Algoritmos de Busca de Palavras em Texto

A busca de padrões dentro de um conjunto de informações tem uma grande aplicação em computação. São muitas as variações deste problema, desde procurar determinadas palavras ou sentenças em um texto até procurar um determinado objeto dentro de uma sequência de bits que representam uma imagem.
Todos eles se resumem a procurar certa sequência de bits ou bytes dentro de uma sequência maior de bits ou bytes.
Vamos considerar a versão de procurar uma sequência de bytes dentro de outra sequencia, ou ainda, procurar uma **`palavra`** dentro de um texto. **`Palavra`** deve ser entendida como uma sequência qualquer de caracteres. Assim, a formulação do problema fica:

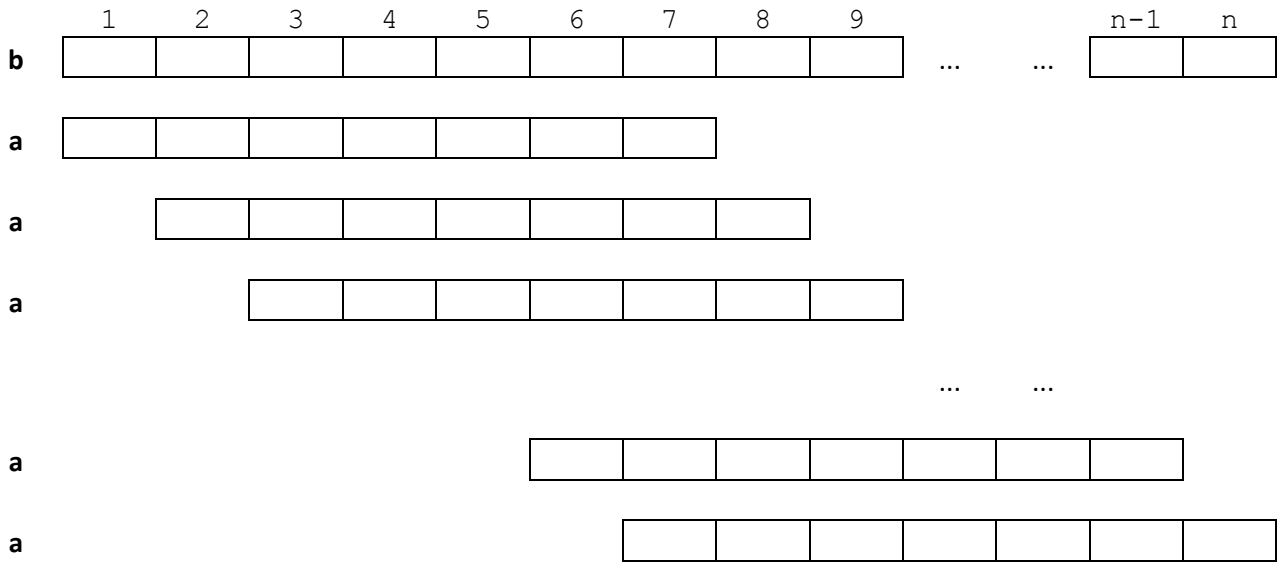
Dada uma sequência `a` de `m>0` bytes (`a[1],..., a[m]` ou `a[1..m]`) **verificar quantas vezes** ela ocorre em uma sequência `b` de `n` elementos (`b[1], ..., b[n]` ou `b[1..n]`).

O algoritmo tradicional

Nos algoritmos abaixo vamos considerar os índices começando do 1 e não do 0. Os elementos `a[0]` e `b[0]` serão ignorados.

A solução mais trivial deste problema consiste então em comparar:

```
a[1] com b[1]; a[2] com b[2]; ... a[m] com b[m]
a[1] com b[2]; a[2] com b[3]; ... a[m] com b[m+1]
...
a[1] com b[n-m+1]; a[2] com b[n-m+2]; ... a[m] com b[n]
```



Na primeira comparação em que **`a[i]`** diferente de **`b[j]`**, passa-se para o próximo passo.

