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.

Usando a terminologia do Python, estamos considerando então procurar uma sub-string dentro de uma string de caracteres. Já existe uma função intrínseca com essa função em Python (find). Essa função devolve o índice da primeira ocorrência de uma sub-string dentro de outra. Exemplos:

```
str1 = "sentimento de aumento do cimento"
str2 = "men"

print(str1.find(str2))              # imprime 5
print(str1.find(str2, 25))          # imprime 27
print(str1.find(str2, 15, 25))      # imprime 16
```

Estamos interessados em saber quantas vezes uma sub-string ocorre dentro de uma string. Tal função também já existe em Python (count). Exemplos:

```
print(str1.count(str2))             # imprime 3
print(str1.count(str2, 25))         # imprime 1
print(str1.count(str2, 15, 25))     # imprime 1
print(str1.count("e"))              # imprime 5
```

O nosso objetivo é saber como construir esse algoritmo. A formulação do problema fica então:

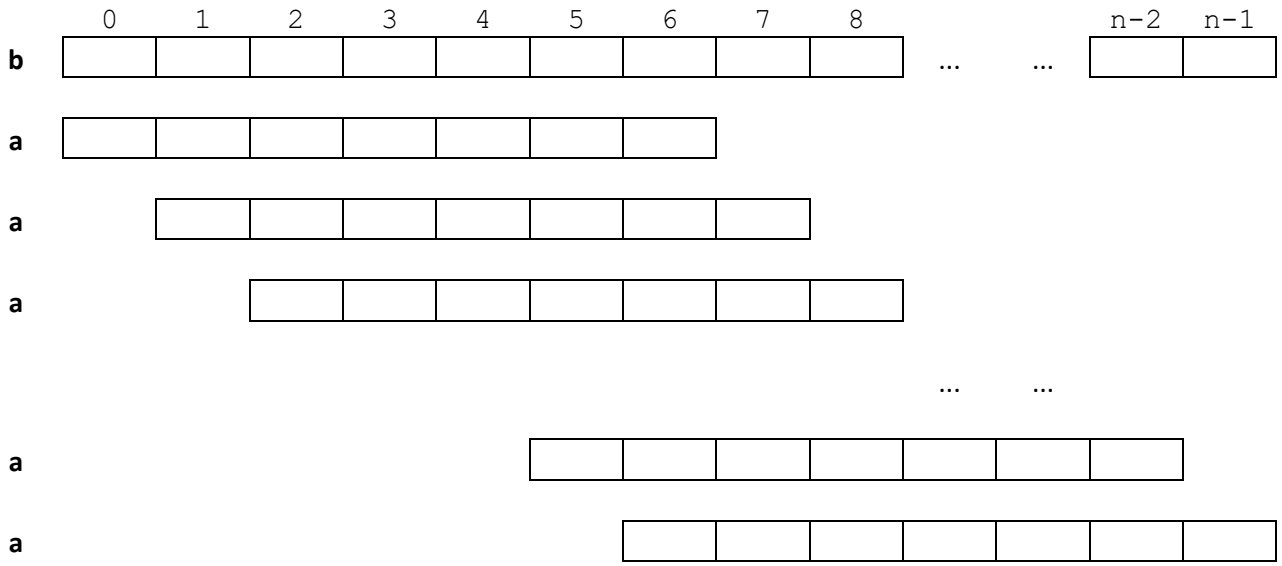
Dada uma sequência `a` de `m > 0` bytes (`a[0],..., a[m - 1]` ou `a[0:m]`) **verificar quantas vezes** ela ocorre em uma sequência `b` de `n` elementos (`b[0], ..., b[n - 1]` ou `b[0:n]`).

Usando a notação do Python – Verificar **quantas vezes** a string **`a[0:m]`** ocorre em **`b[0:n]`**.

O algoritmo tradicional

A solução mais trivial deste problema consiste então em comparar:

```
a[0] com b[0]; a[1] com b[1]; ... a[m - 1] com b[m - 1]
a[0] com b[1]; a[1] com b[2]; ... a[m - 1] com b[m]
...
a[0] com b[n-m]; a[1] com b[n-m+1]; ... a[m-1] com b[n-1]
```



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

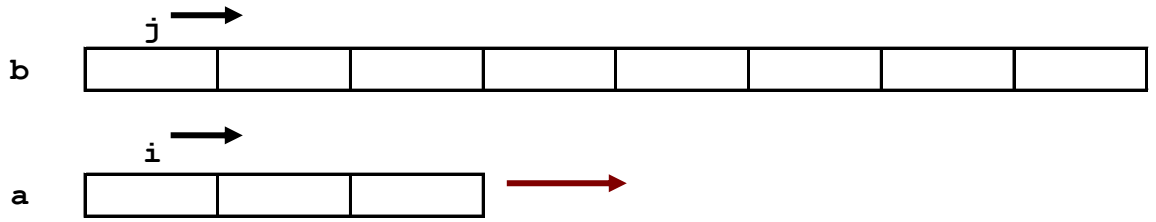
b - abababababa
a - bab – Ocorre 3 vezes
a - abab – Ocorre 3 vezes
a - bababa – Ocorre 2 vezes

