Processamento de Cadeias de Caracteres*

Última alteração: 21 de Setembro de 2004

*Transparências elaboradas por Fabiano Cupertino Botelho, Charles Ornelas Almeida 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 $T[1..n]$ de tamanho n;
- Padrão: arranjo $P[1..m]$ de tamanho $m \leq n$.
- Os elementos de P e T são escolhidos de um alfabeto finito Σ 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

```
1     7        16    22  26
Texto exemplo. Texto tem palavras.
 36       45      53
 Palavras exercem fascínio.
```

```
 exemplo 7
 exercem 45
 fascínio53
 palavras26 36
 tem     22
 texto   1 16
```

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 $V = Kn^{\beta} = O(n^{\beta})$, onde K e β dependem das características de cada texto.
- K geralmente assume valores entre 10 e 100, e β 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

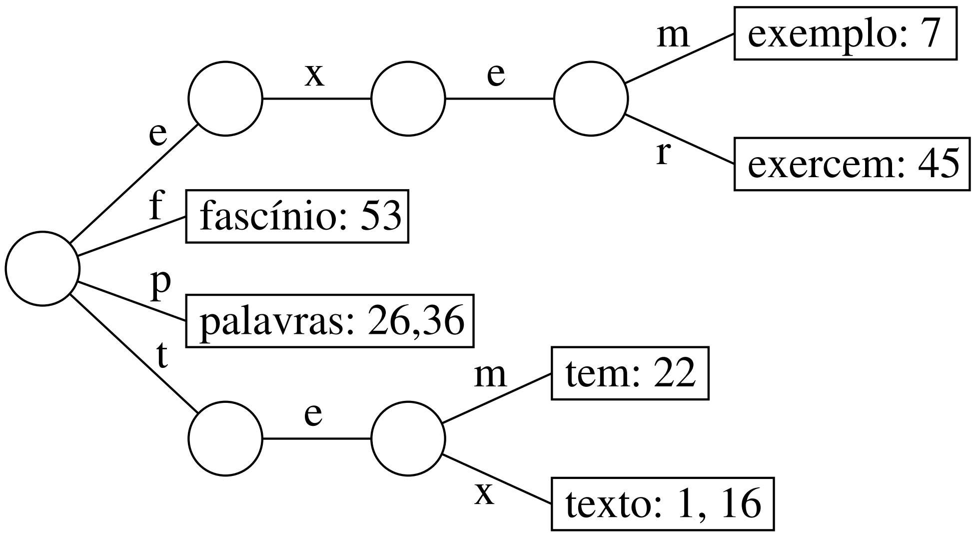
- A pesquisa tem geralmente três passos:
  - Pesquisa no vocabulário: palavras e padrões da consulta são isoladas e pesquisadas no vocabulário.
  - Recuperação das ocorrências: as listas de ocorrências das palavras encontradas no vocabulário são recuperadas.
  - Manipulação das ocorrências: as listas 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 $O(\log n)$.
- 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.

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.

```
procedure ForcaBruta (var T: TipoTexto; var n: integer;
                      var P: TipoPadrao; var m: integer);
{-- Pesquisa P[1..m] em T[1..n] --}
var i, j, k: Integer;
begin
  for i := 1 to n - m + 1 do
    begin
    k := i; j := 1;
    while T[k] = P[j] do
      begin j := j + 1; k := k + 1; end;
    if j>m then writeln ('Casamento na posicao', i:3);
    end;
end;
```

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 $\Sigma \cup \{\epsilon\}$, onde Σ é o alfabeto finito de entrada e ϵ 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 $\alpha \in \Sigma \cup \{\epsilon\}$.

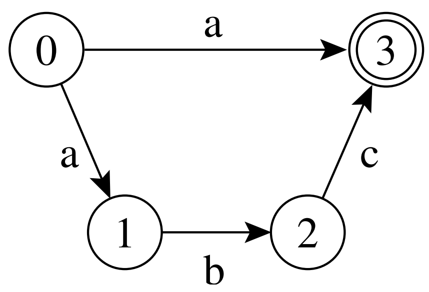
Tipos de Autômatos

- **Autômato finito não-determinista**:
  - Quando $\mathcal{T}$ é tal que existe um estado q associado a um dado caractere α 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 ϵ.
  - 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

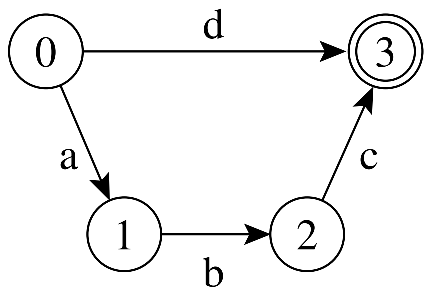
- Autômato finito não-determinista.

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



- Autômato finito determinista.

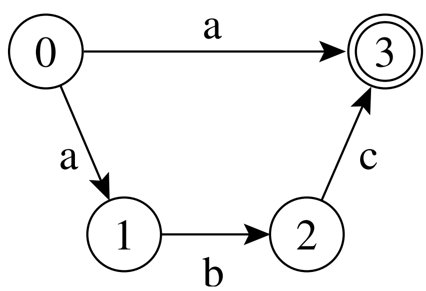
  Para cada caractere de transição todos os estados levam a um único estado.



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

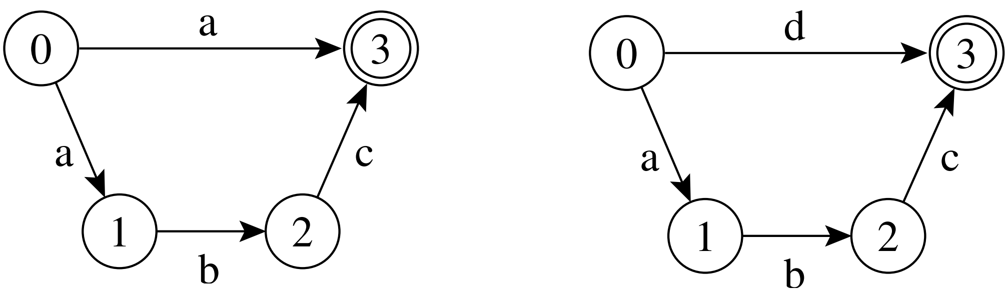
- São transições rotulada com uma cadeia vazia ϵ, também chamadas de **transições-ϵ**, em autômatos não-deterministas
- Não há necessidade de se ler um caractere para caminhar através de uma transição vazia.
- Simplificam a construção do autômato.
- Sempre existe um autômato equivalente que reconhece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

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

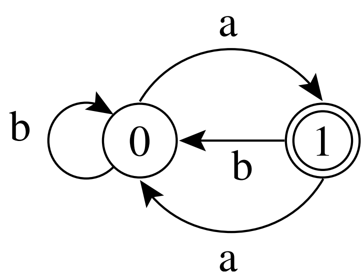
- Os autômatos abaixo são **acíclicos** pois as transições não formam ciclos.



- **Autômatos finitos cíclicos**, deterministas ou não-deterministas, são úteis para **casamento de expressões regulares**

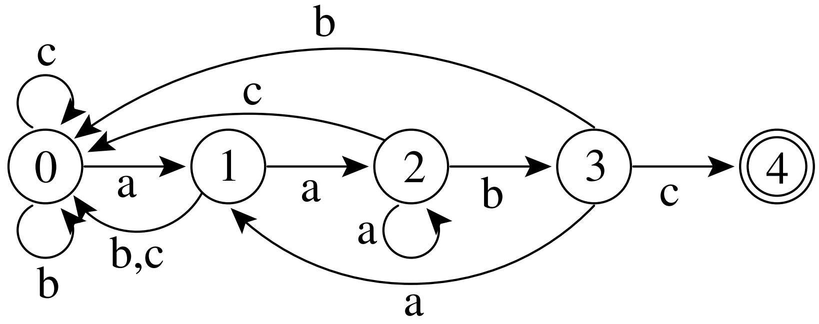
- A linguagem reconhecida por um autômato cíclico pode ser infinita.

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



Exemplo de Uso de Autômato

- O autômato abaixo reconhece P ={aabc}.



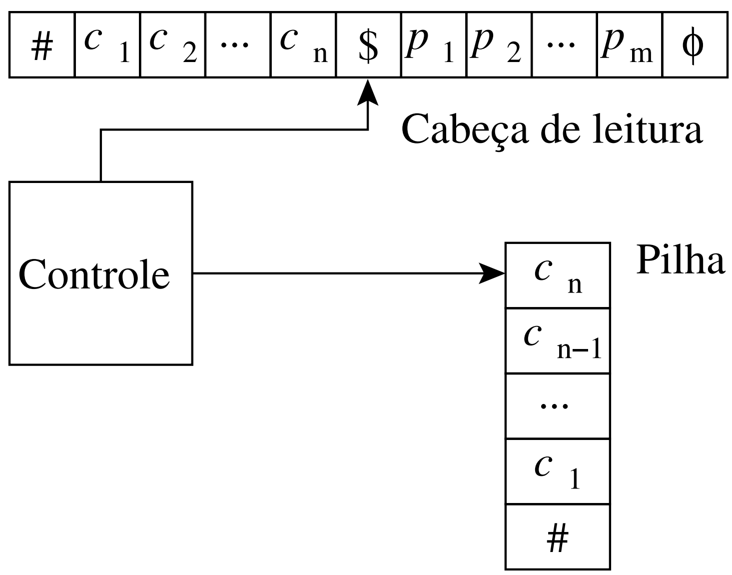
- A pesquisa de P sobre um texto T com alfabeto Σ ={a, b, c} pode ser vista como a simulação do autômato na pesquisa de P sobre T.
- No início, o estado inicial ativa o estado 1.
- Para cada caractere lido do texto, a aresta correspondente é seguida, ativando o estado destino.
- Se o estado 4 estiver 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)

