AULA 24

Busca de Substrings

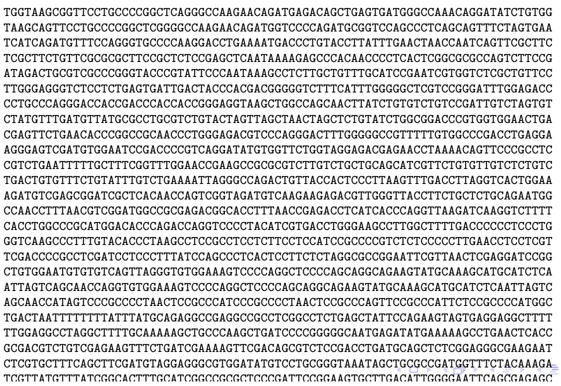
Substring search

Referências: Busca de substring (PF), Substring Searching (SW), slides (SW), vídeo (SW).

Introdução

Problema: Dada uma string `pat` e uma string `txt`, encontrar uma ocorrência de `pat` em `txt`.

Exemplo: encontre `ATTGG` em:



Introdução

Dizemos que um vetor $\texttt{pat}[0\,.\,.\,\texttt{m}-1]$ **casa com** $\texttt{txt}[0\,.\,.\,\texttt{n}-1]$ **a partir de** $\texttt{i}$ se

$$\texttt{pat}[0\,.\,.\,\texttt{m}-1] = \texttt{txt}[\texttt{i}\,.\,.\,\texttt{i}+\texttt{m}-1]$$

para algum $\texttt{i}$ em $[0\,.\,.\,\texttt{n}-\texttt{m}]$.

Exemplo:

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

	0	1	2	3
pat	b	c	b	a

$\texttt{pat}[0\,.\,.\,3]$ casa com $\texttt{txt}[0\,.\,.\,9]$ a partir de 5.

Busca de substrings

Problema alternativo: Dados $\texttt{pat}[0\,.\,.\,\texttt{m}-1]$ e $\texttt{txt}[0\,.\,.\,\texttt{n}-1]$, encontrar o número de ocorrências de `pat` em `txt`.

Exemplo: Para $\texttt{n} = 10$, $\texttt{m} = 4$, e

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

	0	1	2	3
pat	b	a	b	a

`pat` ocorre 2 vezes em `txt`.

Algoritmo de força bruta

pat = a b a b b a b a b b a

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	
	a	b	a	a	b	a	b	a	b	b	a	b	a	b	a	b	b	a	b	a	b	b	a	txt
0	a	b	a	b	b	a	b	a	b	b	a													

Algoritmo de força bruta

pat = a b a b b a b a b b a

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	
	a	b	a	a	b	a	b	a	b	b	a	b	a	b	a	b	b	a	b	a	b	b	a	txt
0	a	b	a	b	b	a	b	a	b	b	a													
1		a	b	a	b	b	a	b	a	b	b	a												

Algoritmo de força bruta

pat = a b a b b a b a b b a

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	
	a	b	a	a	b	a	b	a	b	b	a	b	a	b	a	b	b	a	b	a	b	b	a	txt
0	a	b	a	b	b	a	b	a	b	b	a													
1		a	b	a	b	b	a	b	a	b	b	a												
2			a	b	a	b	b	a	b	a	b	b	a											

Algoritmo de força bruta

pat = a b a b b a b a b b a

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	
	a	b	a	a	b	a	b	a	b	b	a	b	a	b	a	b	b	a	b	a	b	b	a	txt
0	a	b	a	b	b	a	b	a	b	b	a													
1		a	b	a	b	b	a	b	a	b	b	a												
2			a	b	a	b	b	a	b	a	b	b	a											
3				a	b	a	b	b	a	b	a	b	b	a										

