para a direita até o primeiro caractere no padrão que casa com `c`.
- O processo é repetido até encontrar um casamento a partir da posição 6 de $T$.



## BM - Heurística Casamento

- Ao mover o padrão para a direita, faça-o casar com o pedaço do texto anteriormente casado.

Ex.: $P$ ={cacbac} no texto $T$ ={aabcaccacbac}.

```
0 1 2 3 4 5 6 7 8 9 0 1
c a c b a c
a a b c a c c a c b a c
    c a c b a c
      c a c b a c
```

- Novamente, a partir da posição 5, da direita para a esquerda, existe uma colisão na posição 3 de $T$, entre o `b` do padrão e o `c` do texto.
- Neste caso, o padrão deve ser deslocado para a direita até casar com o pedaço do texto anteriormente casado, no caso `ac`, deslocando o padrão 3 posições à direita.
- O processo é repetido mais uma vez e o casamento entre $P$ e $T$ ocorre.



## Escolha da Heurística

- O algoritmo BM escolhe a heurística que provoca o maior deslocamento do padrão.
- Esta escolha implica em realizar uma comparação entre dois inteiros para cada caractere lido do texto, penalizando o desempenho do algoritmo com relação a tempo de processamento.
- Várias propostas de simplificação ocorreram ao longo dos anos.
- As que produzem os melhores resultados são as que consideram apenas a heurística ocorrência.

## Algoritmo Boyer-Moore-Horspool (BMH)

- A simplificação mais importante é devida a Horspool em 1980.
- Executa mais rápido do que o algoritmo BM original.
- Parte da observação de que qualquer caractere já lido do texto a partir do último deslocamento pode ser usado para endereçar a tabela de deslocamentos.
- Endereça a tabela com o caractere no texto correspondente ao último caractere do padrão.



## BMH - Tabela de Deslocamentos

- Para pré-computar o padrão o valor inicial de todas as entradas na tabela de deslocamentos é feito igual a $m$.
- A seguir, apenas para os $m-1$ primeiros caracteres do padrão são usados para obter os outros valores da tabela.
- Formalmente, $d[x] = min\{j \text{ tal que } j = m \mid (1 \leq j < m \ \& \ P[m-j-1] = x)\}$.

Ex.: Para o padrão $P =$`{teste}`, os valores de $d$ são $d[\texttt{t}] = 1$, $d[\texttt{e}] = 3$, $d[\texttt{s}] = 2$, e todos os outros valores são iguais ao valor de $|P|$, nesse caso $m = 5$.



## BMH - Implementação

```
public static void bmh (String T, int n, String P, int m) {
  // Pré-processamento do padrão
  int d[] = new int[maxChar];
  for (int j = 0; j < maxChar; j++) d[j] = m;
  for (int j = 0; j < (m-1); j++) d[(int)P.charAt (j)] = m - j - 1;
  int i = m - 1;
  while (i < n) { // Pesquisa
    int k = i; int j = m - 1;
    while ((j >= 0) && (T.charAt (k) == P.charAt (j))) { j--;k--;}
    if (j < 0)
      System.out.println ("Casamento na posicao: " + (k + 1));
    i = i + d[(int)T.charAt (i)];
  }
}
```

- $d[(int)T.charAt(i)]$ equivale ao endereço na tabela $d$ do caractere que está na $i$-ésima posição no texto, a qual corresponde à posição do último caractere de $P$.



## Algoritmo BMHS - Boyer-Moore-Horspool-Sunday

- Sunday (1990) apresentou outra simplificação importante para o algoritmo BM, ficando conhecida como BMHS.
- Variante do BMH: endereçar a tabela com o caractere no texto correspondente ao próximo caractere após o último caractere do padrão, em vez de deslocar o padrão usando o último caractere como no algoritmo BMH.
- Para pré-computar o padrão, o valor inicial de todas as entradas na tabela de deslocamentos é feito igual a $m+1$.
- A seguir, os $m$ primeiros caracteres do padrão são usados para obter os outros valores da tabela.
- Formalmente $d[x] = min\{j \text{ tal que } j = m \mid (1 \leq j \leq m \ \& \ P[m-j] = x)\}$.
- Para o padrão $P =$ `teste`, os valores de $d$ são $d[\texttt{t}] = 2$, $d[\texttt{e}] = 1$, $d[\texttt{s}] = 3$, e todos os outros valores são iguais ao valor de $|P|+1$.

## BMHS - Implementação

- O pré-processamento do padrão ocorre nas duas primeiras linhas do código.
- A fase de pesquisa é constituída por um anel em que $i$ varia de $m-1$ até $n-1$, com incrementos $d[(int)T.charAt(i+1)]$, o que equivale ao endereço na tabela $d$ do caractere que está na $i+1$-ésima posição no texto, a qual corresponde à posição do último caractere de $P$.

```
public static void bmhs (String T, int n, String P, int m) {
  // Pré-processamento do padrão
  int d[] = new int[maxChar];
  for (int j = 0; j < maxChar; j++) d[j] = m + 1;
  for (int j = 0; j < m; j++) d[(int)P.charAt (j)] = m - j;
  int i = m - 1;
  while (i < n) { // Pesquisa
    int k = i; int j = m - 1;
    while ((j >= 0) && (T.charAt (k) == P.charAt (j))) { j--;k--;}
    if (j < 0)
      System.out.println ("Casamento na posicao: " + (k + 1));
    i = i + d[(int)T.charAt (i+1)];
  }
}
```



## BH - Análise

- Os dois tipos de deslocamento (ocorrência e casamento) podem ser pré-computados com base apenas no padrão e no alfabeto.
- Assim, a complexidade de tempo e de espaço para esta fase é $O(m+c)$.
- O pior caso do algoritmo é $O(nm)$.
- O melhor caso e o caso médio para o algoritmo é $O(n/m)$, um resultado excelente pois executa em tempo sublinear.



## BMH - Análise

- O deslocamento ocorrência também pode ser pré-computado com base apenas no padrão e no alfabeto.
- A complexidade de tempo e de espaço para essa fase é $O(m+c)$.
- Para a fase de pesquisa, o pior caso do algoritmo é $O(nm)$, o melhor caso é $O(n/m)$ e o caso esperado é $O(n/m)$, se $c$ não é pequeno e $m$ não é muito grande.



## BMHS - Análise

- Na variante BMHS, seu comportamento assintótico é igual ao do algoritmo BMH.
- Entretanto, os deslocamentos são mais longos (podendo ser iguais a $m+1$), levando a saltos relativamente maiores para padrões curtos.
- Por exemplo, para um padrão de tamanho $m=1$, o deslocamento é igual a $2m$ quando não há casamento.

## Casamento Aproximado

- O casamento aproximado de cadeias permite operações de inserção, substituição e retirada de caracteres do padrão.

Ex.: Três ocorrências do padrão `teste` em que os casos de inserção, substituição, retirada de caracteres no padrão acontecem:

1. um espaço é inserido entre o terceiro e quarto caracteres do padrão;
2. o último caractere do padrão é substituído pelo caractere `a`;
3. o primeiro caractere do padrão é retirado.

```
    tes te
        testa
              este
os testes testam estes alunos . . .
```



## Distância de Edição

- É número $k$ de operações de inserção, substituição e retirada de caracteres necessário para transformar uma cadeia $x$ em outra cadeia $y$.
- $ed(P, P')$: distância de edição entre duas cadeias $P$ e $P'$; é o menor número de operações necessárias para converter $P$ em $P'$, ou vice versa.

  Ex.: $ed(\texttt{t}este, \texttt{e}stende) = 4$, valor obtido por meio de uma retirada do primeiro `t` de $P$ e a inserção dos 3 caracteres `nde` ao final de $P$.
- O problema do casamento aproximado de cadeias é o de encontrar todas as ocorrências em $T$ de cada $P'$ que satisfaz $ed(P, P') \leq k$.



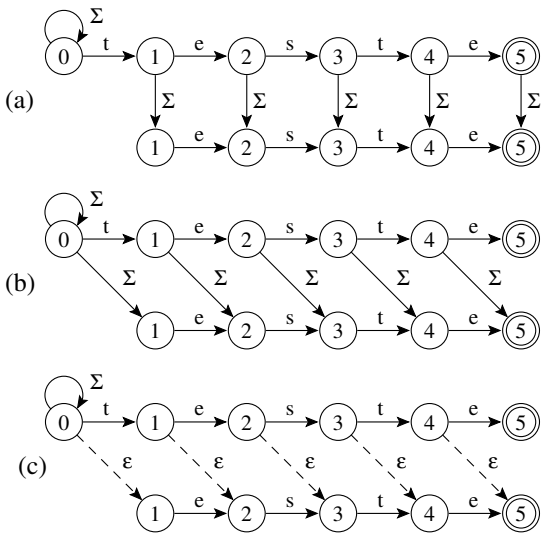
## Casamento Aproximado

- A busca aproximada só faz sentido para $0 < k < m$, pois para $k = m$ toda subcadeia de comprimento $m$ pode ser convertida em $P$ por meio da substituição de $m$ caracteres.
- O caso em que $k = 0$ corresponde ao casamento exato de cadeias.
- O nível de erro $\alpha = k/m$, fornece uma medida da fração do padrão que pode ser alterado.
- Em geral $\alpha < 1/2$ para a maioria dos casos de interesse.
- **Casamento aproximado de cadeias**, ou **casamento de cadeias permitindo erros**: um número limitado $k$ de operações (erros) de inserção, de substituição e de retirada é permitido entre $P$ e suas ocorrências em $T$.
- A pesquisa com casamento aproximado é modelado por autômato não-determinista.
- O algoritmo de casamento aproximado de cadeias usa o **paralelismo de bit**.