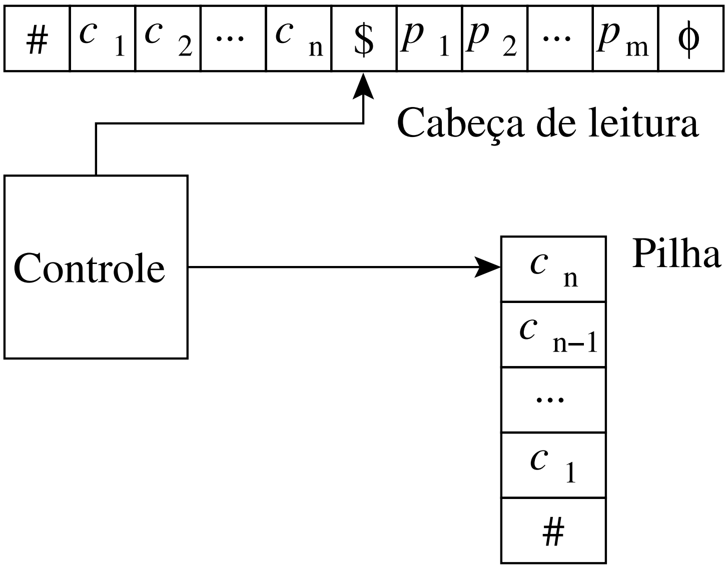
- O KMP é o primeiro algoritmo (1977) cujo pior caso tem complexidade de tempo linear no tamanho do texto, $O(n)$.
- É um dos algoritmos mais famosos para resolver o problema de casamento de cadeias.
- Tem implementação complicada e na prática perde em eficiência para o Shift-And e o Boyer-Moore-Horspool.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

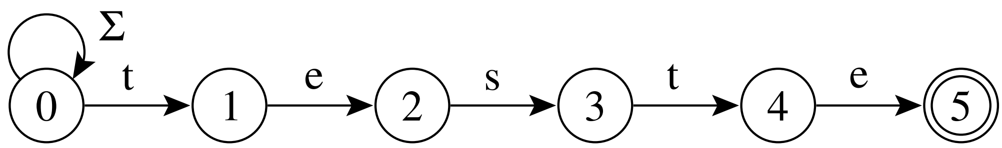
Shift-And - Notação para Operações Usando Paralelismo de *bit*

- Para denotar **repetição de *bit*** é usado exponenciação: $01^3 = 0111$.
- Uma seqüência de *bits* $b_1 \ldots b_c$ 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, $b_1, b_2, \ldots, b_{c-1}, b_c >> 2 = 00b_3, \ldots, b_{c-2}$).

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* $R = (b_1, b_2, \ldots, b_m)$.
- 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 $R = (b_1, b_2, \ldots, b_m)$ se e somente se $p_1 \ldots p_j$ é um sufixo de $t_1 \ldots t_i$, onde i corresponde à posição corrente no texto.
- A j-ésima posição de R é dita estar *ativa*.
- b_m ativo significa um casamento.
- R', o novo valor do conjunto R, é calculado na leitura do próximo caractere t_{i+1}.
- A posição $j + 1$ em R' ficará ativa se e somente se a posição j estava ativa em R ($p_1 \ldots p_j$ era sufixo de $t_1 \ldots t_i$ 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* $b_1 \ldots, b_m$ para cada caractere.

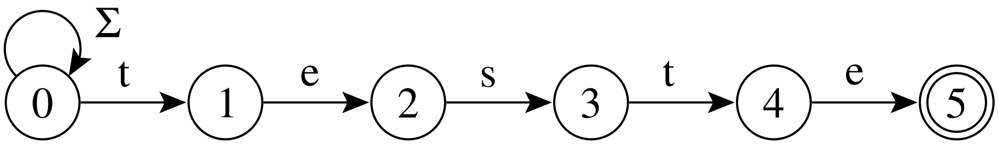
  Ex.: máscaras de *bits* para os caracteres presentes em P ={teste}.

	1	2	3	4	5
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 1 e 4 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 ϵ 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	$(R >> 1)\|10^{m-1}$					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	0	0	0
	1	0	0	0	0	0	0	0	0	0

Shift-And - Implementação

Shift-And ($P = p_1 p_2 \ldots p_m,\ T = t_1 t_2 \ldots t_n$);
  {–– Pré–processamento––}
  for $c \in \Sigma$ **do** $M[c] := 0^m$;
  for j := 1 **to** m **do** $M[p_j] := M[p_j] \mid 0^{j-1}10^{m-j}$;
  {–– Pesquisa––}
  $R := 0^m$;
  for i := 1 **to** n **do**
    $R = ((R >> 1) \mid 10^{m-1}) \ \&\ M[T[i]]$;
    if $R \ \&\ 0^{m-1}1 \neq 0^m$ **then** 'Casamento na posicao $i - m + 1$';

- As operações *and*, *or*, deslocamento à direita e complemento a serem realizadas sobre o conjunto R, não podem ser realizadas com eficiência na linguagem Pascal padrão, o que compromete o conceito de paralelismo de *bit*.
- **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}.

```
1 2 3 4 5 6 7 8 9 0 1 2
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 6, da direita para a esquerda, existe uma colisão na posição 4 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 7 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`}.

```
1 2 3 4 5 6 7 8 9 0 1 2
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 6, da direita para a esquerda, existe uma colisão na posição 4 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, $d[x] = min\{j talque j = m | (1 \leq j < m \& P[m - j] = x)\}$.

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

```
procedure BMH (var T: TipoTexto; var n: integer;
               var P: TipoPadrao; var m: integer);
{–– Pesquisa P[1..m] em T[1..n] ––}
var i, j, k: Integer;
    d: array[0..MaxTamAlfabeto] of integer;
begin
  {–– Pre–processamento do padrao ––}
  for j := 0 to MaxTamAlfabeto do d[j] := m;
  for j := 1 to m–1 do d[ord(P[j])] := m–j;
  i := m;
  while i <= n do {–– Pesquisa ––}
    begin
    k := i; j := m;
    while (T[k] = P[j]) and (j>0) do
      begin k := k–1; j := j–1; end;
    if j = 0 then writeln(' Casamento na posicao: ',k+1:3);
    i := i + d[ord(T[i])];
    end;
end;
```

- d[ord(T[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 $d[x] = min\{j \text{ tal que } j = m \mid (1 \leq j \leq m \;\&\; P[m+1-j] = x)\}$.
- Para o padrão $P = \texttt{teste}$, 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 até n, com incrementos d[ord(T[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.

```
procedure BMHS (var T: TipoTexto; var n: integer;
                var P: TipoPadrao; var m: integer);
var i, j, k: Integer;
    d: array[0..MaxChar] of integer;
begin
  {-- Pre-processamento do padrao--}
  for j := 0 to MaxChar do d[j] := m+1;
  for j := 1 to m do d[ord(P[j])] := m + 1 - j;
  i := m;
  while i <= n do {-- Pesquisa--}
    begin
    k := i; j := m;
    while (T[k] = P[j]) and (j>0) do
      begin k := k - 1; j := j - 1; end;
    if j = 0 then writeln('Casamento na posicao: ', k+1:3);
    i := i + d[ord(T[i+1])];
    end;
end;
```

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 $O(n + rm)$, onde r é igual ao número total de casamentos, o que torna o algoritmo ineficente quando o número de casamentos é grande.
- 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 $O(c)$.
- 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.

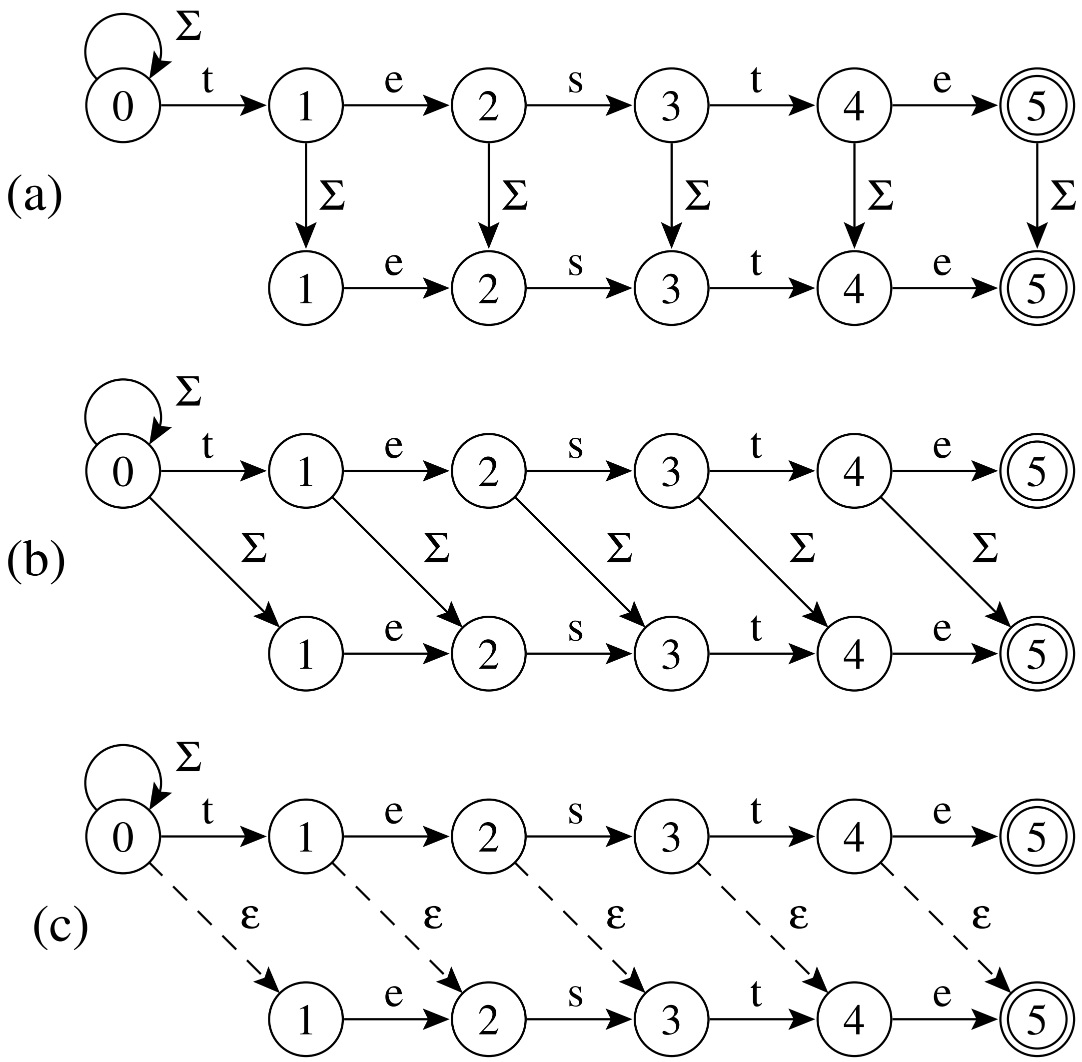
  Ex.: $ed(\texttt{t}este, \texttt{e}stende) = 4$, valor obtido por meio de uma retirada do primeiro `t` de P e a inserção dos 3 caracteres `nde` ao final de P.
- O problema do casamento aproximado de cadeias é o de encontrar todas as ocorrências em T de cada P' que satisfaz $ed(P, P') \leq k$.