Algoritmo de força bruta

pat = a b a b b a b a b b a

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	
	a	b	a	a	b	a	b	a	b	b	a	b	a	b	a	b	b	a	b	a	b	b	a	txt
0	a	b	a	b	b	a	b	a	b	b	a													
1		a	b	a	b	b	a	b	a	b	b	a												
2			a	b	a	b	b	a	b	a	b	b	a											
3				a	b	a	b	b	a	b	a	b	b	a										
4					a	b	a	b	b	a	b	a	b	b	a									
5						a	b	a	b	b	a	b	a	b	b	a								
6							a	b	a	b	b	a	b	a	b	b	a							
7								a	b	a	b	b	a	b	a	b	b	a						
8									a	b	a	b	b	a	b	a	b	b	a					
9										a	b	a	b	b	a	b	a	b	b	a				
10											a	b	a	b	b	a	b	a	b	b	a			
11												a	b	a	b	b	a	b	a	b	b	a		
12													a	b	a	b	b	a	b	a	b	b	a	

Algoritmo de força bruta

i	j	i+j	0	1	2	3	4	5	6	7	8	9	10
		txt →	A	B	A	C	A	D	A	B	R	A	C
0	2	2	A	B	R	A ← pat							
1	0	1		A	B	R	A						
2	1	3			A	B	R	A					
3	0	3				A	B	R	A				
4	1	5					A	B	R	A			
5	0	5						A	B	R	A		
6	4	10							A	B	R	A	

entries in red are mismatches

entries in gray are for reference only

entries in black match the text

return i *when* j *is* M

match

Brute-force substring search

Algoritmo de força bruta

Devolve a primeira de ocorrências de pat em txt.

```
public static
int search(String pat, String txt) {
   int i, n = txt.length();
   int j, m = pat.length();
   for (i = 0; i <= n-m; i++) {
      for (j = 0; j < m; j++)
         if(txt.charAt(i+j)!=pat.charAt(j))
            break;
      if (j == m) return i;
   }
   return n;
}
```

Algoritmo de força bruta

Relação invariante: no início de cada iteração do "`for (j= 0; ...)`" vale que

(i0) $\texttt{pat}[0\,..\,\texttt{j}-1] = \texttt{txt}[\texttt{i}\,..\,\texttt{i}+\texttt{j}-1]$

Consumo de tempo

Consumo de tempo da função `search()`.

linha	todas as execuções da linha
1-2	$= 1$
3	$= \texttt{n} - \texttt{m} + 1$
4	$\leq (\texttt{n} - \texttt{m} + 1)(\texttt{m} + 1)$
5	$\leq (\texttt{n} - \texttt{m} + 1)\,\texttt{m}$
6	$\leq (\texttt{n} - \texttt{m} + 1)$
7	$= \texttt{n} - \texttt{m}$
8-9	$= 1$
total	$< 3\,(\texttt{n} - \texttt{m} + 2) + 2\,(\texttt{n} - \texttt{m} + 1)(\texttt{m} + 1)$ $= \mathrm{O}((\texttt{n} - \texttt{m} + 1)\texttt{m})$

Pior caso

pat = a a a a a a a a a a b

```
    0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22
    a a a a a a a a a a a  a  a  a  a  a  a  a  a  a  a  a  a  txt
0   a a a a a a a a a a b
1     a a a a a a a a a a  b
2       a a a a a a a a a  a  b
3         a a a a a a a a  a  a  b
4           a a a a a a a  a  a  a  b
5             a a a a a a  a  a  a  a  b
6               a a a a a  a  a  a  a  a  b
7                 a a a a  a  a  a  a  a  a  b
8                   a a a  a  a  a  a  a  a  a  b
9                     a a  a  a  a  a  a  a  a  a  b
10                      a  a  a  a  a  a  a  a  a  a  b
11                         a  a  a  a  a  a  a  a  a  a  b
12                            a  a  a  a  a  a  a  a  a  a  b
```

Pior caso

i	j	i+j	0	1	2	3	4	5	6	7	8	9
		txt →	A	A	A	A	A	A	A	A	A	B
0	4	4	A	A	A	A	B ← pat					
1	4	5		A	A	A	A	B				
2	4	6			A	A	A	A	B			
3	4	7				A	A	A	A	B		
4	4	8					A	A	A	A	B	
5	5	10						A	A	A	A	B

Brute-force substring search (worst case)