Abaixo esta primeira solução:

```
def comparapadrao(a, b):
    m, n = len(a), len(b)
    conta = 0
    for k in range(n - m + 1):
        i, j = 0, k
        while i < m:
            print(i, j)
            if a[i] != b[j]: break
```

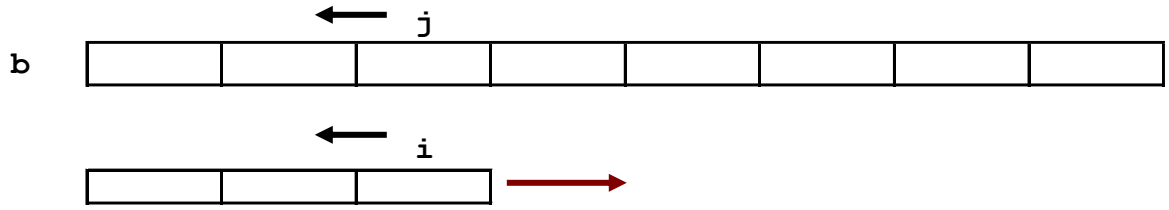
```
            i, j = i + 1, j + 1
        if i == m: conta += 1
    return conta
```

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.



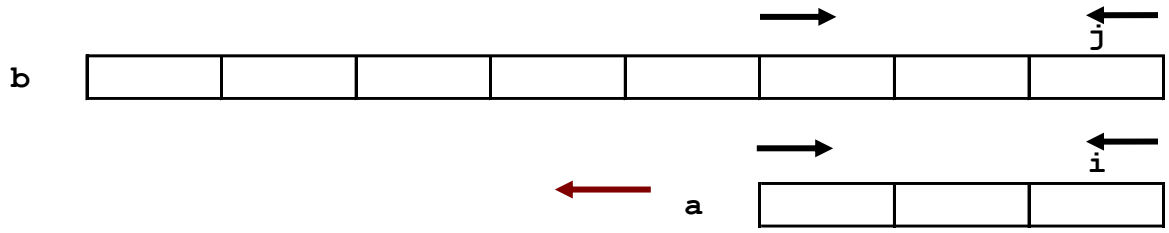
Podemos ter algumas variações, todas equivalentes:

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




```
def comparapadrao1(a, b):
    m, n = len(a), len(b)
    conta = 0
    for k in range(m - 1, n):
        i, j = m - 1, k
        while i >= 0:
            print(i, j)
            if a[i] != b[j]: break
            i, j = i - 1, j - 1
        if i < 0: conta += 1
    return conta
```

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



Exercícios:

1) Adapte o algoritmo acima, varrendo `b` da direita para a esquerda e procurando `a` em `b` da esquerda para a direita.
2) Idem, varrendo `b` da direita para a esquerda e procurando `a` em `b` da direita para a esquerda.
3) Quantas comparações são feitas no mínimo e no máximo? Encontre sequências a e b onde o mínimo e o máximo ocorrem.
4) Por que a complexidade dos algoritmos acima é $O(n^2)$?
5) Explique porque as versões acima são todas equivalentes.
6) 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[0:m]` com `b[i:k]  (k=i+m)`, 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]` , o próximo caractere de b a ser comparado, coincida com a última ocorrência de `b[k]` 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 do próximo elemento de `b` em `a`.
Se sabemos todos os valores possíveis de `b[k]`, 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?

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

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

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

Embora o objetivo seja encontrar o último `a[i]` que coincida com o próximo `b[k]`, **o algoritmo só depende de `a`**.