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

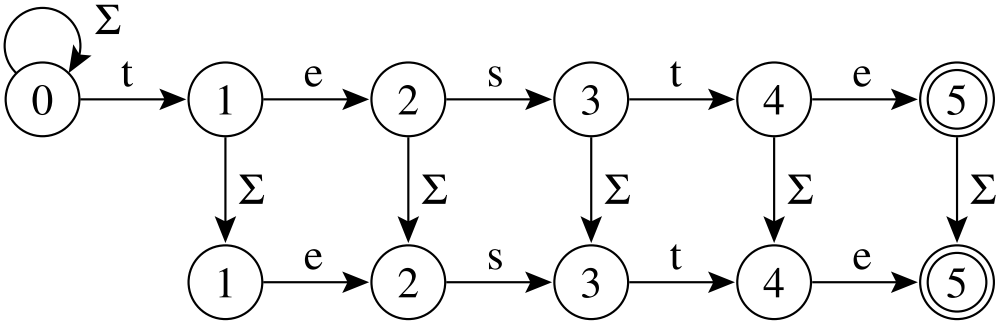
- P ={`teste`} e $k = 1$: (a)inserção; (b)substituição e (c)retirada.



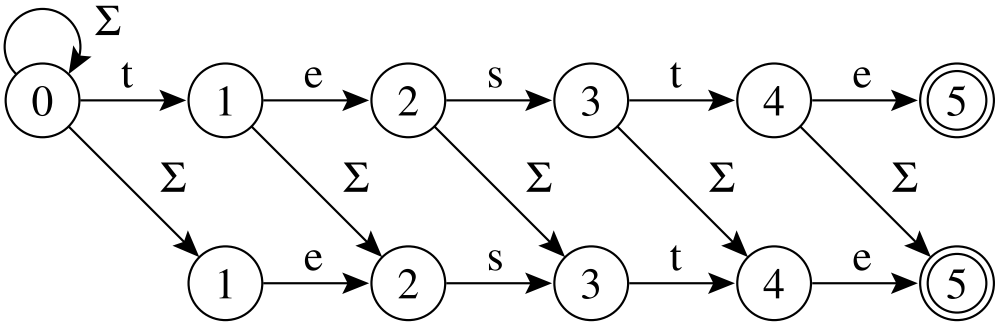
- Casamento de caractere é representado por uma aresta horizontal. Avançamos em P e T.
- O *self-loop* permite que uma ocorrência se inicie em qualquer posição em T.

Exemplo de Autômato para Casamento Aproximado

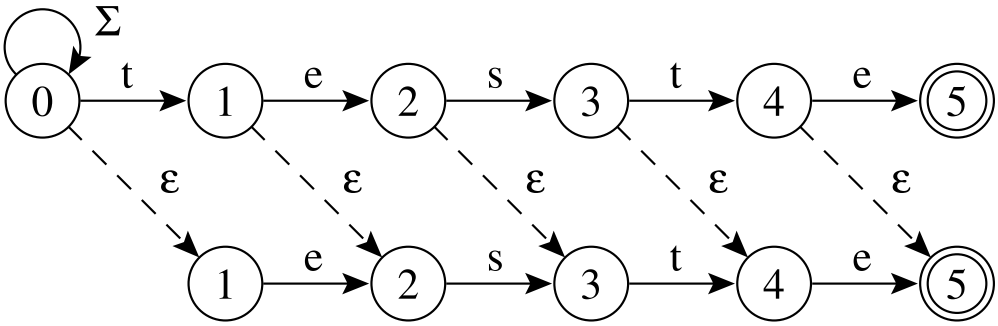
- Uma aresta vertical insere um caractere no padrão. Avançamos em T mas não em P.



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

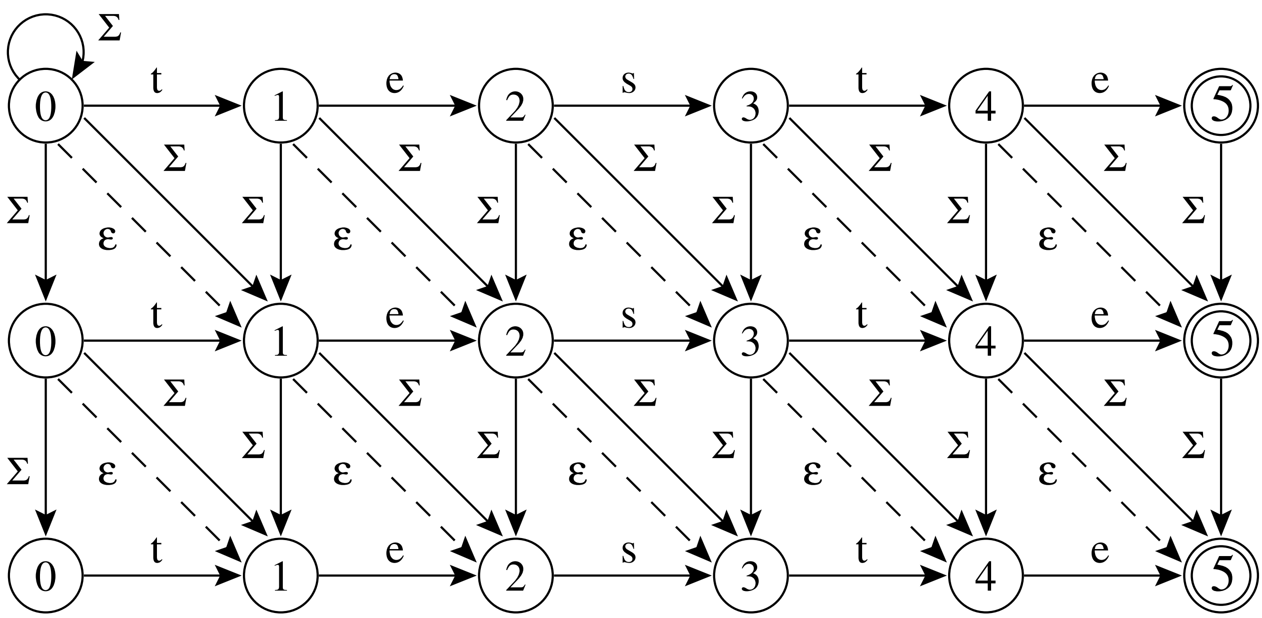


- Uma aresta diagonal tracejada retira um caractere. Avançamos em P mas não em T (transição-ϵ)



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:
  - $R'_0 = ((R_0 >> 1) \mid 10^{m-1})\ \&\ M[T[i]]$
  - $R'_j = ((R_j >> 1)\ \&\ M[T[i]]) \mid R_{j-1} \mid (R_{j-1} >> 1) \mid (R'_{j-1} >> 1)$, 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

```
Shift-And-Aproximado ($P = p_1 p_2 \ldots p_m$, $T = t_1 t_2 \ldots t_n, k$);
  {–– Pré–processamento––}
  for $c \in \Sigma$ do $M[c] := 0^m$;
  for j := 1 to m do $M[p_j] := M[p_j] \mid 0^{j-1}10^{m-j}$;
  {–– Pesquisa––}
  for j := 0 to k do $R_j := 1^j 0^{m-j}$;
  for i := 1 to n do
    $Rant := R_0$;
    $Rnovo := ((Rant >> 1) \mid 10^{m-1})\ \&\ M[T[i]]$;
    $R_0 := Rnovo$;
    for j := 1 to k do
      $Rnovo := ((R_j >> 1\ \&\ M[T[i]]) \mid Rant \mid ((Rant \mid Rnovo) >> 1)$;
      $Rant := R_j$;
      $R_j := Rnovo$;
    if $Rnovo\ \&\ 0^{m-1}1 \neq 0^m$ then 'Casamento na posicao $i$';
```

- As operações *and*, *or*, deslocamento à direita e complemento, a serem realizadas sobre o conjunto R, não podem ser realizadas com eficiência na linguagem Pascal padrão, o que compromete o conceito de paralelismo de *bit*.

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 $R'_1 = (R_1 >> 1)|10^{m-1} \& M[T[i]]|R_0$
- Uma ocorrência exata na posição 8 (“`e`”) e duas, permitindo uma inserção, nas posições 9 e 12 (“`s`” e “`e`”, respectivamente).