Exemplos:

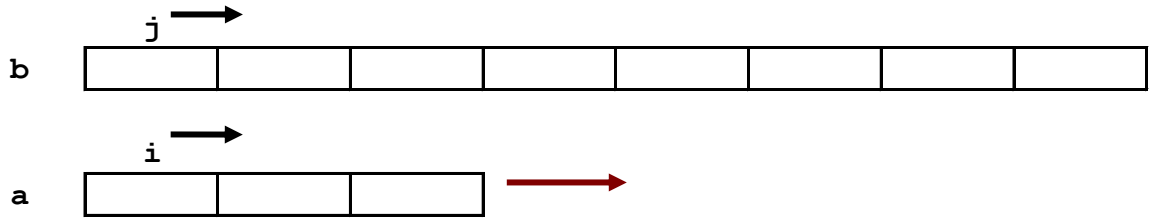
`b - O alinhamento do pensamento provoca casamento`
`a - mento` – Ocorre 3 vezes
`a - n` – Ocorre 5 vezes
`a - casa` – Ocorre 1 vez
`a - ovo` – Ocorre 1 vez
`a - prova` – Ocorre 0 vezes

`b - abababababa`
`a - bab` – Ocorre 3 vezes
`a - abab` – Ocorre 3 vezes
`a - bababa` – Ocorre 2 vezes

Abaixo esta primeira solução:

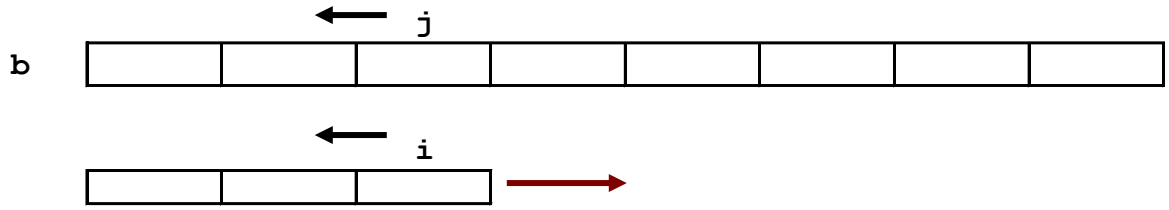
```
int casapadrao1(char a[], int m, char b[], int n) {
     int i, j, k, conta = 0;
     if (m <= 0) return 0;
     for (k = 1; k <= n - m + 1; k++) {
        for (j = k, i = 1; i <= m; j++, i++)
            if (a[i] != b[j]) break;
        if (i > m) conta++;
     }
     return conta;
}
```

Na solução acima, `i` e `j` caminham da esquerda para a direita.
Também `a` se desloca da esquerda para a direita a cada nova tentativa.



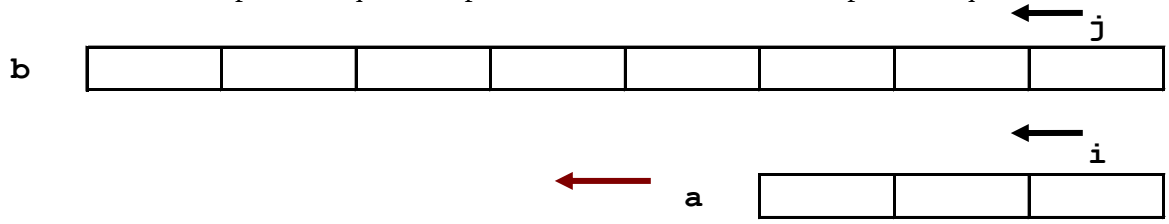
Podemos ter algumas variações, todas equivalentes:

Procurando `a` em `b` da direita para a esquerda:




```
int casapadrao2(char a[], int m, char b[], int n) {
     int i, j, k, conta = 0;
     if (m <= 0) return 0;
     for (k = m; k <= n; k++) {
        for (j = k, i = m; i > 0; j--, i--)
            if (a[i] != b[j]) break;
        if (i == 0) conta++;
     }
     return conta;
}
```

Varrendo `b` da direita para a esquerda e procurando `a` em `b` da direita para a esquerda:




```
int casapadrao3(char a[], int m, char b[], int n) {
     int i, j, k, conta=0;
     if (m <= 0) return 0;
     for (k = n; k >= m; k--) {
        for (j = k, i = m; i > 0; j--, i--)
            if (a[i] != b[j]) break;
        if (i == 0) conta++;
     }
     return conta;
}
```

Exercícios:

1) Adapte o algoritmo acima, varrendo `b` da direita para a esquerda e procurando `a` em `b` da esquerda para a direita.
2) Quantas comparações são feitas no mínimo e no máximo? Encontre sequências a e b onde o mínimo e o máximo ocorrem.
3) Por que a complexidade dos algoritmos acima é $O(n^2)$?