## Exemplo de Autômato para Casamento Aproximado

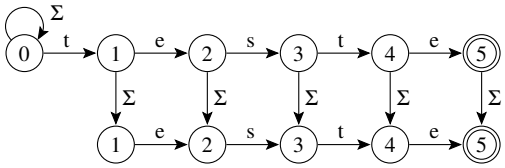
- $P$ ={`teste`} e $k = 1$: (a)inserção; (b)substituição e (c)retirada.



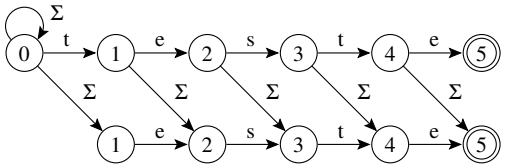

- Casamento de caractere é representado por uma aresta horizontal. Avançamos em $P$ e $T$.
- O *self-loop* permite que uma ocorrência se inicie em qualquer posição em $T$.

## Exemplo de Autômato para Casamento Aproximado

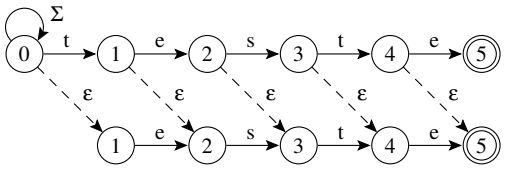
- Uma aresta vertical insere um caractere no padrão. Avançamos em $T$ mas não em $P$.




- Uma aresta diagonal sólida substitui um caractere. Avançamos em $T$ e $P$.



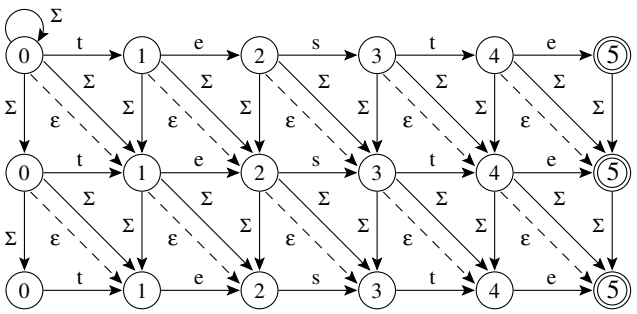

- Uma aresta diagonal tracejada retira um caractere. Avançamos em $P$ mas não em $T$ (transição-$\epsilon$)






## Exemplo de Autômato para Casamento Aproximado

- $P =$ {`teste`} e $K = 2$.
- As três operações de distância de edição estão juntas em um único autômato:
  - Linha 1: casamento exato ($k = 0$);
  - Linha 2: casamento aproximado permitindo um erro ($k = 1$);
  - Linha 3: casamento aproximado permitindo dois erros ($k = 2$).




- Uma vez que um estado no autômato está ativo, todos os estados nas linhas seguintes na mesma coluna também estão ativos.



## Shift-And para Casamento Aproximado

- Utiliza **paralelismo de bit**.
- Simula um autômato não-determinista.
- Empacota cada linha $j$ ($0 < j \leq k$) do autômato não-determinista em uma palavra $R_j$ diferente do computador.
- Para cada novo caractere lido do texto todas as transições do autômato são simuladas usando operações entre as $k + 1$ máscaras de *bits*.
- Todas as $k + 1$ máscaras de *bits* têm a mesma estrutura e assim o mesmo *bit* é alinhado com a mesma posição no texto.



## Shift-And para Casamento Aproximado

- Na posição $i$ do texto, os novos valores $R'_j$, $0 < j \leq k$, são obtidos a partir dos valores correntes $R_j$:
  - $R'_0 = ((R_0 >> 1) \mid 10^{m-1}) \ \& \ M[T[i]]$
  - $R'_j = ((R_j >> 1) \ \& \ M[T[i]]) \mid R_{j-1} \mid (R_{j-1} >> 1) \mid (R'_{j-1} >> 1) \mid 10^{m-1}$, onde $M$ é a tabela do algoritmo Shift-And para casamento exato.
- A pesquisa inicia com $R_j = 1^j 0_{m-j}$.
- $R_0$ equivale ao algoritmo Shift-And para casamento exato.
- As outras linhas $R_j$ recebem 1s (estados ativos) também de linhas anteriores.
- Considerando um automato para casamento aproximado, a fórmula para $R'$ expressa:
  - arestas horizontais indicando casamento;
  - verticais indicando inserção;
  - diagonais cheias indicando substituição;
  - diagonais tracejadas indicando retirada.

## Shif-And para Casamento Aproximado - Implementação

**void** Shift-And-Aproximado ($P = p_0p_1 \ldots p_{m-1}$, $T = t_0t_1 \ldots t_{n-1}, k$)
  *// Pré-processamento*
  **for** ($c \in \Sigma$) $M[c]$ = $0^m$;
  **for** (j = 0; j < m; j++) $M[p_j]$ = $M[p_j] \mid 0^j10^{m-j-1}$;
  *// Pesquisa*
  **for** (j = 0; j <= k; j++) $R_j$ = $1^j0^{m-j}$;
  **for** (i = 0; i < n; i++)
    Rant = $R_0$;
    Rnovo = ((Rant >> 1) | $10^{m-1}$) & $M[T[i]]$;
    $R_0$ = Rnovo;
    **for** (j = 1; j <= k; j++)
      Rnovo = (($R_j$ >> 1 & $M[T[i]]$) | Rant | ((Rant | Rnovo) >> 1);
      Rant = $R_j$;
      $R_j$ = Rnovo | $10^{m-1}$;
    **if** (Rnovo & $0^{m-1}1 \neq 0^m$) "Casamento na posicao $i$";



## Shif-And p/ Casam. Aprox. - Exemplo

- Padrão: `teste`. Texto: `os testes testam`. Permitindo um erro ($k = 1$) de inserção).
- $R'_0 = (R_0 >> 1)|10^{m-1}\&M[T[i]]$ e $R'_1 = (R_1 >> 1)\&M[T[i]]|R_0|(10^{m-1})$
- Uma ocorrência exata na posição 8 ("e") e duas, permitindo uma inserção, nas posições 8 e 11 ("s" e "e", respectivamente).

| Texto | $(R_0 >> 1)\|10^{m-1}$ | $R'_0$ | $R_1 >> 1$ | $R'_1$ |
|---|---|---|---|---|
| o | 1 0 0 0 0 | 0 0 0 0 0 | 0 1 0 0 0 | 1 0 0 0 0 |
| s | 1 0 0 0 0 | 0 0 0 0 0 | 0 1 0 0 0 | 1 0 0 0 0 |
| | 1 0 0 0 0 | 0 0 0 0 0 | 0 1 0 0 0 | 1 0 0 0 0 |
| t | 1 0 0 0 0 | 1 0 0 0 0 | 0 1 0 0 0 | 1 0 0 0 0 |
| e | 1 1 0 0 0 | 0 1 0 0 0 | 0 1 0 0 0 | 1 1 0 0 0 |
| s | 1 0 1 0 0 | 0 0 1 0 0 | 0 1 1 0 0 | 1 1 1 0 0 |
| t | 1 0 0 1 0 | 1 0 0 1 0 | 0 1 1 1 0 | 1 0 1 1 0 |
| e | 1 1 0 0 1 | 0 1 0 0 ***1*** | 0 1 0 1 1 | 1 1 0 1 ***1*** |
| s | 1 0 1 0 0 | 0 0 1 0 0 | 0 1 1 0 1 | 1 1 1 0 ***1*** |
| | 1 0 0 0 0 | 0 0 0 0 0 | 0 1 1 1 0 | 1 0 1 0 0 |
| t | 1 0 0 0 0 | 1 0 0 0 0 | 0 1 0 1 0 | 1 0 0 1 0 |
| e | 1 1 0 0 0 | 0 1 0 0 0 | 0 1 0 0 1 | 1 1 0 0 ***1*** |
| s | 1 0 1 0 0 | 0 0 1 0 0 | 0 1 1 0 0 | 1 1 1 0 0 |
| t | 1 0 0 1 0 | 1 0 0 1 0 | 0 1 1 1 0 | 1 0 1 1 0 |
| a | 1 1 0 0 1 | 0 0 0 0 0 | 0 1 0 1 1 | 1 0 0 1 0 |
| m | 1 0 0 0 0 | 0 0 0 0 0 | 0 1 0 0 1 | 1 0 0 0 0 |



## Shif-And p/ Casam. Aprox. - Exemplo

- Padrão: `teste`. Texto: `os testes testam`. Permitindo um erro de inserção, um de retirada e um de substituição.
- $R'_0 = (R_0 >> 1)|10^{m-1}\&M[T[i]]$ e $R'_1 = (R_1 >> 1)\&M[T[i]]|R_0|(R'_0 >> 1)|(R_0 >> 1)|(10^{m-1})$.
- Uma ocorrência exata na posição 8 ("e") e cinco, permitindo um erro, nas posições 6, 8, 11, 13 e 14 ("t", "s", "e", "t" e "a", respec.).