Texto	$(R_0 >> 1)\|10^{m-1}$	R'_0	$(R_1 >> 1)\|10^{m-1}$	R'_1
`o`	1 0 0 0 0	0 0 0 0 0	1 1 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
	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	1 0 0 0 0	1 0 0 0 0
`e`	1 1 0 0 0	0 1 0 0 0	1 1 0 0 0	1 1 0 0 0
`s`	1 0 1 0 0	0 0 1 0 0	1 1 1 0 0	0 1 1 0 0
`t`	1 0 0 1 0	1 0 0 1 0	1 0 1 1 0	1 0 1 1 0
`e`	1 1 0 0 1	0 1 0 0 ***1***	1 1 0 1 1	1 1 0 1 ***1***
`s`	1 0 1 0 0	0 0 1 0 0	1 1 1 0 1	0 1 1 0 ***1***
	1 0 0 0 0	0 0 0 0 0	1 0 1 1 0	0 0 1 0 0
`t`	1 0 0 0 0	1 0 0 0 0	1 0 0 1 0	1 0 0 1 0
`e`	1 1 0 0 0	0 1 0 0 0	1 1 0 0 1	1 1 0 0 ***1***
`s`	1 0 1 0 0	0 0 1 0 0	1 1 1 0 0	0 1 1 0 0
`t`	1 0 0 1 0	1 0 0 1 0	1 0 1 1 0	1 0 1 1 0
`a`	1 1 0 0 1	0 0 0 0 0	1 1 0 1 1	1 0 0 1 0
`m`	1 0 0 0 0	0 0 0 0 0	1 1 0 0 1	0 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.
- $R'_0 = (R_0 >> 1)|10^{m-1} \& M[T[i]]$. $R'_1 = (R_1 >> 1)|10^{m-1} \& M[T[i]]|R_0|(R'_0 >> 1)|(R_0 >> 1)$.
- Uma ocorrência exata na posição 8 (“`e`”) e cinco, permitindo um erro, nas posições 7, 9, 12, 14 e 15 (“`t`”, “`s`”, “`e`”, “`t`” e “`a`”, respec.).

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

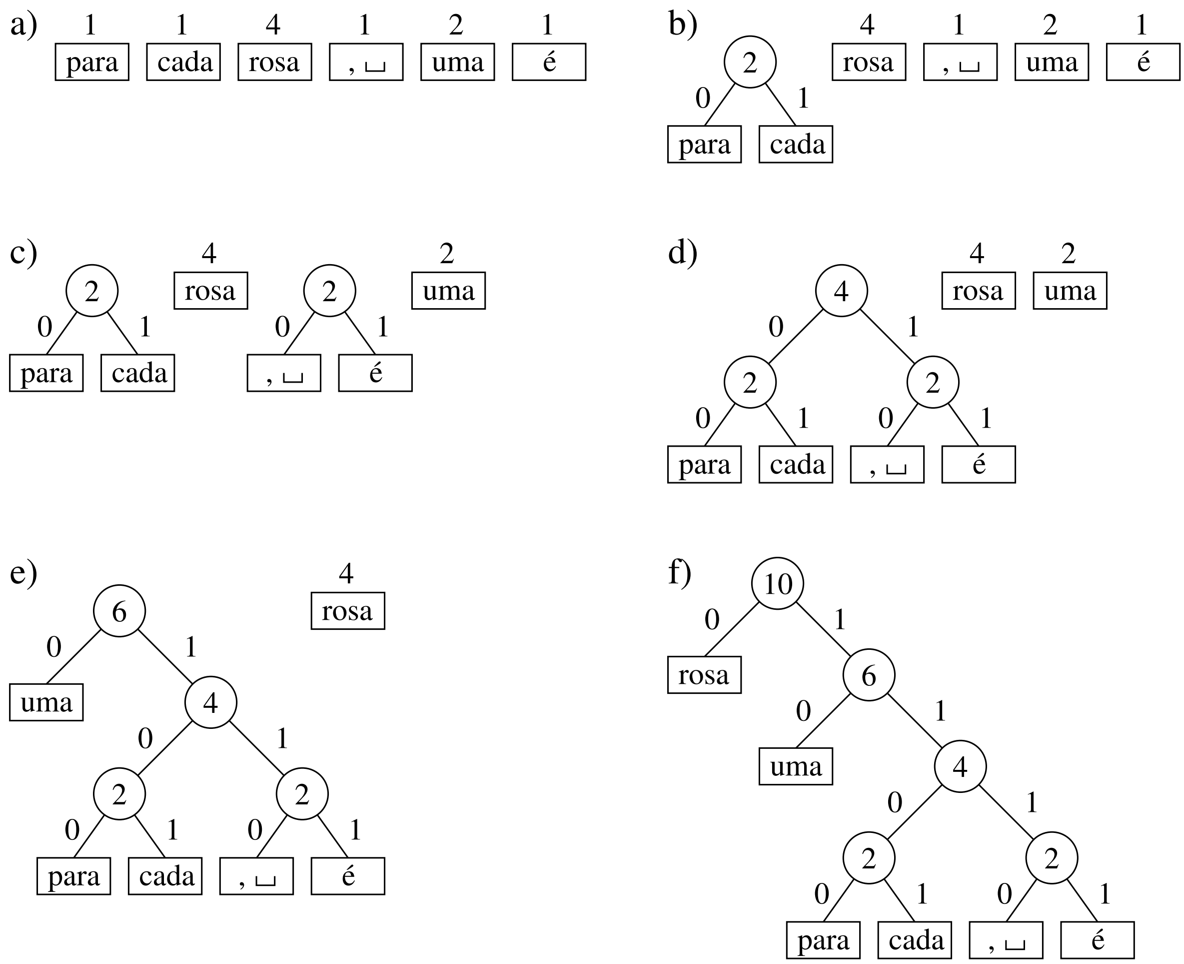
- Técnica de compressão mais eficaz para textos em linguagem natural.
- O método considera cada palavra diferente do texto como um símbolo.
- Conta suas freqüências e gera um código de Huffman para as palavras.
- A seguir, comprime o texto substituindo cada palavra pelo seu código.
- Assim, a compressão 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.

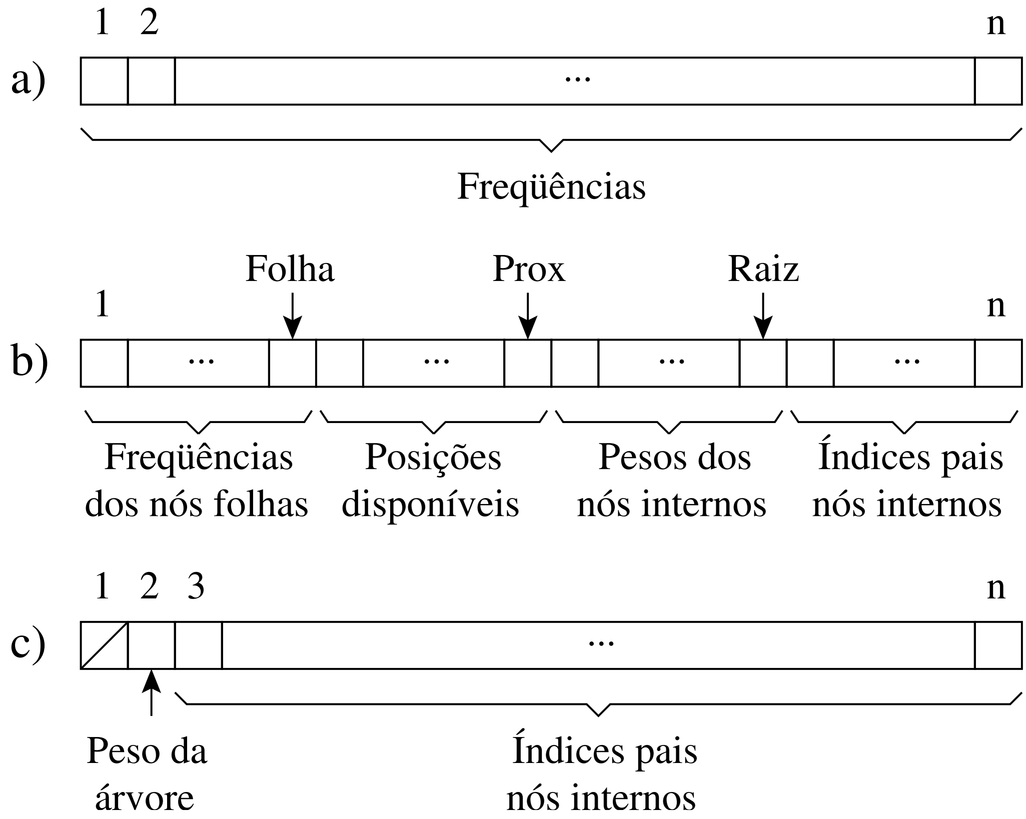
- Freqüências relativas à frase exemplo: "`para cada rosa rosa, uma rosa é uma rosa`"

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 $[1, 2, 4, 4, 4, 4]$.