Melhor caso

pat = b a a a a a a a a a a

```
    0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22
    a a a a a a a a a a a  a  a  a  a  a  a  a  a  a  a  a  a  txt
0   b a a a a a a a a a a
1     b a a a a a a a a a  a
2       b a a a a a a a a  a  a
3         b a a a a a a a  a  a  a
4           b a a a a a a  a  a  a  a
5             b a a a a a  a  a  a  a  a
6               b a a a a  a  a  a  a  a  a
7                 b a a a  a  a  a  a  a  a  a
8                   b a a  a  a  a  a  a  a  a  a
9                     b a  a  a  a  a  a  a  a  a  a
10                      b  a  a  a  a  a  a  a  a  a  a
11                         b  a  a  a  a  a  a  a  a  a  a
12                            b  a  a  a  a  a  a  a  a  a  a
```

Conclusões

O consumo de tempo de `search()` no pior caso é $\mathrm{O}((\texttt{n} - \texttt{m} + 1)\texttt{m})$.

O consumo de tempo de `search()` no melhor caso é $\mathrm{O}(\texttt{n} - \texttt{m} + 1)$.

Isto significa que no pior caso o consumo de tempo é essencialmente proporcional a $\texttt{m}\,\texttt{n}$.

Em geral o algoritmo é rápido e faz não mais que $1.1 \times \texttt{n}$ comparações.

Algoritmo de força bruta: versão alternativa

```java
public static
int search(String pat, String txt) {
  int i, n = txt.length();
  int j, m = pat.length();
  for (i=0, j=0; i < n && j < m; i++) {
     if(txt.charAt(i)==pat.charAt(j))j++;
     else {
        i -= j; // retrocesso
        j = 0;
     }
  }
  if (j == m) return i - m;
  return n; }
```

Algoritmo força bruta: direita para esquerda

pat = a b a b b a b a b b a

```
    0  1  2  3  4  5  6  7  8  9  10 11 12 13 14 15 16 17 18 19 20 21 22
    a  b  a  a  b  a  b  a  b  b  a  b  a  b  a  b  b  a  b  a  b  b  a  txt
0   a  b  a  b  b  a  b  a  b  b  a
```

Algoritmo força bruta: direita para esquerda

pat = a b a b b a b a b b a

```
    0  1  2  3  4  5  6  7  8  9  10 11 12 13 14 15 16 17 18 19 20 21 22
    a  b  a  a  b  a  b  a  b  b  a  b  a  b  a  b  b  a  b  a  b  b  a  txt
0   a  b  a  b  b  a  b  a  b  b  a
1      a  b  a  b  b  a  b  a  b  b  a
```

Algoritmo força bruta: direita para esquerda

pat = a b a b b a b a b b a

```
    0  1  2  3  4  5  6  7  8  9  10 11 12 13 14 15 16 17 18 19 20 21 22
    a  b  a  a  b  a  b  a  b  b  a  b  a  b  a  b  b  a  b  a  b  b  a  txt
0   a  b  a  b  b  a  b  a  b  b  a
1      a  b  a  b  b  a  b  a  b  b  a
2         a  b  a  b  b  a  b  a  b  b  a
```

Algoritmo força bruta: direita para esquerda

pat = a b a b b a b a b b a

```
    0  1  2  3  4  5  6  7  8  9  10 11 12 13 14 15 16 17 18 19 20 21 22
    a  b  a  a  b  a  b  a  b  b  a  b  a  b  a  b  b  a  b  a  b  b  a  txt
0   a  b  a  b  b  a  b  a  b  b  a
1      a  b  a  b  b  a  b  a  b  b  a
2         a  b  a  b  b  a  b  a  b  b  a
3            a  b  a  b  b  a  b  a  b  b  a
```

Algoritmo força bruta: direita para esquerda

pat = a b a b b a b a b b a

```
    0  1  2  3  4  5  6  7  8  9  10 11 12 13 14 15 16 17 18 19 20 21 22
    a  b  a  a  b  a  b  a  b  b  a  b  a  b  a  b  b  a  b  a  b  b  a  txt
0   a  b  a  b  b  a  b  a  b  b  a
1      a  b  a  b  b  a  b  a  b  b  a
2         a  b  a  b  b  a  b  a  b  b  a
3            a  b  a  b  b  a  b  a  b  b  a
4               a  b  a  b  b  a  b  a  b  b  a
```