| Texto | $(R_0 >> 1)\|10^{m-1}$ | $R'_0$ | $R_1 >> 1$ | $R'_1$ |
|---|---|---|---|---|
| o | 1 0 0 0 0 | 0 0 0 0 0 | 0 1 0 0 0 | 1 0 0 0 0 |
| s | 1 0 0 0 0 | 0 0 0 0 0 | 0 1 0 0 0 | 1 0 0 0 0 |
| | 1 0 0 0 0 | 0 0 0 0 0 | 0 1 0 0 0 | 1 0 0 0 0 |
| t | 1 0 0 0 0 | 1 0 0 0 0 | 0 1 0 0 0 | 1 1 0 0 0 |
| e | 1 1 0 0 0 | 0 1 0 0 0 | 0 1 1 0 0 | 1 1 1 0 0 |
| s | 1 0 1 0 0 | 0 0 1 0 0 | 0 1 1 1 0 | 1 1 1 1 0 |
| t | 1 0 0 1 0 | 1 0 0 1 0 | 0 1 1 1 1 | 1 1 1 1 ***1*** |
| e | 1 1 0 0 1 | 0 1 0 0 ***1*** | 0 1 1 1 1 | 1 1 1 1 ***1*** |
| s | 1 0 1 0 0 | 0 0 1 0 0 | 0 1 1 1 1 | 1 1 1 1 ***1*** |
| | 1 0 0 0 0 | 0 0 0 0 0 | 0 1 1 1 1 | 1 0 1 1 0 |
| t | 1 0 0 0 0 | 1 0 0 0 0 | 0 1 0 1 1 | 1 1 0 1 0 |
| e | 1 1 0 0 0 | 0 1 0 0 0 | 0 1 1 0 1 | 1 1 1 0 ***1*** |
| s | 1 0 1 0 0 | 0 0 1 0 0 | 0 1 1 1 0 | 1 1 1 1 0 |
| t | 1 0 0 1 0 | 1 0 0 1 0 | 0 1 1 1 1 | 1 1 1 1 ***1*** |
| a | 1 1 0 0 1 | 0 0 0 0 0 | 0 1 1 1 1 | 1 1 0 1 ***1*** |
| m | 1 0 0 0 0 | 0 0 0 0 0 | 0 1 1 0 1 | 1 0 0 0 0 |



## Compressão - Motivação

- Explosão de informação textual disponível *on-line*:
  - Bibliotecas digitais.
  - Sistemas de automação de escritórios.
  - Bancos de dados de documentos.
  - World-Wide Web.
- Somente a Web tem hoje bilhões de páginas estáticas disponíveis.
- Cada bilhão de páginas ocupando aproximadamente 10 *terabytes* de texto corrido.
- Em setembro de 2003, a máquina de busca Google (www.google.com.br) dizia ter mais de 3,5 bilhões de páginas estáticas em seu banco de dados.

## Características necessárias para sistemas de recuperação de informação

- Métodos recentes de compressão têm permitido:
  1. Pesquisar diretamente o texto comprimido mais rapidamente do que o texto original.
  2. Obter maior compressão em relação a métodos tradicionais, gerando maior economia de espaço.
  3. Acessar diretamente qualquer parte do texto comprimido sem necessidade de descomprimir todo o texto desde o início (Moura, Navarro, Ziviani e Baeza-Yates, 2000; Ziviani, Moura, Navarro e Baeza-Yates, 2000).
- Compromisso espaço X tempo:
  - vencer-vencer.



## Porque Usar Compressão

- **Compressão de texto -** maneiras de representar o texto original em menos espaço:
  - Substituir os símbolos do texto por outros que possam ser representados usando um número menor de *bits* ou *bytes*.
- **Ganho obtido:** o texto comprimido ocupa menos espaço de armazenamento $\Rightarrow$ menos tempo para ser lido do disco ou ser transmitido por um canal de comunicação e para ser pesquisado.
- **Preço a pagar:** custo computacional para codificar e decodificar o texto.
- **Avanço da tecnologia:** De acordo com Patterson e Hennessy (1995), em 20 anos, o tempo de acesso a discos magnéticos tem se mantido praticamente constante, enquanto a velocidade de processamento aumentou aproximadamente 2 mil vezes $\Rightarrow$ melhor investir mais poder de computação em compressão em troca de menos espaço em disco ou menor tempo de transmissão.



## Razão de Compressão

- Definida pela porcentagem que o arquivo comprimido representa em relação ao tamanho do arquivo não comprimido.
- **Exemplo:** se o arquivo não comprimido possui 100 *bytes* e o arquivo comprimido resultante possui 30 *bytes*, então a razão de compressão é de 30%.
- Utilizada para medir O ganho em espaço obtido por um método de compressão.



## Outros Importantes Aspectos a Considerar

Além da economia de espaço, deve-se considerar:

- Velocidade de compressão e de descompressão.
- Possibilidade de realizar **casamento de cadeias** diretamente no texto comprimido.
- Permitir acesso direto a qualquer parte do texto comprimido e iniciar a descompressão a partir da parte acessada:

  **Um sistema de recuperação de informação para grandes coleções de documentos que estejam comprimidos necessitam acesso direto a qualquer ponto do texto comprimido.**

## Compressão de Textos em Linguagem Natural

- Um dos métodos de codificação mais conhecidos é o de **Huffman** (1952):
  - A idéia do método é atribuir códigos mais curtos a símbolos com freqüências altas.
  - Um código único, de tamanho variável, é atribuído a cada símbolo diferente do texto.
  - As implementações tradicionais do método de Huffman consideram caracteres como símbolos.
- Para aliar as necessidades dos algoritmos de compressão às necessidades dos sistemas de recuperação de informação apontadas acima, deve-se considerar palavras como símbolos a serem codificados.
- Métodos de Huffman baseados em caracteres comprimem o texto para aproximadamente 60%.
- Métodos de Huffman baseados em palavras comprimem o texto para valores pouco acima de 25%.



## Vantagens dos Métodos de Huffman Baseados em Palavras

- Permitem acesso randômico a palavras dentro do texto comprimido.
- Considerar palavras como símbolos significa que a tabela de símbolos do codificador é exatamente o vocabulário do texto.
- Isso permite uma integração natural entre o método de compressão e o arquivo invertido.
- Permitem acessar diretamente qualquer parte do texto comprimido sem necessidade de descomprimir todo o texto desde o início.



## Família de Métodos de Compressão Ziv-Lempel

- Substitui uma seqüência de símbolos por um apontador para uma ocorrência anterior daquela seqüência.
- A compressão é obtida porque os apontadores ocupam menos espaço do que a seqüência de símbolos que eles substituem.
- Os métodos Ziv-Lempel são populares pela sua velocidade, economia de memória e generalidade.
- Já o método de Huffman baseado em palavras é muito bom quando a cadeia de caracteres constitui texto em linguagem natural.



## Desvantagens dos Métodos de Ziv-Lempel para Ambiente de Recuperação de Informação

- É necessário iniciar a decodificação desde o início do arquivo comprimido $\Rightarrow$ Acesso randômico muito caro.
- É muito difícil pesquisar no arquivo comprimido sem descomprimir.
- Uma possível vantagem do método Ziv-Lempel é o fato de não ser necesário armazenar a tabela de símbolos da maneira com que o método de Huffman precisa.
- No entanto, isso tem pouca importância em um ambiente de recuperação de informação, já que se necessita o vocabulário do texto para criar o índice e permitir a pesquisa eficiente.

## Compressão de Huffman Usando Palavras

- Técnica de compressão mais eficaz para textos em linguagem natural.
- O método considera cada palavra diferente do texto como um símbolo.
- Conta suas freqüências e gera um código de Huffman para as palavras.
- A seguir, comprime o texto substituindo cada palavra pelo seu código.
- Assim, a compressão é realizada em duas passadas sobre o texto:
  1. Obtenção da freqüência de cada palavra diferente.
  2. Realização da compressão.



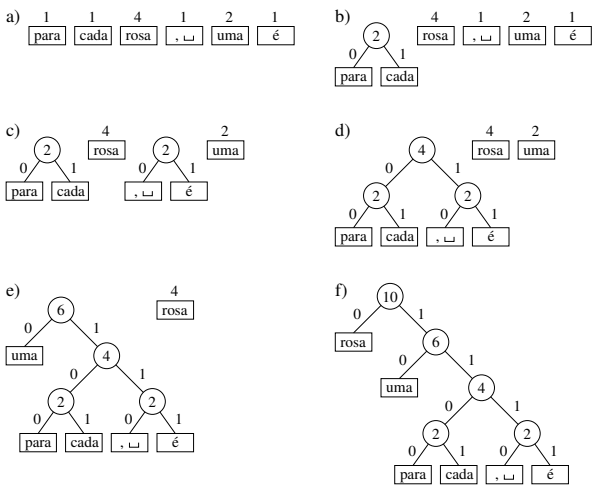
## Forma Eficiente de Lidar com Palavras e Separadores

- Um texto em linguagem natural é constituído de palavras e de separadores.
- Separadores são caracteres que aparecem entre palavras: espaço, vírgula, ponto, ponto e vírgula, interrogação, e assim por diante.
- Uma forma eficiente de lidar com palavras e separadores é representar o espaço simples de forma implícita no texto comprimido.
- Nesse modelo, se uma palavra é seguida de um espaço, então, somente a palavra é codificada.
- Senão, a palavra e o separador são codificados separadamente.
- No momento da decodificação, supõe-se que um espaço simples segue cada palavra, a não ser que o próximo símbolo corresponda a um separador.



## Compressão usando codificação de Huffman

**Exemplo:** "`para cada rosa rosa, uma rosa é uma rosa`"




**OBS:** O algoritmo de Huffman é uma abordagem **gulosa**.



## Árvore de Huffman

- O método de Huffman produz a árvore de codificação que minimiza o comprimento do arquivo comprimido.
- Existem diversas árvores que produzem a mesma compressão.
- Por exemplo, trocar o filho à esquerda de um nó por um filho à direita leva a uma árvore de codificação alternativa com a mesma razão de compressão.
- Entretanto, a escolha preferencial para a maioria das aplicações é a **árvore canônica**.
- Uma árvore de Huffman é canônica quando a altura da subárvore à direita de qualquer nó nunca é menor que a altura da subárvore à esquerda.

## Árvore de Huffman

- A representação do código na forma de árvore facilita a visualização.
- Sugere métodos de codificação e decodificação triviais:
  - **Codificação:** a árvore é percorrida emitindo *bits* ao longo de suas arestas.
  - **Decodificação:** os *bits* de entrada são usados para selecionar as arestas.
- Essa abordagem é ineficiente tanto em termos de espaço quanto em termos de tempo.



## Algoritmo Baseado na Codificação Canônica com Comportamento Linear em Tempo e Espaço

- O algoritmo é atribuído a Moffat e Katajainen (1995).
- Calcula os comprimentos dos códigos em lugar dos códigos propriamente ditos.
- A compressão atingida é a mesma, independentemente dos códigos utilizados.
- Após o cálculo dos comprimentos, há uma forma elegante e eficiente para a codificação e a decodificação.