4) Explique porque as versões acima são todas equivalentes.
5) Adapte os algoritmos acima, devolvendo o índice inicial em b da primeira ocorrência de a ou -1 se não encontrar.

Algoritmo de Boyer-Moore – versão 1 [1977]

Esse algoritmo tenta fazer menos comparações usando uma característica do padrão a ser procurado `a`.

Quando se compara `a[1..m]` com `b[i..k]` (`k=i+m-1` para `i=1,..., n-m+1`), isto é, quando se compara `a` com um segmento qualquer dentro de `b`, a próxima comparação não precisa ser com `b[i+1..k+1]`.
Pode ser com `b[i+d..k+d]` onde `d` é calculado de forma que `b[k+1]` coincida com a última ocorrência de `b[k+1]` em `a`.
Assim, podemos deslocar a comparação com o próximo segmento em mais de um elemento. Não importa o resultado da comparação anterior.

Exemplo:

Procurar `abcd` em `abacacbabcdcdabd`

Veja como podemos fazer a busca avançando no modo proposto:

a	b	a	c	a	c	b	a	b	c	d	c	d	a	b	d
a	b	c	d												
				a	b	c	d								
							a	b	c	d					
									a	b	c	d			

Outro exemplo – Procurar `aba`:

a	b	a	c	a	c	b	a	b	c	d	c	d	a	b	d
a	b	a													
				a	b	a									
					a	b	a								
							a	b	a						
											a	b	a		
													a	b	a

O problema então consiste em saber qual a última ocorrência de `b[k+1]` em `a`.
Se soubermos todos os valores possíveis de `b[k+1]`, podemos calcular este valor para cada elemento de `a`.
Aqui está a particularidade do algoritmo: é necessário conhecer o alfabeto.
Como estamos lidando com caracteres, o alfabeto são todos os caracteres de 0 a 255.
Podemos então previamente calcular qual a última ocorrência de cada um dos caracteres de `a`.

Exemplo – Qual a última ocorrência de cada caractere na sequência abaixo e qual o deslocamento necessário?

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

Caractere	Última Ocorrência	Deslocamento
a	7	3
b	5	5
c	8	2
d	9	1
e	6	4
Todos os demais	0	10

Observe que: **`Deslocamento = Tamanho - Última Ocorrência + 1`**

Embora o objetivo seja encontrar o último `a[i]` que coincida com `b[k+1]`, **o algoritmo só depende de `a`**. No entanto, é necessário conhecer-se o alfabeto de `a`.

Veja abaixo o algoritmo.

```
int boyermoore1(unsigned char a[], int m, unsigned char b[], int n) {
     int ult[256];
     int i, j, k, conta=0;
     if (m <= 0) return 0;

     /* verifica última ocorrência de cada letra em a */
     for (i = 0; i < 256; i++) ult[i] = 0;
     for (i = 1; i <= m; i++) ult[a[i]] = i;

     /* procura a em b da direita para a esquerda */
     for (k = m; k <= n; k = k + m - ult[b[k+1]] + 1) {
        for (j = k, i = m; i > 0; j--, i--)
            if (a[i] != b[j]) break;
        if (i == 0) conta++;
        /* pode ser que já tenha chegado ao fim de b e neste caso */
        /* não dá para usar [b[k+1]] como índice de ult[] */
        if (k + 1 > n) break;
     }
     return conta;
}
```

Este algoritmo é O(n.m).
A fase de pré-processamento é O(m+K), onde K depende do alfabeto.

A fase de busca é O(n.m).

Entretanto, no caso geral se comporta melhor que o algoritmo tradicional.

Exercícios:

1) Porque a declaração de a e b tem que ser `unsigned char` e não `char` simplesmente?
2) Usando a mesma ideia do algoritmo acima, é possível avançar a comparação mais do que `b[k+1]`? E menos?
3) Adapte o algoritmo acima para fazer a busca de `a` em `b` da direita para a esquerda, isto é, comparando com `b[n-m+1..n], b[n-m-2..n-1], ..., b[1..m]`.
4) No algoritmo acima é necessário conhecer o alfabeto, ou os valores possíveis de `b[k+1]`. Se `a` e `b` fossem do tipo `int`, como ficaria o algoritmo?
5) Quando `b[k+1]` não coincide com nenhum de `a[1..m]` já vimo que o deslocamento será de `m+1`. Uma pequena variação é procurar primeiro `b[p] (p>k+1)` tal que `b[p]=a[1]`. Ou seja, vamos aumentar o deslocamento.