Veja abaixo o algoritmo.

```
def BoyerMoore1(a, b):
    m, n = len(a), len(b)
    conta = 0
    # tabela de últimas ocorrências de cada caractere em a
    ult = [-1] * 256
    # varrer a e definir as últimas ocorrências de cada caractere
    for k in range(m): ult[ord(a[k])] = k
    # procura a em b - da esquerda para a direita
    k = m - 1
```

```
    while k < n:
        j, i = k, m - 1
        while i >= 0:
            if a[i] != b[j]: break
            j, i = j - 1, i - 1
        # comparação chegou ao fim
        if i < 0: conta += 1
        # caso particular - se k é n-1 (último de b)
        # então k+1 é índice inválido
        # o if abaixo evita esse caso
        if k + 1 >= n: break
        # desloca baseado no valor de b[k+1]
        k = k + m - ult[ord(b[k+1])]
    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) Na versão acima, comparamos a da direita para a esquerda (do fim para o começo). Claro que é equivalente a comparar da esquerda para a direita (do início para o fim). Faça essa modificação no algoritmo.
2) Usando a mesma ideia do algoritmo acima, é possível avançar a comparação mais ainda à frente? O que teria que ser feito?
3) No algoritmo acima é necessário conhecer o alfabeto, ou os valores possíveis do próximo caractere de `b` a ser comparado. Se em vez de caracteres de 1 byte, fossem considerados caracteres de 2 bytes como ficaria o algoritmo?
4) Quando não há coincidência do próximo caractere de **`b`** com nenhum dos caracteres de **`a`**, já vimos que os deslocamento será de **`m+1`**posições. Uma pequena variação seria procurar o primeiro **`b[p]`** tal que **`b[p] == a[0]`**. Ou seja, vamos aumentar o deslocamento. Adicione essa modificação no algoritmo.

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 <u>da direita para a esquerda</u>:

Para `i = m-1, m, ..., n-1`

Comparar `a[m-1]` com `b[i]`; `a[m-2]` com `b[i-1]`; ...; `a[0]` com `b[i-m+1]`

A ideia básica é a seguinte:

Suponha que numa das comparações já descobrimos que existe um trecho (parcial ou total) no meio de **`b`** que é igual a um trecho corresponde dentro de **`a`**. Só haverá casamento se **`a`** tiver outro trecho interno igual a este onde houve a coincidência. Se não houver tal trecho em **`a`**, podemos deslocar de **`m`** posições.

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 A B A B
```

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-1]`** em **`a[0..m-2]`**.
Vamos chamar essa ocorrência de **`deslocamento[h]`** e vamos defini-la como o último índice onde houve a coincidência.

Assim, se **`a[h..m-1] = a[p:q-1]`** é a última ocorrência de **`a[h..m-1]`** em **`a[0..m-2]`**, então temos que deslocar **`a`** de modo que **`a[h]`** coincida com **`a[p]`**.

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	0	1	2	3	4
a	C	B	A	B	A
deslocamento	5	5	5	2	2

h	0	1	2	3	4
a	A	B	A	B	A
deslocamento	2	2	2	2	2

h	0	1	2	3	4	5	6	7	8
a	A	B	C	A	B	B	C	A	B
deslocamento	7	7	7	7	7	4	4	4	3

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 deslocamento (`desloc[h]`). Determinado esse deslocamento, ele será usado como para a próxima tentativa de fazer-se o casamento.

A função abaixo é uma forma de calcular tal deslocamento. Devolve uma lista onde:
desloc[0] = 0 # não é usado
desloc[1] = deslocamento quando houve coincidência de 1 caractere
desloc[2] = deslocamento quando houve coincidência com 2 caracteres
...
desloc[m-1] = deslocamento quando houve coincidência com m-1 caracteres

```
def SubCadeias(a):
    m = len(a)
    # desloc[i] (1 <= i, <= m)
    # deslocamento quando coincide com sub-cadeia de tamanho i
    desloc = [0] + (m - 1) * [m]
    # testa todos os tamanhos de 1 até m - 1
    for k in range(1, m):
        # ii percorre a partir do final
        # jj percorre a a partir da posição corrente
        ii = m - 1
        jj = m - 2
        ultimojj = jj
        tam = 0
        achou = False
        while not achou:
            # se não coincide, continua a procura
            # no próximo candidato
            if a[ii] != a[jj]:
                ii = m - 1
                jj = ultimojj = ultimojj - 1
                if jj < 0: break
            else:
                tam += 1
                # verifica se chegou ao tamanho procurado
                if tam == k or jj == 0:
                    achou = True
                    desloc[k] = m - 1 - ultimojj
                else:
                    # continua procurando
                    ii -= 1
                    jj -= 1
    return desloc
```

A função acima é uma forma de determinar o deslocamento. Existem outras formas.

Abaixo o algoritmo completo:

```
def BoyerMoore2(a, b):
    # Calcula os deslocamentos para a
    dsl = SubCadeias(a)
    # Procura a em b
    conta = 0
    m, n = len(a), len(b)
    k = m - 1
    while k < n:
        i = m - 1
```

```
        j = k
        while i >= 0:
            if a[i] != b[j]:
                # Se nada coincidiu, desloca apenas 1
                if i == m - 1: k += 1
                else:
                    # Se 1 ou mais coincidiram, desloca conforme dsl
                    # Coincidiu até o i anterior (i + 1)
                    # Desloca conforme dsl
                    k = k + dsl[i + 1]
                # Sai do while interno para comparar com novo trecho
                break
            # continua a comparação
            i -= 1
            j -= 1
        # Sai do while interno por break ou por i < 0
        if i < 0:
            conta += 1
            k += 1
    # Fim de todas as comparações possíveis
    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=0,1,2,..,m-1)` tal que `ult_dif[i]=k` onde `k` é o maior índice menor que `i` e `a[k]` é diferente de `a[i]`. Exemplo:

```
0 1 2 3 4 5 6
b b c a a b b

i    ult_dif desloc
6      4       2
5      4       1
4      2       2
3      2       1
2      1       1
1      2       2
0      1       1
```

Há 2 casos particulares:

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