## O Algoritmo

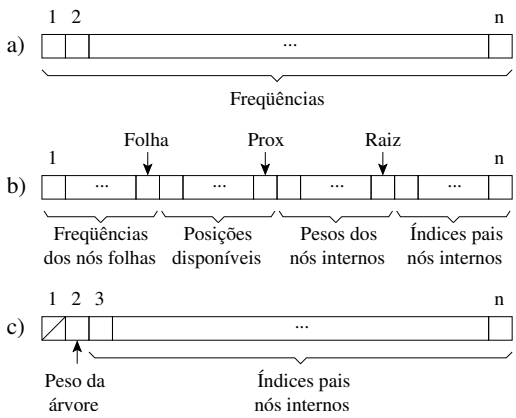
- A entrada do algoritmo é um vetor $A$ contendo as freqüências das palavras em ordem não-crescente.
- Freqüências relativas à frase exemplo: "`para cada rosa rosa, uma rosa é uma rosa`"

| 4 | 2 | 1 | 1 | 1 | 1 |
|---|---|---|---|---|---|

- Durante sua execução, são utilizados diversos vetores logicamente distintos, mas capazes de coexistirem no mesmo vetor das freqüências.
- O algoritmo divide-se em três fases:
  1. Combinação dos nós.
  2. Conversão do vetor no conjunto das profundidades dos nós internos.
  3. Calculo das profundidades dos nós folhas.



## Primeira Fase - Combinação dos nós




- A primeira fase é baseada em duas observações:
  1. A freqüência de um nó só precisa ser mantida até que ele seja processado.
  2. Não é preciso manter apontadores para os pais dos nós folhas, pois eles podem ser inferidos.

     **Exemplo:** nós internos nas profundidades $[0, 1, 2, 3, 3]$ teriam nós folhas nas profundidades $[1, 2, 4, 4, 4, 4]$.

# Pseudocódigo para a Primeira Fase

```
void primeiraFase (A, n) {
  raiz = n; folha = n;
  for (prox = n; prox >= 2; prox--){
    // Procura Posição
    if ((não existe folha) || ((raiz > prox) && (A[raiz] <= A[folha]))) {
      // Nó interno
      A[prox] = A[raiz]; A[raiz] = prox; raiz—;
    }
    else { // Nó folha
         A[prox] = A[folha]; folha—;
    }
    // Atualiza Freqüências
    if ((não existe folha) || ((raiz > prox) && (A[raiz] <= A[folha]))) {
      // Nó interno
      A[prox] = A[prox] + A[raiz]; A[raiz] = prox; raiz—;
    }
    else { // Nó folha
         A[prox] = A[prox] + A[folha]; folha—;
    }
  }
}
```

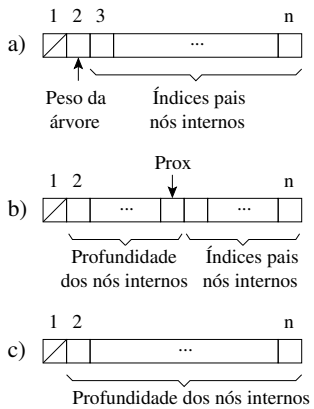


# Exemplo de processamento da primeira fase

| | 1 | 2 | 3 | 4 | 5 | 6 | Prox | Raiz | Folha |
|---|---|---|---|---|---|---|---|---|---|
| a) | 4 | 2 | 1 | 1 | 1 | 1 | 6 | 6 | 6 |
| b) | 4 | 2 | 1 | 1 | 1 | 1 | 6 | 6 | 5 |
| c) | 4 | 2 | 1 | 1 | 1 | 2 | 5 | 6 | 4 |
| d) | 4 | 2 | 1 | 1 | 1 | 2 | 5 | 6 | 3 |
| e) | 4 | 2 | 1 | 1 | 2 | 2 | 4 | 6 | 2 |
| f) | 4 | 2 | 1 | 2 | 2 | 4 | 4 | 5 | 2 |
| g) | 4 | 2 | 1 | 4 | 4 | 4 | 3 | 4 | 2 |
| h) | 4 | 2 | 2 | 4 | 4 | 4 | 3 | 4 | 1 |
| i) | 4 | 2 | 6 | 3 | 4 | 4 | 2 | 3 | 1 |
| j) | 4 | 4 | 6 | 3 | 4 | 4 | 2 | 3 | 0 |
| k) | / | 10 | 2 | 3 | 4 | 4 | 1 | 2 | 0 |



# Segunda Fase - Conversão do vetor no conjunto das profundidades dos nós internos





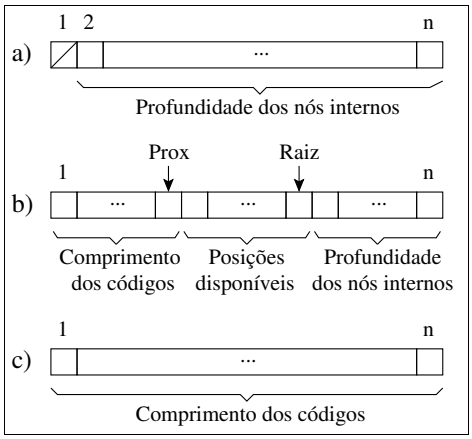

# Pseudocódigo para a Segunda Fase

```
void segundaFase (A, n) {
  A[2] = 0;
  for (prox = 3; prox <= n; prox++) A[prox] = A[A[prox]] + 1;
}
```

**Profundidades dos nós internos obtida com a segunda fase tendo como entrada o vetor exibido na letra k) da transparência 65:**

| / | 0 | 1 | 2 | 3 | 3 |
|---|---|---|---|---|---|

# Terceira Fase - Calculo das profundidades dos nós folhas






# Pseudocódigo para a Terceira Fase

```
void terceiraFase (A, n) {
  disp = 1; u = 0; h = 0; raiz = 2; prox = 1;
  while (disp > 0) {
    while ((raiz <= n) && (A[raiz] == h)) { u++; raiz++; }
    while (disp > u) { A[prox] = h; prox++; disp--;}
    disp = 2 * u; h++; u = 0;
  }
}
```

- Aplicando-se a Terceira Fase sobre:

| | 0 | 1 | 2 | 3 | 3 |
|---|---|---|---|---|---|

  Os comprimentos dos códigos em número de *bits* são obtidos:

| 1 | 2 | 4 | 4 | 4 | 4 |
|---|---|---|---|---|---|



# Cálculo do comprimento dos códigos a partir de um vertor de freqüências

```
void calculaCompCodigo (A, n) {
  primeiraFase (A, n);
  segundaFase (A, n);
  terceiraFase (A, n);
}
```



# Código Canônico

- Propriedades:
  1. Os comprimentos dos códigos obedecem ao algoritmo de Huffman.
  2. Códigos de mesmo comprimento são inteiros consecutivos.
- A partir dos comprimentos obtidos, o cálculo dos códigos propriamente dito é trivial: o primeiro código é composto apenas por zeros e, para os demais, adiciona-se 1 ao código anterior e faz-se um deslocamento à esquerda para obter-se o comprimento adequado quando necessário.
- **Codificação Canônica Obtida:**

| $i$ | Símbolo | Código Canônico |
|---|---|---|
| 1 | `rosa` | 0 |
| 2 | `uma` | 10 |
| 3 | `para` | 1100 |
| 4 | `cada` | 1101 |
| 5 | `,␣` | 1110 |
| 6 | `é` | 1111 |

## Elaboração de Algoritmos Eficientes para a Codificação e para a Decodificação

- Os algoritmos são baseados na seguinte observação:
  - Códigos de mesmo comprimento são inteiros consecutivos.
- Os algoritmos são baseados no uso de dois vetores com $maxCompCod$ elementos,sendo $maxCompCod$ o comprimento do maior código.



## Vetores $base$ e $offset$

- **Vetor** $base$**:** indica, para um dado comprimento $c$, o valor inteiro do primeiro código com esse comprimento.
- O vetor Base é calculado pela relação:

$$\text{base}[c] = \begin{cases} 0 & \text{se } c = 1, \\ 2 \times (\text{base}[c-1] + w_{c-1}) & \text{caso contrário,} \end{cases}$$

  sendo $w_c$ o número de códigos com comprimento $c$.
- $offset$ indica o índice no vocabulário da primeira palavra de cada comprimento de código $c$.
- Vetores $base$ e $offset$ para a tabela da transparência 71:

| $c$ | $base[c]$ | $offset[c]$ |
|---|---|---|
| 1 | 0 | 1 |
| 2 | 2 | 2 |
| 3 | 6 | 2 |
| 4 | 12 | 3 |



## Pseudocódigo para Codificação

```
Codigo codifica (i, maxCompCod) {
  c = 1;
  while ((c + 1 <= maxCompCod) && (i >= offset[c + 1])) c++;
  codigo = i − offset[c] + base[c];
  return (codigo, c);
}
```

**Obtenção do código:**

- O método de codificação recebe como parâmetros o índice $i$ do símbolo a ser codificado e o comprimento $maxCompCod$ dos vetores $base$ e $offset$.
- No anel **while** é feito o cálculo do comprimento $c$ de código a ser utilizado.
- A seguir, basta saber qual a ordem do código para o comprimento $c$ ($i - offset[c]$) e somar esse valor à $base[c]$.



## Exemplo de Codificação

- Para a palavra $i = 4$ (“`cada`”):
  1. Verifica-se que é um código de comprimento 4.
  2. Verifica-se também que é o segundo código com esse comprimento.
  3. Assim, seu código é 13 ($4 - offset[4] + base[4]$), o que corresponde a “`1101`” em binário.