Algoritmo força bruta: direita para esquerda

pat = a b a b b a b a b b a

```
      0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22
      a b a a b a b a b b a  b  a  b  a  b  b  a  b  a  b  b  a   txt
0     a b a b b a b a b b a
1       a b a b b a b a b b  a
2         a b a b b a b a b  b  a
3           a b a b b a b a  b  b  a
4             a b a b b a b  a  b  b  a
5               a b a b b a  b  a  b  b  a
```

Algoritmo força bruta: direita para esquerda

pat = a b a b b a b a b b a

```
      0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22
      a b a a b a b a b b a  b  a  b  a  b  b  a  b  a  b  b  a   txt
0     a b a b b a b a b b a
1       a b a b b a b a b b  a
2         a b a b b a b a b  b  a
3           a b a b b a b a  b  b  a
4             a b a b b a b  a  b  b  a
5               a b a b b a  b  a  b  b  a
6                 a b a b b  a  b  a  b  b  a
```

Algoritmo força bruta: direita para esquerda

pat = a b a b b a b a b b a

```
      0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22
      a b a a b a b a b b a  b  a  b  a  b  b  a  b  a  b  b  a   txt
0     a b a b b a b a b b a
1       a b a b b a b a b b  a
2         a b a b b a b a b  b  a
3           a b a b b a b a  b  b  a
4             a b a b b a b  a  b  b  a
5               a b a b b a  b  a  b  b  a
6                 a b a b b  a  b  a  b  b  a
7                   a b a b  b  a  b  a  b  b  a
8                     a b a  b  b  a  b  a  b  b  a
9                       a b  a  b  b  a  b  a  b  b  a
10                        a  b  a  b  b  a  b  a  b  b  a
11                           a  b  a  b  b  a  b  a  b  b  a
12                              a  b  a  b  b  a  b  a  b  b  a
```

Força bruta: direita para esquerda

Devolve a primeira de ocorrências de pat em txt.

```
public static
int search(String pat, String txt) {
  int i, n = txt.length();
  int j, m = pat.length();
  for (i = 0; i <= n-m; i+=1 /*skip*/) {
     for (j = m-1; j >= 0; j--)
        if(txt.charAt(i+j)!=pat.charAt(j))
           break;
     if (j == -1) return i;
  }
  return n;
}
```

Próximos passos

Existe algoritmo mais rápido que o força bruta?

Existe algoritmo que faz apenas n comparações entre caracteres?

Existe algoritmo que faz menos que n comparações?

Algoritmo KMP para busca de substring

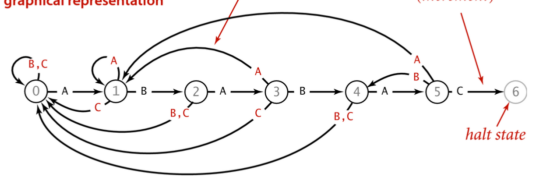
internal representation

	j	0	1	2	3	4	5
	pat.charAt(j)	A	B	A	B	A	C
dfa[][j]	A	1	1	3	1	5	1
	B	0	2	0	4	0	4
	C	0	0	0	0	0	6

mismatch transition (back up)

match transition (increment)

graphical representation



DFA corresponding to the string A B A B A C

Algoritmo de força bruta

$P =$ a b a b b a b a b b a

```
    0  1  2  3  4  5  6  7  8  9 10 11 12 13 14 15 16 17 18 19 20 21 22
    a  b  a  a  b  a  b  a  b  b  a  b  a  b  a  b  b  a  b  a  b  b  a  T
 0  a  b  a  b
 1     a
 2        a  b
 3           a  b  a  b  b
 4              a
 5                 a  b  a  b  b  a  b  a  b  b
 6                    a
 7                       a  b  a
 8                          a
 9                             a
10                                a  b  a  b
11                                   a
12                                      a  b  a  b  b  a  b  a  b  b  a
```

Algoritmo de força bruta: versão alternativa

```
public static
int search(String pat, String txt) {
  int i, n = txt.length();
  int j, m = pat.length();
  for(i=0, j=0; i < n && j < m; i++) {
     if(txt.charAt(i)==pat.charAt(j))j++;
     else {
        i -= j; // retrocesso
        j = 0;
     }
  }
  if (j == m) return i - m;
  return n; }
```