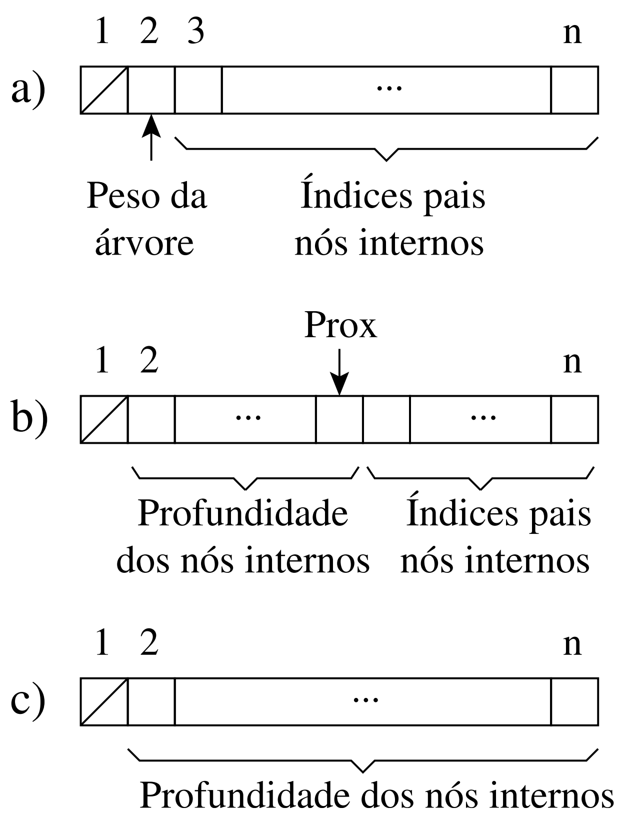
Pseudocódigo para a Primeira Fase

```
function PrimeiraFase (A, n);
begin
  Raiz := n; Folha := n;
  for Prox := n downto 2 do
    begin
    { Procura Posicao }
    if ((nao existe Folha) or ((Raiz > Prox) and
         (A[Raiz] <= A[Folha])))
    then begin { No interno }
           A[Prox] := A[Raiz]; A[Raiz] := Prox;
           Raiz:= Raiz − 1;
           end
    else begin { No folha }
           A[Prox] := A[Folha]; Folha := Folha − 1;
           end;
    { Atualiza Frequencias }
    if ((nao existe Folha) or ((Raiz > Prox) and
        (A[Raiz] <= A[Folha])))
    then begin { No interno }
           A[Prox] := A[Prox] + A[Raiz]; A[Raiz] := Prox;
           Raiz := Raiz − 1;
           end
    else begin { No folha }
           A[Prox] := A[Prox] + A[Folha]; Folha := Folha − 1;
           end;
    end;
end;
```

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



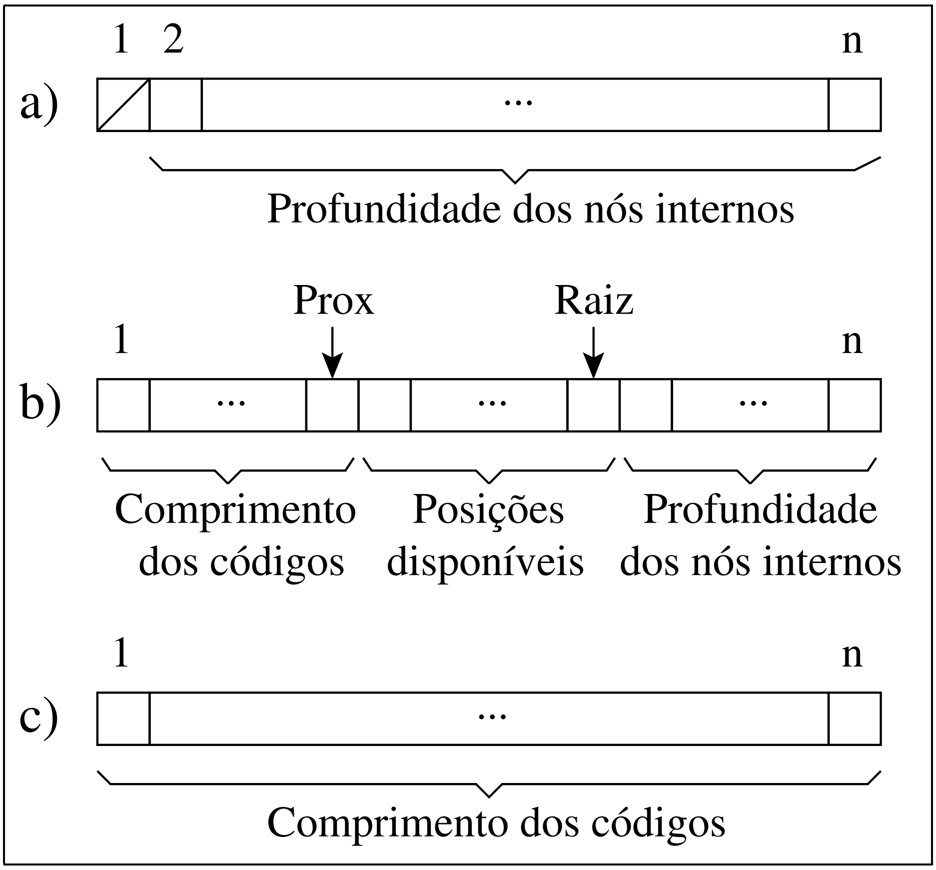
Pseudocódigo para a Segunda Fase

```
function SegundaFase (A, n);
begin
  A[2] := 0;
  for Prox := 3 to n do A[Prox] := A[A[Prox]] + 1;
end;
```

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

```
function TerceiraFase (A, n);
begin
  Disp := 1; u := 0; h := 0; Raiz := 2; Prox := 1;
  while Disp > 0 do
    begin
    while (Raiz <= n) and (A[Raiz] = h) do
      begin u := u + 1; Raiz := Raiz + 1 end;
    while Disp > u do
      begin
      A[Prox] := h; Prox := Prox + 1;
      Disp := Disp – 1;
      end;
    Disp := 2 * u; h := h + 1; u := 0;
    end;
end;
```

- 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

```
function CalculaCompCodigo (A, n);
begin
  A := PrimeiraFase (A, n);
  A := SegundaFase (A, n);
  A := TerceiraFase (A, n);
end;
```

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

```
function Codifica (Base, Offset, i, MaxCompCod);
begin
  c := 1;
  while ( i >= Offset[c + 1] ) and
        (c + 1 <= MaxCompCod ) do c := c + 1;
  Codigo := i − Offset[c] + Base[c];
end;
```

Obtenção do código:

- Parâmetros: vetores Base e Offset, índice i do símbolo (Tabela da transparência 71) 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

```
function Decodifica (Base, Offset, ArqComprimido, MaxCompCod);
begin
  c := 1;
  Codigo := LeBit (ArqComprimido);
  while ( Codigo << 1 ) >= Base[c + 1]) and
        ( c + 1 <= MaxCompCod ) do
    begin
    Codigo := (Codigo << 1) or LeBit (ArqComprimido);
    c := c + 1;
    end;
  i := Codigo – Base[c] + Offset[c];
end;
```

- Parâmetros: vetores Base e Offset, o arquivo comprimido e 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 $i = \text{Codigo} - \text{Base}[c] + \text{Offset}[c]$.

Pseudocódigo para realizar a compressão

```
function Compressao (ArqTexto, ArqComprimido);
begin
  { Primeira etapa }
  while not Eof (ArqTexto) do
    begin
    Palavra := ExtraiProximaPalavra (ArqTexto);
    Pos := Pesquisa (Palavra, Vocabulario);
    if Pos é uma posicao valida
    then Vocabulario[Pos].Freq := Vocabulario[Pos].Freq + 1
    else Insere (Palavra, Vocabulario);
    end;
  { Segunda etapa }
  Vocabulario := OrdenaPorFrequencia (Vocabulario);
  Vocabulario := CalculaCompCodigo (Vocabulario, n);
  ConstroiVetores (Base, Offset, ArqComprimido);
  Grava (Vocabulario, ArqComprimido);
  LeVocabulario (Vocabulario, ArqComprimido);
  { Terceira etapa }
  PosicionaPrimeiraPosicao (ArqTexto);
  while not Eof (ArqTexto) do
    begin
    Palavra := ExtraiProximaPalavra (ArqTexto);
    Pos := Pesquisa (Palavra, Vocabulario);
    Codigo := Codifica(Base,Offset,Vocabulario[Pos].Ordem,
                       MaxCompCod);
    Escreve (ArqComprimido, Codigo);
    end;
end;
```

Pseudocódigo para realizar a descompressão

```
function Descompressao (ArqTexto, ArqComprimido);
begin
  LerVetores (Base, Offset, ArqComprimido);
  LeVocabulario (Vocabulario, ArqComprimido);
  while not Eof (ArqComprimido) do
    begin
    i := Decodifica (Base, Offset, ArqComprimido, MaxCompCod);
    Grava (Vocabulario[i], ArqTexto);
    end;
end;
```

Codificação de Huffman Usando Bytes

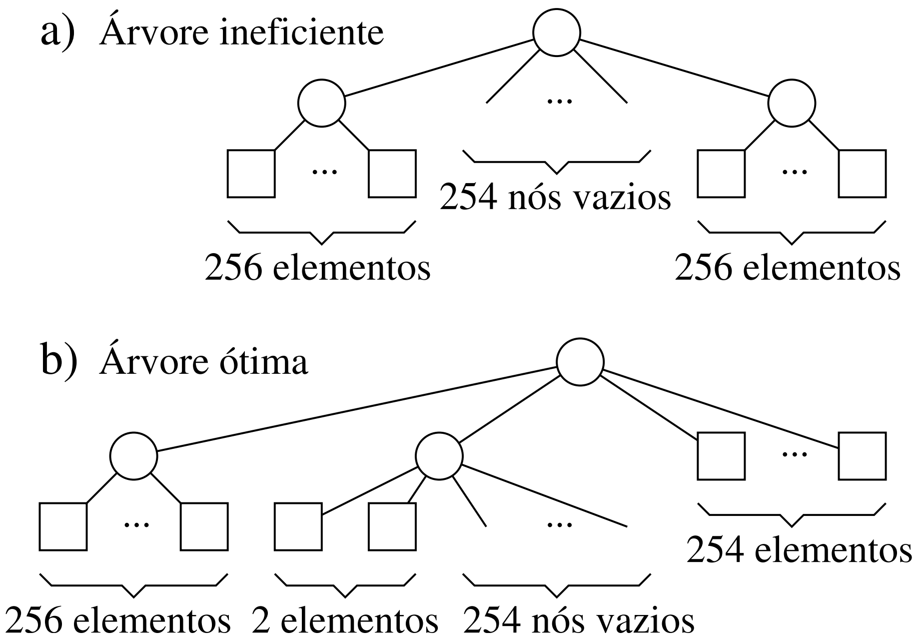
- O método original proposto por Huffman (1952) tem sido usado como um código binário.
- Moura, Navarro, Ziviani e Baeza-Yates (2000) modificaram a atribuição de códigos de tal forma que uma seqüência de *bytes* é associada a cada palavra do texto.
- Conseqüentemente, o grau de cada nó passa de 2 para 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 $1 + ((512 - 256) \mod 255) = 2$.