## Pseudocódigo para Decodificação

```
int decodifica (maxCompCod) {
  c = 1; codigo = leBit (arqComp);
  while ((codigo << 1 ) >= base[c + 1] && ( c + 1 <= maxCompCod )) {
    codigo = (codigo << 1) | leBit (arqComp); c++;
  }
  i = codigo − base[c] + offset[c];
  return i;
}
```

- O programa recebe como parâmetro o comprimento $maxCompCod$ dos vetores $base$ e $offset$.
- Na decodificação, o arquivo de entrada é lido *bit*-a-*bit*, adicionando-se os *bits* lidos ao código e comparando-o com o vetor Base.
- O anel **while** mostra como identificar o código a partir de uma posição do arquivo comprimido.



## Exemplo de Decodificação

- Decodificação da seqüência de *bits* "`1101`":

| $c$ | LeBit | Codigo | Codigo $<< 1$ | Base$[c+1]$ |
|---|---|---|---|---|
| 1 | 1 | 1 | - | - |
| 2 | 1 | 10 **or** $1 = 11$ | 10 | 10 |
| 3 | 0 | 110 **or** $0 = 110$ | 110 | 110 |
| 4 | 1 | 1100 **or** $1 = 1101$ | 1100 | 1100 |

- A primeira linha da tabela representa o estado inicial do anel **while** quando já foi lido o primeiro *bit* da seqüência, o qual foi atribuído à variável $codigo$.
- A linha dois e seguintes representam a situação do anel **while** após cada respectiva iteração.
- No caso da linha dois da tabela, o segundo *bit* da seqüência foi lido (*bit* "`1`") e a variável $codigo$ recebe o código anterior deslocado à esquerda de um *bit* seguido da operação *or* com o *bit* lido.
- De posse do código, $base$ e $offset$ são usados para identificar qual o índice $i$ da palavra no vocabulário, sendo $i = codigo - base[c] + offset[c]$.



## Pseudocódigo para realizar a compressão

```
void compressao (nomeArqTxt, nomeArqComp) {
  arqComp = new RandomAccessFile (nomeArqComp, "rws");
  arqTxt = new BufferedReader (new FileReader(nomeArqTxt));
  // Primeira etapa
  String palavra = null; TabelaHash vocabulario;
  while (existirem palavras) {
    palavra = proximaPalavra (arqTxt);
    itemVoc = vocabulario.pesquisa (palavra);
    if (itemVoc != null) itemVoc.freq = itemVoc.freq + 1;
    else vocabulario.insere (palavra);
  }
  // Segunda etapa
  A[] = ordenaPorFrequencia (vocabulario); calculaCompCodigo (A, n);
  maxCompCod = constroiVetores (A, n); gravaVocabulario (A, arqComp);
  // Terceira etapa
  while (existirem palavras) {
    palavra = proximaPalavra (arqTxt);
    itemVoc = vocabulario.pesquisa (palavra);
    codigo = codifica(itemVoc.ordem, maxCompCod);
    escreve (codigo, maxCompCod);
  }
}
```



## Pseudocódigo para realizar a descompressão

```
void descompressao (nomeArqTxt, nomeArqComp) {
  arqComp = new RandomAccessFile (nomeArqComp, "rws");
  arqTxt = new BufferedWriter (new FileWriter (nomeArqTxt));
  int maxCompCod = leVetores ();
  String vocabulario[] = leVocabulario ();
  while ((i = decodifica (maxCompCod)) >= 0) {
    if ((palavra anterior não é delimitador) && (vocabulario[i] não é delimitador))
      arqTxt.write (" ");
    arqTxt.write (vocabulario[i]);
  }
}
```

## Codificação de Huffman Usando Palavras - Análise

- A representação do código de Huffman na forma de uma árvore é ineficiente em termos de espaço e de tempo (não é usado na prática).
- **Codificação canônica**: forma mais eficiente baseada nos comprimentos dos códigos em vez dos códigos propriamente ditos (Moffat e Katajainen - 1995).
- Feito *in situ* a partir de um vetor $A$ contendo as freqüências das palavras em ordem não crescente a um custo $O(n)$ em tempo e em espaço.
- O algoritmo requer apenas os dois vetores $base$ e $offset$ de tamanho $maxCompCod$, sendo $maxCompCod$ o comprimento do maior código.
- A decodificação é também muito eficiente pois apenas os vetores $base$ e $offset$ são consultados.
- Não há necessidade de realizar a decodificação *bit* a *bit*, como na árvore de Huffman.



## Codificação de Huffman Usando Bytes

- O método original proposto por Huffman (1952) tem sido usado como um código binário.
- Moura, Navarro, Ziviani e Baeza-Yates (2000) modificaram a atribuição de códigos de tal forma que uma seqüência de *bytes* é associada a cada palavra do texto.
- Conseqüentemente, o grau de cada nó passa de 2 para 256. Essa versão é chamada de *código de Huffman pleno*.
- Outra possibilidade é utilizar apenas 7 dos 8 *bits* de cada *byte* para a codificação, e a árvore passa então a ter grau 128.
- Nesse caso, o oitavo *bit* é usado para marcar o primeiro *byte* do código da palavra, sendo chamado de *código de Huffman com marcação*.



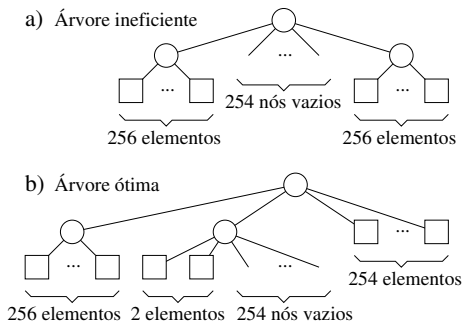
## Exemplo de Códigos Plenos e com Marcação

- O código de Huffman com marcação ajuda na pesquisa sobre o texto comprimido.
- **Exemplo:**
  - Código pleno para a palavra "`uma`" com 3 *bytes* "`47 81 8`".
  - Código com marcação para a palavra "`uma`" com 3 *bytes* "`175 81 8`"
  - Note que o primeiro *byte* é $175 = 47 + 128$.
- Assim, no código com marcação o oitavo *bit* é 1 quando o *byte* é o primeiro do código, senão ele é 0.



## Árvore de Huffman orientada a *bytes*

- A construção da árvore de Huffman orientada a *bytes* pode ocasionar o aparecimento de nós internos não totalmente preenchidos quando a árvore não é binária:




- Na Figura, o alfabeto possui 512 símbolos (nós folhas), todos com a mesma freqüência de ocorrência.
- O segundo nível tem 254 espaços vazios que poderiam ser ocupados com símbolos, mudando o comprimento de seus códigos de 2 para 1 *byte*.

## Movendo Nós Vazios para Níveis mais Profundos

- Um meio de assegurar que nós vazios sempre ocupem o nível mais baixo da árvore é combiná-los com os nós de menores freqüências.
- O objetivo é movê-los para o nível mais profundo da árvore.
- Para isso, devemos selecionar o número de símbolos que serão combinados com os nós vazios.
- Essa seleção é dada pela equação $1 + ((n - \text{baseNum}) \mod (\text{baseNum} - 1))$.
- No caso da Figura da transparência anterior é igual a $1 + ((512 - 256) \mod 255) = 2$.



## Classe *HuffmanByte*

```
package cap8;
import java.io.*;
import cap5.endaberto.TabelaHash;
import cap4.ordenacaointerna.Ordenacao;
public class HuffmanByte {
  private int baseNum;
  private int base[], offset[];
  private RandomAccessFile arqComp; // Arquivo comprimido
  private String nomeArqTxt; // Nome do arquivo texto a ser comprimido
  private String nomeArqDelim; // Nome do arquivo que contém os delimitadores
  private TabelaHash vocabulario;
  private static class Codigo {
    int codigo; int c; // Comprimento do código
  }
  public HuffmanByte (String nomeArqDelim, int baseNum, int m,
                      int maxTamChave) throws Exception {
    this.baseNum = baseNum; this.base = null; this.offset = null;
    this.nomeArqTxt = null; this.nomeArqDelim = nomeArqDelim;
    this.vocabulario = new TabelaHash (m, maxTamChave);
  }
  public void compressao (String nomeArqTxt,
                          String nomeArqComp) throws Exception {
    public void descompressao (String nomeArqTxt,
                               String nomeArqComp) throws Exception {
  }
```



## Generalização do Cálculo dos Comprimentos dos Códigos

```
private void calculaCompCodigo (ItemVoc[] A, int n) {
  int resto = 0;
  if (n > (this.baseNum - 1)) {
    resto = 1 + ((n - this.baseNum) % (this.baseNum - 1));
    if (resto < 2) resto = this.baseNum;
  }
  else resto = n - 1;
  // noInt: Número de nós internos
  int noInt = 1 + ((n - resto) / (this.baseNum - 1));
  int freqn = ((Integer)A[n].recuperaChave ()).intValue ();
  for (int x = (n - 1); x >= (n - resto + 1); x--){
    int freqx = ((Integer)A[x].recuperaChave ()).intValue ();
    freqn = freqn + freqx;
  }
  A[n].alteraChave (new Integer (freqn));
  // Primeira Fase
  int raiz = n; int folha = n - resto; int prox;
  for (prox = n - 1; prox >= (n - noInt + 1); prox--){
    // Procura Posição
    int freqraiz = ((Integer)A[raiz].recuperaChave ()).intValue ();
    if ((folha < 1) || ((raiz > prox) &&
        (freqraiz <= ((Integer)A[folha].recuperaChave()).intValue()))) {
      // Nó interno
      A[prox].alteraChave (new Integer (freqraiz));
      A[raiz].alteraChave (new Integer (prox)); raiz--;
    }
    else { // Nó folha
      int freqfolha = ((Integer)A[folha].recuperaChave ()).intValue ();
      A[prox].alteraChave (new Integer (freqfolha)); folha--;
    }
```