Algoritmo de Boyer-Moore – versão 2

A versão 2 do algoritmo é intuitiva, mas tem uma implementação mais engenhosa.
Não é necessário conhecer-se o alfabeto de `a`.
Também só depende de `a`.

Neste algoritmo é necessário que a comparação de `a` com `b`, seja feita da direita para a esquerda:

> Para `i=m, m+1, ..., n`
> Comparar `a[m]` com `b[i]`; `a[m-1]` com `b[i-1]`; ...; `a[1]` com `b[i-m+1]`

A ideia básica é a seguinte:

Suponha que numa das comparações já descobrimos que **`a[h..m]`** é igual a **`b[k-m+h..k]`**, ou seja, descobrimos que existe um trecho (parcial ou total) no meio de **`b`** que é igual a um trecho corresponde dente de **`a`**. Só haverá casamento se **`a`** tiver um trecho igual em **`a[1..m-1]`**. Se não houver tal trecho em `a`, podemos deslocar de `m` elementos.

Exemplos:

```
a: B A B
b: A B A B C B A B C
```

A	B	A	B	C	B	A	B	C
B	A	B						
	B	A	B					

			B	A	B			
					B	A	B	

a: A B A B A
b: B C A B A B C C A B A D D A B A B A

B	C	A	B	A	B	C	C	A	B	A	D	D	A	B	A	B	A	B	A	B	A	B
A	B	A	B	A																		
		A	B	A	B	A																
			A	B	A	B	A															
				A	B	A	B	A														
						A	B	A	B	A												
								A	B	A	B	A										
									A	B	A	B	A									
											A	B	A	B	A							
													A	B	A	B	A					
															A	B	A	B	A			
																	A	B	A	B	A	

Portanto é necessário localizar a última ocorrência de `a[h..m]` em `a[1..m-1]`.
Vamos chamar essa ocorrência de `alcance[h]` e vamos defini-la como o último índice onde houve a coincidência.

Assim, se `a[h..m] = a[p..q] (h,p >=1 e q<=m-1)` e `a[p..q]` é a última ocorrência de `a[h..m]` em `a[1..m-1]`, então `alcance[h] = q`.

Um caso particular ocorre quando o início de `a` coincide com o final. Neste caso temos que considerar como se houvesse o casamento no restante da cadeia. Veja exemplos:

h	1	2	3	4	5
a	C	B	A	B	A
alcance	0	0	0	3	3

h	1	2	3	4	5
a	A	B	A	B	A
alcance	3	3	3	3	3

h	1	2	3	4	5	6	7	8	9
a	A	B	C	A	B	B	C	A	B
alcance	2	2	2	2	2	5	5	5	6

Veja abaixo outros exemplos de deslocamento:

b	x	x	x	x	x	x	x	x	a	x	x	x	x	x	x
a					a	b	a	b	a						
deslocamento							a	b	a	b	a				

b	x	x	x	x	x	x	x	b	a	x	x	x	x	x	x
a					a	b	a	b	a						
deslocamento							a	b	a	b	a				

b	x	x	x	x	x	x	a	b	a	x	x	x	x	x	x
a					a	b	a	b	a						
deslocamento							a	b	a	b	a				

b	x	x	x	x	x	b	a	b	a	x	x	x	x	x	x
a					a	b	a	b	a						
deslocamento							a	b	a	b	a				

b	x	x	x	x	x	x	a	x	x	x	x	x	x	x	x
a					a	b	a								
deslocamento							a	b	a						

b	x	x	x	x	x	x	c	x	x	x	x	x	x	x	x
a					a	b	c								
deslocamento								a	b	c					

b	x	x	x	x	x	x	x	a	x	x	x	x	x	x	x
a					a	b	b	a							
deslocamento								a	b	b	a				

b	x	x	x	x	x	x	b	a	x	x	x	x	x	x	x
a					a	b	b	a							
deslocamento								a	b	b	a				

b	x	x	x	x	x	x	x	a	x	x	x	x	x	x	x
a					a	a	a	a							
deslocamento						a	a	a	a						

b	x	x	x	x	x	x	a	a	x	x	x	x	x	x	x
a					a	a	a	a							
deslocamento						a	a	a	a						

b	x	x	x	x	x	a	a	a	x	x	x	x	x	x	x
a					a	a	a	a							
deslocamento						a	a	a	a						

b	x	x	x	x	a	a	a	a	x	x	x	x	x	x	x
a					a	a	a	a							
deslocamento						a	a	a	a						

Da mesma forma que o algoritmo anterior temos que fazer um pré-processamento em `a` para determinar o `alcance[h]`. Determinado `alcance[h]`, ele será usado como deslocamento para a próxima tentativa de fazer-se o casamento.