Generalização do Cálculo dos Comprimentos dos Códigos

```
procedure CalculaCompCodigo (var A: TipoDicionario; n: integer);
var u,       { Nos internos usados }
    h,       { Altura da arvore }
    NoInt,   { Numero de nos internos }
    Prox, Raiz, Folha, Disp, x, Resto: integer;
begin
  if n > (BaseNum - 1)
  then Resto := 1 + ((n - BaseNum) mod (BaseNum - 1))
  else Resto := n - 1;
  NoInt := 1 + ((n - Resto) div (BaseNum - 1));
  if Resto < 2 then Resto := BaseNum;
  for x := (n - 1) downto (n - Resto + 1) do
    A[n].Freq := A[n].Freq + A[x].Freq;
  { Primeira Fase }
  Raiz := n; Folha := n - Resto;
  for Prox := n - 1 downto (n - NoInt + 1) do
    begin
    { Procura Posicao }
    if ((Folha<1) or ((Raiz>Prox) and (A[Raiz].Freq<=A[Folha].Freq)))
    then begin { No interno }
         A[Prox].Freq := A[Raiz].Freq; A[Raiz].Freq := Prox;
         Raiz := Raiz - 1;
         end
    else begin { No-folha }
         A[Prox].Freq := A[Folha].Freq;
         Folha := Folha - 1;
         end;
```

Generalização do Cálculo dos Comprimentos dos Códigos

```
   { Atualiza Frequencias }
   for x := 1 to (BaseNum - 1) do
     begin
     if ((Folha<1) or ((Raiz>Prox) and
        (A[Raiz].Freq<=A[Folha].Freq)))
     then begin { No interno }
          A[Prox].Freq := A[Prox].Freq + A[Raiz].Freq;
          A[Raiz].Freq := Prox; Raiz := Raiz - 1;
          end
     else begin { No-folha }
          A[Prox].Freq := A[Prox].Freq+A[Folha].Freq;
          Folha := Folha-1;
          end;
     end;
   end;
 { Segunda Fase }
 A[Prox].Freq := 0; Raiz := Prox;
 for Prox := Raiz + 1 to n do
   A[Prox].Freq := A[A[Prox].Freq].Freq + 1;
```

Generalização do Cálculo dos Comprimentos dos Códigos

```
  { Terceira Fase }
  Disp := 1; u := 0; h := 0; Prox := 1;
  while Disp > 0 do
    begin
    while (Raiz <= n) and (A[Raiz].Freq = h) do
      begin
      u := u + 1; Raiz := Raiz + 1
      end;
    while Disp > u do
      begin
      A[Prox].Freq := h; Prox := Prox + 1; Disp := Disp - 1;
      if Prox > n then begin u := 0; break end
      end;
    Disp := BaseNum * u; h := h + 1; u := 0;
    end;
end;
```

OBS: A constante 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 mais sensível está no código inserido antes da primeira fase, o qual tem como função 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 é alterada para indicar quantos nós estão disponíveis em cada nível, o que é representado pela variável Disp.

Codificação orientada a *bytes*

```
function Codifica (var VetoresBaseOffset: TipoVetoresBO; Ordem: integer;
                   var c: integer; MaxCompCod: integer): integer;
begin
  c := 1;
  while (Ordem >= VetoresBaseOffset[c + 1].Offset) and
        (c + 1 <= MaxCompCod) do c := c + 1;
  Codifica := Ordem − VetoresBaseOffset[c].Offset +
              VetoresBaseOffset[c].Base;
end;
```

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

Decodificação orientada a *bytes*

```
function Decodifica (var VetoresBaseOffset: TipoVetoresBO;
                     var ArqComprimido: TipoArqResult;
                     MaxCompCod: integer): integer;
var c, Codigo, CodigoTmp, LogBase2: integer;
begin
  LogBase2 := Round (Ln(BaseNum)/Ln(2));
  c := 1;
  read(ArqComprimido, Codigo);
  Codigo := Codigo − 128; { remove o bit de marcacao }
  while ((c + 1) <= MaxCompCod) and
        ((Codigo shl LogBase2) >= VetoresBaseOffset[c+1].Base) do
    begin
    read(ArqComprimido, CodigoTmp);
    Codigo := (Codigo shl LogBase2) or CodigoTmp;
    c := c + 1;
    end;
  Decodifica := Codigo − VetoresBaseOffset[c].Base +
                VetoresBaseOffset[c].Offset
end;
```

Alterações:

1. Permitir a leitura *byte* a *byte* do arquivo comprimido, em vez de *bit* a *bit*.
2. O 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, 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 na relação abaixo:

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

Construção dos vetores Base e Offset

```
function ConstroiVetores (var VetoresBaseOffset: TipoVetoresBO;
                          var Vocabulario: TipoDicionario; n: integer;
                          var ArqComprimido: TipoArqResult): integer;
var Wcs: array[1..MaxTamVetoresBO] of integer;
    i, MaxCompCod: integer;
begin
  MaxCompCod := Vocabulario[n].Freq;
  for i := 1 to MaxCompCod do Wcs[i] := 0;
  for i := 1 to n do
    begin
    Wcs[Vocabulario[i].Freq] := Wcs[Vocabulario[i].Freq] + 1;
    VetoresBaseOffset[Vocabulario[i].Freq].Offset := i −
      Wcs[Vocabulario[i].Freq] + 1;
    end;
  VetoresBaseOffset[1].Base := 0;
  for i := 2 to MaxCompCod do
  begin
    VetoresBaseOffset[i].Base := BaseNum*(VetoresBaseOffset[i−1].Base +
                                 Wcs[i−1]);
    if VetoresBaseOffset[i].Offset = 0
    then VetoresBaseOffset[i].Offset := VetoresBaseOffset[i−1].Offset
  end;
  { Salvar as tabelas em disco }
  GravaNumInt (ArqComprimido, MaxCompCod);
  for i := 1 to MaxCompCod do
    begin
    GravaNumInt(ArqComprimido, VetoresBaseOffset[i].Base);
    GravaNumInt(ArqComprimido, VetoresBaseOffset[i].Offset);
    end;
  ConstroiVetores := MaxCompCod;
end;
```

Procedimentos para ler e para escrever números inteiros em um arquivo de *bytes*

```
function LeNumInt (var ArqComprimido: TipoArqResult): integer;
var i, Num, NumResp: integer;
begin
  NumResp := 0;
  for i := sizeof (integer) - 1 downto 0 do
    begin
    read (ArqComprimido, Num); Num := Num shl (i * 8);
    NumResp := NumResp or Num;
    end;
  LeNumInt := NumResp;
end;
procedure GravaNumInt (var ArqComprimido: TipoArqResult; Num: integer);
var i: integer;
begin
  for i := sizeof (integer) - 1 downto 0 do
    write (ArqComprimido, Num shr (i * 8));
end;
```

- O procedimento GravaNumInt grava no disco cada *byte* (da esquerda para a direita) do número inteiro passado como parâmetro.
- O procedimento LeNumInt lê do disco cada *byte* de um número inteiro e o recompõe.

O Por Quê da Existência de LeNumInt e GravaNumInt

- São necessários em razão de a variável ArqComprimido, passada como parâmetro, ter sido declarada no programa principal como um arquivo de *bytes*.

- Isso faz com que o procedimento **write** (**read**) do Pascal escreva (leia) do disco o *byte* mais à direita do número.

- Por exemplo, considere o número 300 representado em 4 *bytes*, como mostrado na Figura abaixo.

- Caso fosse utilizado o procedimento **write**, seria gravado o número 44 em disco, que é o número representado no *byte* mais à direita.

- Um problema análogo ocorre ao se utlizar o procedimento **read** para ler do disco um número inteiro representado em mais de um *byte*.

Byte 0	Byte 1	Byte 2	Byte 3
0 0 0 0 0 0 0 0	0 0 0 0 0 0 0 0	0 0 0 0 0 0 0 1	0 0 1 0 1 1 0 0

Extração do próximo símbolo a ser codificado

```
procedure DefineAlfabeto (var Alfabeto: TipoAlfabeto;
                          var ArqAlf: text);
var Simbolos: String[MaxAlfabeto];
    i: integer;
begin { Os Simbolos devem estar juntos em uma linha no arquivo }
  for i := 0 to MaxAlfabeto do Alfabeto[i] := false;
  readln(ArqAlf, Simbolos);
  for i:=1 to length(Simbolos) do
    Alfabeto[ord(Simbolos[i])+127] := true;
  Alfabeto[0] := false; { caractere de codigo zero: separador }
end;
```

OBS: O procedimento DefineAlfabeto lê de um arquivo "alfabeto.txt" todos o caracteres que serão utilizados para compor palavras.

```
function ExtraiProximaPalavra (var Indice: integer;
                               var Linha: String;
                               var ArqTxt: text;
                               var Alfabeto: TipoAlfabeto):
                               TipoPalavra;
var FimPalavra, Aux: boolean;
begin
  FimPalavra := False;
  Aux := False;
  Result := '';
  if Indice = Length(Linha)
  then
```

Extração do próximo símbolo a ser codificado

```
        if eof(ArqTxt)
        then begin
               Linha := char (0);
               FimPalavra := True
               end
        else begin
               readln (ArqTxt, Linha);
               { Coloca o caractere de fim de linha em Linha }
               Linha := Linha + char(10) + char(0);
               Indice := 1
               end;
   while (Indice <= length (Linha)) and not FimPalavra do
     begin
     if Alfabeto[ord (Linha[Indice]) + 127]
     then begin
            Result := Result + Linha[Indice];
            Aux := true;
            end
     else begin
            if Aux
            then begin if Linha[Indice]<>char(0) then Indice:=Indice−1 end
            else Result := Result + Linha[Indice];
            FimPalavra := True;
            end;
     Indice := Indice + 1;
     end;
end;
```