Ideia básica do algoritmo

$P =$ a b a b b a b a b b a

```
    0  1  2  3  4  5  6  7  8  9 10 11 12 13 14 15 16 17 18 19 20 21 22
    a  b  a  a  b  a  b  a  b  b  a  b  a  b  a  b  b  a  b  a  b  b  a  T
 0  a
```

Ideia básica do algoritmo

$P =$ a b a b b a b a b b a

```
    0  1  2  3  4  5  6  7  8  9 10 11 12 13 14 15 16 17 18 19 20 21 22
    a  b  a  a  b  a  b  a  b  b  a  b  a  b  a  b  b  a  b  a  b  b  a  T
 0  a
 1  a  b
```

Ideia básica do algoritmo

$P =$ a b a b b a b a b b a

```
    0  1  2  3  4  5  6  7  8  9 10 11 12 13 14 15 16 17 18 19 20 21 22
    a  b  a  a  b  a  b  a  b  b  a  b  a  b  a  b  b  a  b  a  b  b  a  T
 0  a
 1  a  b
 2  a  b  a
```

Ideia básica do algoritmo

$P =$ a b a b b a b a b b a

```
    0  1  2  3  4  5  6  7  8  9 10 11 12 13 14 15 16 17 18 19 20 21 22
    a  b  a  a  b  a  b  a  b  b  a  b  a  b  a  b  b  a  b  a  b  b  a  T
 0  a
 1  a  b
 2  a  b  a
 3        a
```

Ideia básica do algoritmo

$P =$ a b a b b a b a b b a

```
   0  1  2  3  4  5  6  7  8  9  10 11 12 13 14 15 16 17 18 19 20 21 22
   a  b  a  a  b  a  b  a  b  b  a  b  a  b  a  b  b  a  b  a  b  b  a  T
0  a
1  a  b
2  a  b  a
3           a
4           a  b
```

Ideia básica do algoritmo

$P =$ a b a b b a b a b b a

```
   0  1  2  3  4  5  6  7  8  9  10 11 12 13 14 15 16 17 18 19 20 21 22
   a  b  a  a  b  a  b  a  b  b  a  b  a  b  a  b  b  a  b  a  b  b  a  T
0  a
1  a  b
2  a  b  a
3           a
4           a  b
5           a  b  a
```

Ideia básica do algoritmo

$P =$ a b a b b a b a b b a

```
   0  1  2  3  4  5  6  7  8  9  10 11 12 13 14 15 16 17 18 19 20 21 22
   a  b  a  a  b  a  b  a  b  b  a  b  a  b  a  b  b  a  b  a  b  b  a  T
0  a
1  a  b
2  a  b  a
3           a
4           a  b
5           a  b  a
6           a  b  a  b
```

Ideia básica do algoritmo

$P =$ a b a b b a b a b b a

```
   0  1  2  3  4  5  6  7  8  9  10 11 12 13 14 15 16 17 18 19 20 21 22
   a  b  a  a  b  a  b  a  b  b  a  b  a  b  a  b  b  a  b  a  b  b  a  T
0  a
1  a  b
2  a  b  a
3           a
4           a  b
5           a  b  a
6           a  b  a  b
7                 a  b  a
```

Ideia básica do algoritmo

$P =$ a b a b b a b a b b a

```
   0  1  2  3  4  5  6  7  8  9  10 11 12 13 14 15 16 17 18 19 20 21 22
   a  b  a  a  b  a  b  a  b  b  a  b  a  b  a  b  b  a  b  a  b  b  a  T
0  a
1  a  b
2  a  b  a
3           a
4           a  b
5           a  b  a
6           a  b  a  b
7                 a  b  a
8                 a  b  a  b
```

Ideia básica do algoritmo

$P =$ a b a b b a b a b b a

```
   0  1  2  3  4  5  6  7  8  9  10 11 12 13 14 15 16 17 18 19 20 21 22
   a  b  a  a  b  a  b  a  b  b  a  b  a  b  a  b  b  a  b  a  b  b  a  T
0  a
1  a  b
2  a  b  a
3           a
4           a  b
5           a  b  a
6           a  b  a  b
7                 a  b  a
8                 a  b  a  b
9                 a  b  a  b  b
```