## Generalização do Cálculo dos Comprimentos dos Códigos

```
    // Atualiza Freqüências
    for (int x = 1; x <= (this.baseNum - 1); x++) {
      freqraiz = ((Integer)A[raiz].recuperaChave ()).intValue ();
      int freqprox = ((Integer)A[prox].recuperaChave ()).intValue ();
      if ((folha < 1) || ((raiz > prox) &&
          (freqraiz<=((Integer)A[folha].recuperaChave()).intValue()))) {
        // Nó interno
        A[prox].alteraChave (new Integer (freqprox + freqraiz));
        A[raiz].alteraChave (new Integer (prox)); raiz--;
      }
      else { // Nó folha
        int freqfolha = ((Integer)A[folha].recuperaChave ()).intValue ();
        A[prox].alteraChave(new Integer(freqprox + freqfolha)); folha--;
      }
    }
  }
  // Segunda Fase
  A[raiz].alteraChave (new Integer (0));
  for (prox = raiz + 1; prox <= n; prox++) {
    int pai = ((Integer)A[prox].recuperaChave ()).intValue ();
    int profundidadepai = ((Integer)A[pai].recuperaChave ()).intValue ();
    A[prox].alteraChave (new Integer (profundidadepai + 1));
  }
```

## Generalização do Cálculo dos Comprimentos dos Códigos

```
  // Terceira Fase
  int disp = 1; int u = 0; int h = 0; prox = 1;
  while (disp > 0) {
    while ((raiz <= n) &&
      (((Integer)A[raiz].recuperaChave()).intValue() == h)) {u++; raiz++;}
    while (disp > u) {
      A[prox].alteraChave (new Integer (h)); prox++; disp--;
      if (prox > n) { u = 0; break; }
    }
    disp = this.baseNum * u; h = h + 1; u = 0;
  }
}
```

**OBS:** *baseNum* pode ser usada para trabalharmos com quaisquer bases numéricas menores ou iguais a um *byte*. Por exemplo, para a codificação plena o valor é 256 e para a codificação com marcação o valor é 128.



## Mudanças em Relação ao Pseudocódigo Apresentado

- A mudança maior está no código inserido antes da primeira fase para eliminar o problema causado por nós internos da árvore não totalmente preenchidos.
- Na primeira fase, as *baseNum* árvores de menor custo são combinadas a cada passo, em vez de duas como no caso da codificação binária:
  - Isso é feito pelo anel **for** introduzido na parte que atualiza freqüências na primeira fase.
- A segunda fase não sofre alterações.
- A terceira fase recebe a variável $disp$ para indicar quantos nós estão disponíveis em cada nível.



## Codificação orientada a *bytes*

```
private Codigo codifica (int ordem, int maxCompCod) {
  Codigo cod = new Codigo (); cod.c = 1;
  while ((cod.c + 1 <= maxCompCod) && (ordem >= this.offset[cod.c + 1]))
    cod.c++;
  cod.codigo = ordem - this.offset[cod.c] + this.base[cod.c];
  return cod;
}
```

**OBS:** a codificação orientada a *bytes* não requer nenhuma alteração em relação à codificação usando *bits*



## Decodificação orientada a *bytes*

```
private int decodifica (int maxCompCod) throws Exception {
  int logBase2 = (int)(Math.log(this.baseNum)/Math.log(2));
  int c = 1; int codigo = this.arqComp.read ();
  if (codigo < 0) return codigo; // Fim de arquivo
  if (logBase2 == 7) codigo = codigo - 128; // Remove o bit de marca-
cao
  while (((c + 1) <= maxCompCod) &&
         ((codigo << logBase2) >= this.base[c+1])) {
    int codigoTmp = this.arqComp.read ();
    codigo = (codigo << logBase2) | codigoTmp; c++; }
  return (codigo - this.base[c] + this.offset[c]);
}
```

**Alterações:**

1. Permitir a leitura *byte* a *byte* do arquivo comprimido, em vez de *bit* a *bit*. em relação ao número de *bits* que devem ser deslocados à esquerda para se encontrar o comprimento $c$ do código, o qual indexa os vetores $base$ e $offset$.
2. O número de *bits* que devem ser deslocados à esquerda para se encontrar o comprimento $c$, o qualindexa os vetores $base$ e $offset$, é dado por: $\log_2 \mathrm{BaseNum}$

## Cálculo dos Vetores $base$ e $offset$

- O cálculo do vetor $offset$ não requer alteração alguma.
- Para generalizar o cálculo do vetor $base$, basta substituir o fator $2$ por $baseNum$, como abaixo:

$$base[c] = \begin{cases} 0 & \text{se } c = 1, \\ baseNum \times (base[c-1] + w_{c-1}) & \text{caso contrário.} \end{cases}$$



## Construção dos Vetores $base$ e $offset$

```
private int constroiVetores (ItemVoc A[], int n) throws Exception {
  int maxCompCod = ((Integer)A[n].recuperaChave()).intValue();
  int wcs[] = new int[maxCompCod + 1]; // Ignora a posição 0
  this.offset = new int[maxCompCod + 1]; // Ignora a posição 0
  this.base = new int[maxCompCod + 1]; // Ignora a posição 0
  for (int i = 1; i <= maxCompCod; i++) wcs[i] = 0;
  for (int i = 1; i <= n; i++) {
    int freq = ((Integer)A[i].recuperaChave()).intValue();
    wcs[freq]++; this.offset[freq] = i - wcs[freq] + 1;
  }
  this.base[1] = 0;
  for (int i = 2; i <= maxCompCod; i++) {
    this.base[i] = this.baseNum * (this.base[i-1] + wcs[i-1]);
    if (this.offset[i] == 0) this.offset[i] = this.offset[i-1];
  }
  // Salvando as tabelas em disco
  this.arqComp.writeInt (maxCompCod);
  for (int i = 1; i <= maxCompCod; i++) {
    this.arqComp.writeInt (this.base[i]);
    this.arqComp.writeInt (this.offset[i]); }
  return maxCompCod;
}
```



## Extração do próximo símbolo a ser codificado

```
package cap8;
import java.util.StringTokenizer;
import java.io.*;
public class ExtraiPalavra {
  private BufferedReader arqDelim, arqTxt;
  private StringTokenizer palavras;
  private String delimitadores, palavraAnt, palavra;
  private boolean eDelimitador (char ch) {
    return (this.delimitadores.indexOf (ch) >= 0);
  }
  public ExtraiPalavra (String nomeArqDelim, String nomeArqTxt)
                        throws Exception {
    this.arqDelim = new BufferedReader (new FileReader (nomeArqDelim));
    this.arqTxt = new BufferedReader (new FileReader (nomeArqTxt));
    // Os delimitadores devem estar juntos em uma única linha do arquivo
    this.delimitadores = arqDelim.readLine() + "\r\n";
    this.palavras = null; this.palavra = null; this.palavraAnt = " ";
  }
  public String proximaPalavra () throws Exception{
    String palavraTemp = ""; String resultado = "";
    if (this.palavra != null) {
      palavraTemp = palavra; palavra = null;
      palavraAnt = palavraTemp; return palavraTemp;
    }
    if (palavras == null || !palavras.hasMoreTokens ()) {
      String linha = arqTxt.readLine();
      if (linha == null) return null;
      linha += "\n";
      this.palavras=new StringTokenizer (linha,this.delimitadores,true);
    }
```



## Extração do próximo símbolo a ser codificado

```
      String aux = this.palavras.nextToken();
    while (eDelimitador (aux.charAt (0)) && palavras.hasMoreTokens ()) {
      palavraTemp += aux; aux = this.palavras.nextToken();
    }
    if (palavraTemp.length () == 0) resultado = aux;
    else {
      this.palavra = aux;
      if (palavraTemp.length () == 1 && palavraTemp.equals(" ") &&
          palavraAnt.length () > 0 && palavra.length () > 0 &&
          !eDelimitador (palavraAnt.charAt (0)) &&
          !eDelimitador (palavra.charAt (0)))
        palavraTemp = palavraTemp.trim ();
      resultado = palavraTemp;
    } this.palavraAnt = resultado; return resultado;
  }
  public void fecharArquivos () throws Exception {
    this.arqDelim.close(); this.arqTxt.close();
  }
}
```

## Classe para representar as informações de uma entrada do vocabulário

```
package cap8;
import cap4.Item; // vide Programa ??
public class ItemVoc implements Item {
  private String palavra;
  private int freq, ordem;
  // outros componentes do registro
  public ItemVoc (String palavra, int freq, int ordem) {
    this.palavra = palavra;
    this.freq = freq; this.ordem = ordem;
  }
  public int compara (Item it) {
    ItemVoc item = (ItemVoc) it;
    if (this.freq < item.freq) return 1;
    else if (this.freq > item.freq) return -1;
    return 0;
  }
  public void alteraChave (Object freq) {
    Integer ch = (Integer) freq; this.freq = ch.intValue ();
  }
  public Object recuperaChave () { return new Integer (this.freq); }
  public void alteraOrdem (int ordem) { this.ordem = ordem; }
  public int recuperaOrdem () { return this.ordem; }
  public String palavra () { return this.palavra; }
}
```



## Código para Fazer a Compressão

- O Código para fazer a compressão é dividio em três etapas:
  1. Na primeira, as palavras são extraídas do texto a ser comprimido e suas respectivas freqüências são contabilizadas.
  2. Na segunda, são gerados os vetores $base$ e $offset$, os quais são gravados no arquivo comprimido seguidamente do vocabulário. Para delimitar os símbolos do vocabulário no disco, cada um deles é separado pelo caractere zero.
  3. Na terceira, o arquivo texto é percorrido pela segunda vez, sendo seus símbolos novamente extraídos, codificados e gravados no arquivo comprimido.