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

```
procedure Compressao (var ArqTxt, ArqAlf: text;
                      var ArqComprimido: TipoArqResult);
var Alfabeto: TipoAlfabeto; Palavra, Linha: TipoPalavra;
    Ind, MaxCompCod: integer; Vocabulario: TipoDicionario;
    p: TipoPesos; VetoresBaseOffset: TipoVetoresBO;
begin
  { Le alfabeto definido em arquivo }
  DefineAlfabeto(Alfabeto, ArqAlf);
  Ind := 0;  Linha := '';
  {Inicializacao do Vocabulario }
  Inicializa (Vocabulario);
  GeraPesos (p);
  { Inicio da Compressao }
  PrimeiraEtapa (ArqTxt, Alfabeto, Ind, Palavra, Linha,
                 Vocabulario, p);
  MaxCompCod := SegundaEtapa (Vocabulario, VetoresBaseOffset,
                              p, ArqComprimido);
  { Coloca o cursor de leitura no inicio do arquivo}
  Seek (ArqTxt, 0);
  Ind := 0;  Linha := '';
  TerceiraEtapa (ArqTxt, Alfabeto, Ind, Palavra, Linha, Vocabulario,
                 p, VetoresBaseOffset, ArqComprimido, MaxCompCod);
end;
```

Primeira etapa da compressão

```
procedure PrimeiraEtapa (var ArqTxt: text;var Alfabeto: TipoAlfabeto;
                         var Indice: integer; var Palavra, Linha:
                         String; var Vocabulario: TipoDicionario;
                         p: TipoPesos);
var Elemento: TipoItem;
    i: integer;
begin
  repeat
    Palavra := ExtraiProximaPalavra (Indice,Linha,ArqTxt,Alfabeto);
    Elemento.Chave := Palavra + char(0); Elemento.Freq := 1;
    if Palavra <> ''
    then begin
         i := Pesquisa (Elemento.Chave, p, Vocabulario);
         if i < M
         then Vocabulario[i].Freq := Vocabulario[i].Freq + 1
         else Insere (Elemento, p, Vocabulario);
         repeat
           Palavra:=ExtraiProximaPalavra (Indice,Linha,ArqTxt,
                                          Alfabeto);
           Elemento.Chave := Palavra + char(0);
           { O primeiro espaco depois da palavra nao e codificado }
           if (Trim (Palavra) <> '') and (Trim (Palavra) <> char(0))
           then begin
                i := Pesquisa (Elemento.Chave, p, Vocabulario);
                if i < M
                then Vocabulario[i].Freq := Vocabulario[i].Freq + 1
                else Insere (Elemento, p, Vocabulario);
                end
         until Trim (Palavra) = '';
         end
  until Palavra = '';
end;
```

Segunda etapa da compressão

```
function SegundaEtapa (var Vocabulario: TipoDicionario;
                       var VetoresBaseOffset: TipoVetoresBO;
                       var p: TipoPesos;
                       var ArqComprimido: TipoArqResult): integer;
var i, j, NumNodosFolhas, PosArq: integer;  Ch: Char;
    Elemento: TipoItem;  Palavra: TipoPalavra;
begin
  NumNodosFolhas := OrdenaPorFrequencia (Vocabulario);
  CalculaCompCodigo (Vocabulario, NumNodosFolhas);
  SegundaEtapa := ConstroiVetores (VetoresBaseOffset, Vocabulario,
                                   NumNodosFolhas, ArqComprimido);
  { Grava Vocabulario }
  GravaNumInt (ArqComprimido, NumNodosFolhas);
  PosArq := FilePos(ArqComprimido);
  for i := 1 to NumNodosFolhas do
    begin
    j := 1;
    while Vocabulario[i].Chave[j] <> char(0) do
      begin
      write(ArqComprimido, Byte(Vocabulario[i].Chave[j])); j := j + 1;
      end;
    write(ArqComprimido, Byte(char(0)));
    end;
```

Segunda etapa da compressão

```
{ Le e reconstroi a condicao de hash no vetor contendo vocabulario }
 Seek(ArqComprimido, PosArq); Inicializa (Vocabulario);
 for i := 1 to NumNodosFolhas do
   begin
   Palavra := '';
   repeat
     read(ArqComprimido, Byte(Ch));
     if Ch <> char(0)
     then Palavra := Palavra + Ch;
   until Ch = char(0);
   Elemento.Chave := Palavra + char(0); Elemento.Ordem := i;
   j := Pesquisa (Elemento.Chave, p, Vocabulario);
   if j >= M
   then Insere (Elemento, p, Vocabulario);
   end;
end;
```

Função para ordenar o vocabulário por freqüência

- O objetivo dessa função é ordenar *in situ* o vetor Vocabulario, utilizando a própria tabela *hash*.
- Para isso, os símbolos do vetor Vocabulario são copiados para as posições de 1 a n no próprio vetor (n é o número de símbolos presentes no vocabulário) e ordenados de forma não crescente por suas respectivas freqüências de ocorrência.
- O algoritmo de ordenação usado foi o Quicksort alterado para:
  1. Receber como parâmetro uma variável definida como TipoDicionario.
  2. Mudar a condição de ordenação para não crescente.
  3. Fazer com que a chave de ordenação seja o campo que representa as freqüências dos símbolos no arquivo texto.
- A função OrdenaPorFrequencia retorna o número de símbolos presentes no vocabulário.

Função para ordenar o vocabulário por freqüência

```
function OrdenaPorFrequencia(var Vocabulario:TipoDicionario):Indice;
var i, n: Indice;  Item: TipoItem;
begin
  n := 1; Item := Vocabulario[1];
  for i := 0 to M - 1 do
    if Vocabulario[i].Chave <> Vazio
    then if i <> 1
         then begin
              Vocabulario[n] := Vocabulario[i];  n := n + 1;
              end;
  if Item.Chave <> Vazio
  then Vocabulario[n] := Item
  else n := n - 1;
  Quicksort (Vocabulario, n);
  OrdenaPorFrequencia := n;
end;
```

Terceira etapa da compressão

```
procedure TerceiraEtapa(var ArqTxt: text; var Alfabeto: TipoAlfabeto;
                        var Indice: integer; var Palavra,Linha: String;
                        var Vocabulario:TipoDicionario;var p:TipoPesos;
                        var VetoresBaseOffset: TipoVetoresBO;
                        var ArqComprimido: TipoArqResult;
                        MaxCompCod: integer);
var Pos: Apontador;  Chave: TipoChave;  Codigo, c: integer;
begin
repeat
  Palavra := ExtraiProximaPalavra (Indice,Linha,ArqTxt,Alfabeto);
  Chave := Palavra + char(0);
  if Palavra <> ''
  then begin
       Pos := Pesquisa (Chave, p, Vocabulario);
       Codigo := Codifica (VetoresBaseOffset,
                           Vocabulario[Pos].Ordem, c, MaxCompCod);
       Escreve(ArqComprimido, Codigo, c);
       repeat
         Palavra:=ExtraiProximaPalavra(Indice,Linha,ArqTxt,Alfabeto);
         { O primeiro espaco depois da palavra nao e codificado }
         if (Trim (Palavra) <> '') and (Trim (Palavra) <> char(0))
         then begin
              Chave := Palavra + char(0);
              Pos := Pesquisa (Chave, p, Vocabulario);
              Codigo:=Codifica(VetoresBaseOffset,
                               Vocabulario[Pos].Ordem,c, MaxCompCod);
              Escreve(ArqComprimido, Codigo, c);
              end;
       until Trim (Palavra) = '';
       end
until Palavra = '';
end;
```

Procedimento Escreve

- O procedimento Escreve recebe o código e seu comprimento c.
- 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 procedimento Escreve extrai o primeiro *byte* e coloca a marcação no oitavo *bit* fazendo uma operação *or* do *byte* com a constante 128 (que em hexadecimal é 80.)
- Esse *byte* é então colocado na primeira posição do vetor Saida.
- No anel **while**, 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 **for**.

Implementação do Procedimento Escreve

```
procedure Escreve (var ArqComprimido: TipoArqResult;
                   var Codigo, c: integer);
var Saida: array[1..MaxTamVetoresBO] of byte; i,
          cTmp: integer;
begin
  i := 1; cTmp := c;
  Saida[i] := (Codigo shr (8*(c − 1) − c + 1)) or $80;
  i := i + 1; c := c − 1;
  while c > 0 do
    begin
    Saida[i] := Codigo shr (8*(c − 1) − c + 1);
    i := i + 1; c := c − 1;
    end;
  for i := 1 to cTmp do write(ArqComprimido, Saida[i]);
end;
```

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

```
procedure Descompressao (var ArqComprimido: TipoArqResult;
                         var ArqTxt, ArqAlf: text);
var Alfabeto: TipoAlfabeto;
    Ind, MaxCompCod: integer;
    Vocabulario: TipoVetorPalavra;
    VetoresBaseOffset: TipoVetoresBO;
    PalavraAnt: TipoPalavra;
begin
  DefineAlfabeto (Alfabeto, ArqAlf); { Le alfabeto em arquivo }
  MaxCompCod := LeVetores (ArqComprimido, VetoresBaseOffset);
  LeVocabulario (ArqComprimido, Vocabulario);
  Ind := Decodifica (VetoresBaseOffset, ArqComprimido, MaxCompCod);
  write (ArqTxt, Vocabulario[Ind]);
  while not Eof (ArqComprimido) do
    begin
    Ind := Decodifica (VetoresBaseOffset, ArqComprimido,
                       MaxCompCod);
    if (Alfabeto [Ord(Vocabulario [Ind][1]) + 127]) and
       (PalavraAnt[1] <> char(10))
    then write (ArqTxt, ' ');
    PalavraAnt := Vocabulario [Ind];
    write (ArqTxt, Vocabulario[Ind]);
    end;
end;
```

OBS: Observe que na descompressão, o vocabuário é representado por um vetor de símbolos do tipo TipoVetorPalavra.