A determinação de `h` é seguinte:

```
for (h = m; h >= 1; h--) {
    mm = m-1;
    ii = mm; i = m;
    while (ii >= 1 && i >= h)
        if (a[ii] == a[i]) {
           --ii; --i; /* continua comparando */
        }
        else {
          --mm; /* reduz o candidato a alcance[h] */
          ii = mm; i = m; /* reinicia a comparação */
        }
    alcance[h] = mm;
}
```

O algoritmo completo com uma versão mais otimizada da determinação de `alcance[h]` está abaixo:

```
#define MAXm 100 /* maior valor de m */

int boyermoore2(unsigned char a[], int m, unsigned char b[], int n) {
     int alcance[MAXm];
     int i, j, k, h, mm, ii, conta=0, d=0;
     if (m <= 0) return 0;
```

```
      /* pré-processamento de a - versão mais otimizada */
      h = mm = m;
      do {
            ii = --mm;
            i = m;
            while (ii >= 1 && a[ii] == a[i]){
                  ii--; i--;
            }
            while (h > i) alcance[h--] = mm;
      } while (ii >= 1);
      while (h >= 1) alcance[h--] = mm;

      /* procura a em b */
      k = m;
      while (k <= n) {
            for (i = m, j = k; i >= 1; i--, j--)
                  if (a[i] != b[j]) break;
            if (i < 1) ++conta;
            if (i == m) k++; /* desloca apenas 1 */
                  /* o último que coincidiu foi a[i] */
            else k = k + m - alcance[i+1];
      }
     return conta;
}
```

Este algoritmo também é O(n.m).
Neste caso, tanto a fase de pré-processamento quanto a fase de busca são O(n.m).

Da mesma forma que a versão 1, no caso geral se comporta melhor que o algoritmo tradicional.
Esse algoritmo se comporta melhor quando há muita repetição de trechos em `a`, o que pode ocorrer com maior frequência se `m` é grande.
Entretanto quando não há coincidência o deslocamento é de apenas 1 enquanto que na versão 1 o deslocamento é em geral maior.

Versão 1 versus Versão 2

Finalmente, reforçamos que na versão 1 é necessário conhecer o alfabeto enquanto que na versão 2 não é necessário.
Ambas as versões só dependem de `a`.

Exercício – versão híbrida

Quando o alfabeto é conhecido, é possível usar as duas versões ao mesmo tempo.
A cada repetição do algoritmo, calcula-se o deslocamento referente a cada versão e usa-se o maior deles. Fica como exercício.

Outra versão (uma pequena variação)

Uma pequena variação das versões 1 e 2. Deslocando-se para o próximo caractere de **a** diferente daquele que não houve coincidência. Exemplo:

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

Para isso é necessário construir-se uma tabela `ult_dif[i] (i=1,2,..,m)` tal que `ult_dif[i]=k` onde `k` é o maior índice menor que `i` e `a[k]` é diferente de `a[i]`. Exemplo:

```
1 2 3 4 5 6 7
b b c a a b b

i    ult_dif
7    5
6    5
5    3
4    3
3    2
2    2
1    1
```

Há 2 casos particulares:

1) **Quando não há diferentes à esquerda. Neste caso o deslocamento deve ser igual ao índice do elemento.**
2) **Quando coincide totalmente. Neste caso o deslocamento deve ser 1**

Outros algoritmos

Existem outros algoritmos de complexidade mais baixa que os anteriores.

O mais conhecido é o algoritmo de Knuth-Morris-Pratt [**KNUTH D.E.**, **MORRIS** (Jr) **J.H.**, **PRATT V.R.**, 1977, Fast pattern matching in strings, *SIAM Journal on Computing* 6(1):323-350].

Sua complexidade é O(m) na fase de pré-processamento e de O(m+n) na fase de busca.
Portanto um algoritmo linear.

Outras referências para este assunto:

No site http://www-igm.univ-mlv.fr/~lecroq/string/index.html há informações sobre vários algoritmos de busca de palavras em texto e também uma simulação do seu funcionamento.