## Código para Fazer a Compressão

```
public void compressao (String nomeArqTxt,
                        String nomeArqComp) throws Exception {
  this.nomeArqTxt = nomeArqTxt;
  this.arqComp = new RandomAccessFile (nomeArqComp, "rws");
  this.primeiraEtapa ();
  int maxCompCod = this.segundaEtapa ();
  this.terceiraEtapa (maxCompCod);
  this.arqComp.close ();
}
```



## Primeira etapa da compressão

```
private void primeiraEtapa ( ) throws Exception {
  ExtraiPalavra palavras = new ExtraiPalavra (nomeArqDelim, nomeArqTxt);
  String palavra = null;
  while ((palavra = palavras.proximaPalavra()) != null) {
    // O primeiro espaço depois da palavra não é codificado
    if (palavra.equals ("")) continue;
    ItemVoc itemVoc = (ItemVoc) this.vocabulario.pesquisa (palavra);
    if ( itemVoc != null) { // Incrementa freqüência
      int freq = ((Integer)itemVoc.recuperaChave ()).intValue ();
      itemVoc.alteraChave (new Integer (freq + 1));
    } else { // Insere palavra com freqüência 1
      itemVoc = new ItemVoc (palavra, 1, 0);
      this.vocabulario.insere (palavra, itemVoc);
    }
  }
  palavras.fecharArquivos();
}
```

## Segunda etapa da compressão

```
private int segundaEtapa () throws Exception {
  ItemVoc A[] = this.ordenaPorFrequencia ();
  int n = A.length − 1;
  this.calculaCompCodigo (A, n);
  int maxCompCod = this.constroiVetores (A, n);
  // Grava Vocabulário
  this.arqComp.writeInt (n);
  for (int i = 1; i <= n; i++) {
    this.arqComp.writeChars (A[i].palavra ());
    this.arqComp.writeChar ('\0');
    A[i].alteraOrdem (i);
  }
  return maxCompCod;
}
```



## Método para ordenar o vocabulário por freqüência

- O objetivo desse método é recuperar as entradas do vocabulário, armazená-las contigüamente em um vetor e ordenar o vetor obtido na ordem não crescente pela freqüência das palavras no texto.
- Para isso, foi criado o operador $recuperaItens$, o qual retorna nas posições de $0$ a $n-1$ do vetor $itens$ as $n$ referências às entradas do vocabulário, as quais são objetos do tipo $ItemVoc$.
- Recuperados os itens, o método $ordenaPorFrequencia$ copia as referências aos objetos do tipo $ItemVoc$ que representam as entradas do vocabulário do vetor $aux$ para as posições de $1$ a $n$ do vetor $A$.
- Por fim, na classe $ItemVoc$ o vetor $A$ é ordenado de forma não crescente por suas respectivas freqüências de ocorrência ($Quicksort$).
- O método $ordenaPorFrequencia$ retorna o vetor ordenado.



## Método para ordenar o vocabulário por freqüência

```
private ItemVoc[] ordenaPorFrequencia () {
  Object aux[] = this.vocabulario.recuperaItens ();
  ItemVoc A[] = new ItemVoc[aux.length+1]; // Ignora a posição 0
  for (int i = 0; i < aux.length; i++) A[i+1] = (ItemVoc)aux[i];
  Ordenacao.quicksort (A, aux.length);
  return A;
}
```



## Operador para recuperar os objetos contidos em uma tabela $hash$

```
public Object[] recuperaItens () {
  int n = 0;
  for (int i = 0; i < this.M; i++)
    if (this.tabela[i] != null && !this.tabela[i].retirado) n++;
  Object itens[] = new Object[n]; n = 0;
  for (int i = 0; i < this.M; i++)
    if (this.tabela[i] != null && !this.tabela[i].retirado)
      itens[n++] = this.tabela[i].item;
  return itens;
}
```

## Terceira etapa da compressão

```
private void terceiraEtapa (int maxCompCod) throws Exception {
  ExtraiPalavra palavras = new ExtraiPalavra (nomeArqDelim, nomeArqTxt);
  String palavra = null;
  while ((palavra = palavras.proximaPalavra()) != null) {
    // O primeiro espaço depois da palavra não é codificado
    if (palavra.equals ("")) continue;
    ItemVoc itemVoc = (ItemVoc) this.vocabulario.pesquisa (palavra);
    int ordem = itemVoc.recuperaOrdem ();
    Codigo cod = this.codifica (ordem, maxCompCod);
    this.escreve (cod, maxCompCod);
  }
  palavras.fecharArquivos();
}
```



## Método $escreve$

- O método $escreve$ recebe o código e seu comprimento $c$ em um objeto do tipo $Codigo$.
- O código é representado por um inteiro, o que limita seu comprimento a, no máximo, 4 *bytes* em um compilador que usa 4 *bytes* para representar inteiros.
- Primeiramente, o método $escreve$ extrai o primeiro *byte* e, caso o código de Huffman utilizado seja o de marcação (baseNum = 128), coloca a marcação no oitavo *bit*, fazendo uma operação *or* do *byte* com a constante 128.
- Esse *byte* é então colocado na primeira posição do vetor $saida$.
- No anel **w**hile, caso o comprimento $c$ do código seja maior do que um, os demais *bytes* são extraídos e armazenados em $saida[i]$, em que $2 \leq i \leq c$.
- Por fim, o vetor de *bytes* $saida$ é gravado em disco no anel **f**or.



## Implementação do Método $escreve$

```
private void escreve (Codigo cod, int maxCompCod) throws Exception {
  int saida[] = new int[maxCompCod + 1]; // Ignora a posição 0
  int logBase2 = (int)(Math.log(this.baseNum)/Math.log(2));
  int mask = (int)Math.pow (2, logBase2) - 1;
  int i = 1; int cTmp = cod.c;
  saida[i] = cod.codigo >> (logBase2*(cod.c - 1));
  if (logBase2 == 7) saida[i] = saida[i] | 128; // Marcação
  i++; cod.c--;
  while (cod.c > 0) {
    saida[i] = (cod.codigo >> (logBase2*(cod.c - 1))) & mask;
    i++; cod.c--;
  }
  for (i = 1; i <= cTmp; i++) this.arqComp.writeByte (saida[i]);
}
```



## Descrição do Código para Fazer a Descompressão

- O primeiro passo é recuperar o modelo usado na compressão. Para isso, lê o alfabeto, o vetor $base$, o vetor $offset$ e o vetor $vocabulario$.
- Em seguida, inicia a decodificação, tomando o cuidado de adicionar um espaço em branco entre dois símbolos que sejam palavras.
- O processo de decodificação termina quando o arquivo comprimido é totalmente percorrido.

## Código para Fazer a Descompressão

```
public void descompressao (String nomeArqTxt,
                           String nomeArqComp) throws Exception {
  this.nomeArqTxt = nomeArqTxt;
  this.arqComp = new RandomAccessFile (nomeArqComp, "rws");
  BufferedReader arqDelim = new BufferedReader (
                            new FileReader (this.nomeArqDelim));
  BufferedWriter arqTxt = new BufferedWriter (
                          new FileWriter (this.nomeArqTxt));
  String delim = arqDelim.readLine() + "\r\n";
  int maxCompCod = this.leVetores ();
  String vocabulario[] = this.leVocabulario ();
  int ind = 0; String palavraAnt = " ";
  while ((ind = this.decodifica (maxCompCod)) >= 0) {
    if (!eDelimitador (delim, palavraAnt.charAt(0)) &&
        !eDelimitador (delim, vocabulario[ind].charAt(0)))
      arqTxt.write (" ");
    arqTxt.write (vocabulario[ind]);
    palavraAnt = vocabulario [ind];
  }
  arqTxt.close ();
}
```

**OBS:** Observe que na descompressão, o vocabulário é representado por um vetor de símbolos do tipo $String$.



## Métodos auxiliares da descompressão

```
private int leVetores () throws Exception {
  int maxCompCod = this.arqComp.readInt ();
  this.offset = new int[maxCompCod + 1]; // Ignora a posição 0
  this.base = new int[maxCompCod + 1]; // Ignora a posição 0
  for (int i = 1; i <= maxCompCod; i++) {
    this.base[i] = this.arqComp.readInt ();
    this.offset[i] = this.arqComp.readInt ();
  }
  return maxCompCod;
}
private String[] leVocabulario () throws Exception{
  int n = this.arqComp.readInt ();
  String vocabulario[] = new String[n+1]; // Ignora a posição 0
  for (int i = 1; i <= n; i++) {
    vocabulario[i] = ""; char ch;
    while ((ch = this.arqComp.readChar ()) != '\0') {
      vocabulario[i] += ch;
    }
  }
  return vocabulario;
}
private boolean eDelimitador (String delim, char ch) {
  return (delim.indexOf (ch) >= 0);
}
```



## Resultados Experimentais

- Mostram que não existe grande degradação na razão de compressão na utilização de *bytes* em vez de *bits* na codificação das palavras de um vocabulário.
- Por outro lado, tanto a descompressão quanto a pesquisa são muito mais rápidas com uma codificação de Huffman usando *bytes* do que uma codificação de Huffman usando *bits*, isso porque deslocamentos de *bits* e operações usando máscaras não são necessárias.
- Os experimentos foram realizados em uma máquina PC Pentium de 200 MHz com 128 *megabytes* de *RAM*.



## Resultados Experimentais - Comparação das técnicas de compressão sobre o arquivo WSJ

Dados sobre a coleção usada nos experimentos:

| Texto | | Vocabulário | | Vocab./Texto | |
|---|---|---|---|---|---|
| Tam (bytes) | #Palavras | Tam (bytes) | #Palavras | Tamanho | #Palavras |
| 262.757.554 | 42.710.250 | 1.549.131 | 208.005 | 0,59% | 0,48% |

| Método | Razão de Compressão | Tempo (min) de Compressão | Tempo (min) de Descompressão |
|---|---|---|---|
| Huffman binário | 27,13 | 8,77 | 3,08 |
| Huffman pleno | 30,60 | 8,67 | 1,95 |
| Huffman com marcação | 33,70 | 8,90 | 2,02 |
| Gzip | 37,53 | 25,43 | 2,68 |
| Compress | 42,94 | 7,60 | 6,78 |

## Pesquisa em Texto Comprimido

- Uma das propriedades mais atraentes do método de Huffman usando *bytes* em vez de *bits* é que o texto comprimido pode ser pesquisado exatamente como qualquer texto não comprimido.
- Basta comprimir o padrão e realizar uma pesquisa diretamente no arquivo comprimido.
- Isso é possível porque o código de Huffman usa *bytes* em vez de *bits*; de outra maneira, o método seria complicado ou mesmo impossível de ser implementado.