Procedimentos auxiliares da descompressão

```
function LeVetores (var ArqComprimido: TipoArqResult;
                    var VetoresBaseOffset: TipoVetoresBO): integer;
var MaxCompCod, i: integer;
begin
  MaxCompCod := LeNumInt (ArqComprimido);
  for i := 1 to MaxCompCod do
    begin
    VetoresBaseOffset[i].Base := LeNumInt (ArqComprimido);
    VetoresBaseOffset[i].Offset := LeNumInt (ArqComprimido);
    end;
  LeVetores := MaxCompCod;
end;

function LeVocabulario (var ArqComprimido: TipoArqResult;
                        var Vocabulario: TipoVetorPalavra): integer;
var NumNodosFolhas, i: integer; Palavra: TipoPalavra; Ch: Char;
begin
  NumNodosFolhas := LeNumInt (ArqComprimido);
  for i := 1 to NumNodosFolhas do
    begin
    Palavra := '';
    repeat
      read(ArqComprimido, Byte (Ch));
      if Ch <> char(0) { As palavras estao separadas pelo caratere 0 }
      then Palavra := Palavra + Ch;
    until Ch = char(0);
    Vocabulario[i] := Palavra;
    end;
  LeVocabulario := NumNodosFolhas;
end;
```

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.

Procedimento para realizar busca no arquivo comprimido

```
procedure Busca (var ArqComprimido: TipoArqResult; var ArqAlf: text);
var Alfabeto: TipoAlfabeto;  Vocabulario: TipoVetorPalavra;
    VetoresBaseOffset: TipoVetoresBO;  PalavraAnt, p: TipoPalavra;
    T: TipoTexto;  Padrao: TipoPadrao;
    Ind, Codigo, MaxCompCod, c, Ord, NumNodosFolhas, n: integer;
begin
  {Le o alfabeto definido em arquivo}
  DefineAlfabeto (Alfabeto, ArqAlf);
  MaxCompCod := LeVetores (ArqComprimido, VetoresBaseOffset);
  NumNodosFolhas := LeVocabulario (ArqComprimido, Vocabulario);
  n := 1;
  while not Eof (ArqComprimido) do
    begin read(ArqComprimido, Byte(T[n])); n := n + 1 end;
  repeat
    write('Padrao:'); readln(p);
    for Ind := 1 to NumNodosFolhas do
      if Vocabulario [Ind] = p then Ord := Ind;
    Codigo := Codifica (VetoresBaseOffset, Ord, c, MaxCompCod);
    Atribui(Padrao, Codigo, c);
    BMH (T, n, Padrao, c);
  until p = 's';
end;
```

Procedimento para atribuir o código ao padrão

```
procedure Atribui (var P: TipoPadrao;
                   Codigo, c: integer);
var i, cTmp: integer;
begin
  i := 1; cTmp := c;
  P[i] := Char((Codigo shr (8*(c - 1) - c + 1)) or $80);
  i := i + 1; c := c - 1;
  while c > 0 do
    begin
    P[i] := Char(Codigo shr (8*(c - 1) - c + 1));
    i := i + 1; c := c - 1;
    end;
end;
```

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

```
program huffman;
{-- Entram aqui os tipos do Programa 5.26 --}
{-- Entram aqui os tipos do Programa 8.1 --}
const
  BaseNum = 128; { Base numerica que o algoritmo trabalha }
  MaxAlfabeto = 255; { Constante utilizada em ExtraiProximaPalavra }
  MaxTamVetoresBO = 10;
type
  TipoAlfabeto     = array [0..MaxAlfabeto] of boolean;
  TipoBaseOffset   = record
                       Base  : integer;
                       Offset: integer;
                     end;
  TipoVetoresBO    = array [1..MaxTamVetoresBO] of TipoBaseOffset;
  TipoArqResult    = File of Byte;
  TipoPalavra      = String[255];
  TipoVetorPalavra= array[1..M] of TipoPalavra;
{-- Entra aqui o procedimento GeraPeso do Programa 5.22 --}
{-- Entra aqui a função de transformação do Programa 5.23 --}
{-- Entram aqui os operadores apresentados no Programa 5.27 --}
{-- Entram aqui os procedimentos Particao e--}
{-- Quicksort dos Programas 4.6 e 4.7 --}
var ArqTxt, ArqAlf: text;  ArqComprimido: TipoArqResult;
    NomeArqTxt, Opcao, NomeArqComp: TipoPalavra;
```

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

```
begin
  writeln ('*********************************************************');
  writeln ('*                        Opcoes                         *');
  writeln ('*-------------------------------------------------------*');
  writeln ('* (c) Compressao                                        *');
  writeln ('* (d) Descompressao                                     *');
  writeln ('* (p) Pesquisa no texto comprimido                      *');
  writeln ('*********************************************************');
  write ('* Opcao:'); readln (Opcao);
  Assign (ArqAlf, 'alfabeto.txt'); reset (ArqAlf);
```

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

```
 if Opcao = 'c'
then begin
     write ('Arquivo texto a ser comprimido:'); readln (NomeArqTxt);
     write ('Arquivo comprimido a ser gerado:');readln (NomeArqComp);
     Assign (ArqTxt, NomeArqTxt);Assign (ArqComprimido, NomeArqComp);
     reset (ArqTxt); Rewrite (ArqComprimido);
     Compressao (ArqTxt, ArqAlf, ArqComprimido);
     close (ArqTxt); close (ArqComprimido);
     end
else if Opcao = 'd'
     then begin
          write ('Arquivo comprimido a ser descomprimido:');
          readln (NomeArqComp);write ('Arquivo texto a ser gerado:');
          readln (NomeArqTxt); Assign (ArqTxt, NomeArqTxt);
          Assign (ArqComprimido, NomeArqComp); Rewrite (ArqTxt);
          Reset (ArqComprimido);
          Descompressao (ArqComprimido, ArqTxt, ArqAlf);
          close (ArqTxt); close (ArqComprimido);
          end
     else if Opcao = 'p'
          then begin
               write ('Arquivo comprimido para ser pesquisado:');
               readln (NomeArqComp);
               Assign (ArqComprimido, NomeArqComp);
               reset (ArqComprimido); Busca (ArqComprimido, ArqAlf);
               close (ArqComprimido);
               end;
 close (ArqAlf);
end.
```

Casamento Aproximado

Algoritmo:

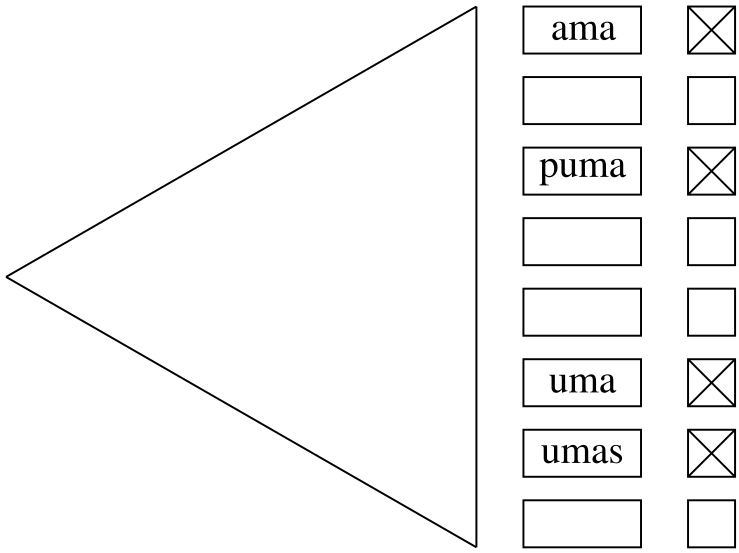
- Pesquisar o padrão no vocabulário. Neste caso, podemos ter:
  - Casamento exato, o qual pode ser uma **pesquisa binária** no vocabulário, e uma vez que a palavra tenha sido encontrada a folha correspondente na árvore de Huffman é marcada.
  - Casamento aproximado, o qual pode ser realizado por meio 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:

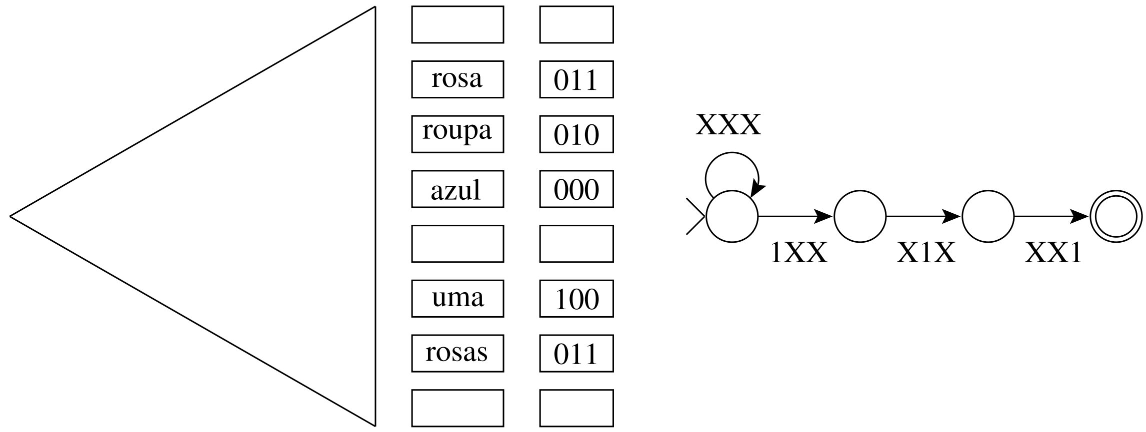
- Se uma frase tem j palavras, então uma máscara de j *bits* é colocada junto a cada palavra do vocabulário (folha da árvore de Huffman).
- Para uma palavra x da frase, o i-ésimo *bit* da máscara é feito igual a 1 se x é a i-ésima palavra da frase.
- Assim, cada palavra i da frase é pesquisada no vocabulário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