## Casamento Exato

**Algoritmo:**

- Buscar a palavra no vocabulário, podendo usar busca binária nesta fase:
  - Se a palavra for localizada no vocabulário, então o código de Huffman com marcação é obtido.
  - Senão a palavra não existe no texto comprimido.
- A seguir, o código é pesquisado no texto comprimido usando qualquer algoritmo para casamento exato de padrão.
- Para pesquisar um padrão contendo mais de uma palavra, o primeiro passo é verificar a existência de cada palavra do padrão no vocabulário e obter o seu código:
  - Se qualquer das palavras do padrão não existir no vocabulário, então o padrão não existirá no texto comprimido.
  - Senão basta coletar todos os códigos obtidos e realizar a pesquisa no texto comprimido.



## Método para realizar busca no arquivo comprimido

```
public void busca (String nomeArqComp) throws Exception {
  BufferedReader in = new BufferedReader (
                      new InputStreamReader (System.in));
  this.arqComp = new RandomAccessFile (nomeArqComp, "rws");
  int maxCompCod = this.leVetores ();
  String vocabulario[] = this.leVocabulario ();
  int codigo; String T = ""; String P = "";
  while ((codigo = this.arqComp.read ()) >= 0) T += (char)codigo;
  while (true) {
    System.out.print ("Padrao (ou s para sair):"); P = in.readLine();
    if (P.equals ("s")) break; int ord = 1;
    for (ord = 1; ord < vocabulario.length; ord++)
      if (vocabulario[ord].equals (P)) break;
    if (ord == vocabulario.length) {
      System.out.println("Padrao:" + P + " nao encontrado"); continue;
    }
    Codigo cod = this.codifica (ord, maxCompCod);
    String Padrao = this.atribui (cod);
    CasamentoExato.bmh (T, T.length (), Padrao, Padrao.length ());
  }
}
```



## Método para atribuir o código ao padrão

```
private String atribui (Codigo cod) {
  String P = "";
  P += (char)((cod.codigo >> (7*(cod.c - 1))) | 128);
  cod.c--;
  while (cod.c > 0) {
    P += (char)((cod.codigo >> (7*(cod.c - 1))) & 127);
    cod.c--;
  }
  return P;
}
```

## Programa para teste dos algoritmos de compressão, descompressão e busca exata em texto comprimido

```
package cap8;
import java.io.*;
public class Huffman {
  private static BufferedReader in = new BufferedReader (
                                     new InputStreamReader (System.in));
  private static final int baseNum = 128;
  private static final int m = 1001;
  private static final int maxTamPalavra = 15;
  private static void imprime (String msg) {
    System.out.print (msg);
  }
  public static void main (String[] args) throws Exception {
    imprime ("Arquivo com os delimitadores em uma linha:");
    String nomeArqDelim = in.readLine ();
    String opcao = "";
    do {
      imprime ("*************************************\n");
      imprime ("*                Opcoes               *\n");
      imprime ("*-------------------------------------*\n");
      imprime ("* (c) Compressao                      *\n");
      imprime ("* (d) Descompressao                   *\n");
      imprime ("* (p) Pesquisa no texto comprimido    *\n");
      imprime ("* (f) Termina                         *\n");
      imprime ("*************************************\n");
      imprime ("* Opcao:"); opcao = in.readLine();
```



## Programa para teste dos algoritmos de compressão, descompressão e busca exata em texto comprimido

```
      if (opcao.toLowerCase().equals ("c")) {
        imprime ("Arquivo texto a ser comprimido:");
        String nomeArqTxt = in.readLine ();
        imprime ("Arquivo comprimido a ser gerado:");
        String nomeArqComp = in.readLine ();
        HuffmanByte huff = new HuffmanByte (nomeArqDelim, baseNum,
                                            m, maxTamPalavra);
        huff.compressao (nomeArqTxt, nomeArqComp);
      }
      else if (opcao.toLowerCase().equals ("d")) {
        imprime ("Arquivo comprimido a ser descomprimido:");
        String nomeArqComp = in.readLine ();
        imprime ("Arquivo texto a ser gerado:");
        String nomeArqTxt = in.readLine ();
        HuffmanByte huff = new HuffmanByte (nomeArqDelim, baseNum,
                                            m, maxTamPalavra);
        huff.descompressao (nomeArqTxt, nomeArqComp);
      }
      else if (opcao.toLowerCase().equals ("p")) {
        imprime ("Arquivo comprimido para ser pesquisado:");
        String nomeArqComp = in.readLine ();
        HuffmanByte huff = new HuffmanByte (null, baseNum,
                                            m, maxTamPalavra);
        huff.busca (nomeArqComp);
      }
    } while (!opcao.toLowerCase().equals ("f"));
  }
}
```



## Casamento Aproximado

### Algoritmo:

- Pesquisar o padrão no vocabulário. Neste caso, podemos ter:
  - Casamento exato, o qual pode ser uma **pesquisa binária** no vocabulário, e uma vez que a palavra tenha sido encontrada a folha correspondente na árvore de Huffman é marcada.
  - Casamento aproximado, o qual pode ser realizado por meio de pesquisa seqüencial no vocabulário, usando o algoritmo Shift-And.
  - Neste caso, várias palavras do vocabulário podem ser encontradas e a folha correspondente a cada uma na árvore de Huffman é marcada.

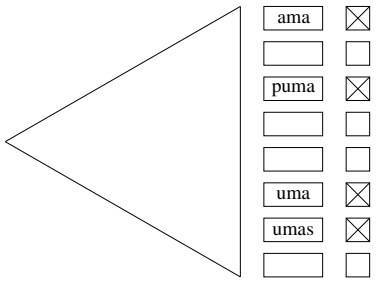


## Casamento Aproximado

### Algoritmo (Continuação):

- A seguir, o arquivo comprimido é lido *byte* a *byte*, ao mesmo tempo que a árvore de decodificação de Huffman é percorrida sincronizadamente.
- Ao atingir uma folha da árvore:
  - se ela estiver marcada, então existe casamento com a palavra do padrão.
- Seja uma folha marcada ou não, o caminhamento na árvore volta à raiz ao mesmo tempo que a leitura do texto comprimido continua.

## Esquema geral de pesquisa para a palavra "*uma*" permitindo 1 erro






## Casamento Aproximado Usando uma Frase como Padrão

- **Frase:** seqüência de padrões (palavras), em que cada padrão pode ser desde uma palavra simples até uma expressão regular complexa permitindo erros.

**Pré-Processamento:**

- Se uma frase tem $j$ palavras, então uma máscara de $j$ *bits* é colocada junto a cada palavra do vocabulário (folha da árvore de Huffman).
- Para uma palavra $x$ da frase, o $i$-ésimo *bit* da máscara é feito igual a 1 se $x$ é a $i$-ésima palavra da frase.
- Assim, cada palavra $i$ da frase é pesquisada no vocabulário e a $i$-ésima posição da máscara é marcada quando a palavra é encontrada no vocabulário.



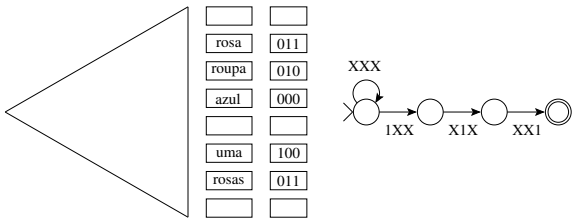
## Casamento Aproximado Usando uma Frase como Padrão

**Leitura do Texto Comprimido:**

- O estado da pesquisa é controlado por um **autômato finito não-determinista** de $j + 1$ estados.
- O autômato permite mover do estado $i$ para o estado $i + 1$ sempre que a $i$-ésima palavra da frase é reconhecida.
- O estado zero está sempre ativo e uma ocorrência é relatada quando o estado $j$ é ativado.
- Os *bytes* do texto comprimido são lidos e a árvore de Huffman é percorrida como antes.
- Cada vez que uma folha da árvore é atingida, sua máscara de *bits* é enviada para o autômato.
- Um estado ativo $i - 1$ irá ativar o estado $i$ apenas se o $i$-ésimo *bit* da máscara estiver ativo.
- Conseqüentemente, o autômato realiza uma transição para cada palavra do texto.



## Esquema geral de pesquisa para a frase "`uma ro* rosa`"




- O autômato pode ser implementado eficientemente por meio do algoritmo Shift-And
- Separadores podem ser ignorados na pesquisa de frases.
- Da mesma maneira, os artigos, preposições etc., também podem ser ignorados se for conveniente.
- Neste caso, basta ignorar as folhas correspondentes na árvore de Huffman quando a pesquisa chega a elas.
- É raro encontrar esta possibilidade em sistemas de pesquisa *on-line*.

# Tempos de pesquisa (em segundos) para o arquivo WSJ, com intervalo de confiança de 99%

| Algoritmo | $k = 0$ | $k = 1$ | $k = 2$ | $k = 3$ |
|---|---|---|---|---|
| Agrep | 23,8 $\pm$ 0,38 | 117,9 $\pm$ 0,14 | 146,1 $\pm$ 0,13 | 174,6 $\pm$ 0,16 |
| Pesquisa direta | 14,1 $\pm$ 0,18 | 15,0 $\pm$ 0,33 | 17,0 $\pm$ 0,71 | 22,7 $\pm$ 2,23 |
| Pesquisa com autômato | 22,1 $\pm$ 0,09 | 23,1 $\pm$ 0,14 | 24,7 $\pm$ 0,21 | 25,0 $\pm$ 0,49 |