Ideia básica do algoritmo

$P =$ a b a b b a b a b b a

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	
	a	b	a	a	b	a	b	a	b	b	a	b	a	b	a	b	b	a	b	a	b	b	a	T
0	a																							
1	a	b																						
2	a	b	a																					
3				a																				
4				a	b																			
5				a	b	a																		
6				a	b	a	b																	
7						a	b	a																
8						a	b	a	b															
9						a	b	a	b	b														
10						a	b	a	b	b	a													

Ideia básica do algoritmo

$P =$ a b a b b a b a b b a

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	
	a	b	a	a	b	a	b	a	b	b	a	b	a	b	a	b	b	a	b	a	b	b	a	T
0	a																							
1	a	b																						
2	a	b	a																					
3				a																				
4				a	b																			
5				a	b	a																		
6				a	b	a	b																	
7						a	b	a																
8						a	b	a	b															
9						a	b	a	b	b														
10						a	b	a	b	b	a													
11						a	b	a	b	b	a	b												

Ideia básica do algoritmo

$P =$ a b a b b a b a b b a

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	
	a	b	a	a	b	a	b	a	b	b	a	b	a	b	a	b	b	a	b	a	b	b	a	T
0	a																							
1	a	b																						
2	a	b	a																					
3				a																				
4				a	b																			
5				a	b	a																		
6				a	b	a	b																	
7						a	b	a																
8						a	b	a	b															
9						a	b	a	b	b														
10						a	b	a	b	b	a													
11						a	b	a	b	b	a	b												
12						a	b	a	b	b	a	b	a											

Ideia básica do algoritmo

$P =$ a b a b b a b a b b a

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	
	a	b	a	a	b	a	b	a	b	b	a	b	a	b	a	b	b	a	b	a	b	b	a	T
0	a																							
1	a	b																						
2	a	b	a																					
3				a																				
4				a	b																			
5				a	b	a																		
6				a	b	a	b																	
7						a	b	a																
8						a	b	a	b															
9						a	b	a	b	b														
10						a	b	a	b	b	a													
11						a	b	a	b	b	a	b												
12						a	b	a	b	b	a	b	a											
13						a	b	a	b	b	a	b	a	b										

Ideia básica do algoritmo

$P =$ a b a b b a b a b b a

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	
	a	b	a	a	b	a	b	a	b	b	a	b	a	b	a	b	b	a	b	a	b	b	a	T
0	a																							
1	a	b																						
2	a	b	a																					
3				a																				
4				a	b																			
5				a	b	a																		
6				a	b	a	b																	
7						a	b	a																
8						a	b	a	b															
9						a	b	a	b	b														
10						a	b	a	b	b	a													
11						a	b	a	b	b	a	b												
12						a	b	a	b	b	a	b	a											
13						a	b	a	b	b	a	b	a	b										
14													a	b	a									

Ideia básica do algoritmo

$P =$ a b a b b a b a b b a

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	
	a	b	a	a	b	a	b	a	b	b	a	b	a	b	a	b	b	a	b	a	b	b	a	T
0	a																							
1	a	b																						
2	a	b	a																					
3				a																				
4				a	b																			
5				a	b	a																		
6				a	b	a	b																	
7						a	b	a																
8						a	b	a	b															
9						a	b	a	b	b														
10						a	b	a	b	b	a													
11						a	b	a	b	b	a	b												
12						a	b	a	b	b	a	b	a											
13						a	b	a	b	b	a	b	a	b										
14													a	b	a									
15													a	b	a	b								

Ideia básica do algoritmo

$P =$ a b a b b a b a b b a

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	
	a	b	a	a	b	a	b	a	b	b	a	b	a	b	a	b	b	a	b	a	b	b	a	T
0	a																							
1	a	b																						
2	a	b	a																					
3				a																				
4				a	b																			
5				a	b	a																		
6				a	b	a	b																	
7						a	b	a																
8						a	b	a	b															
9						a	b	a	b	b														
10						a	b	a	b	b	a													
11						a	b	a	b	b	a	b												
12						a	b	a	b	b	a	b	a											
13						a	b	a	b	b	a	b	a	b										
14													a	b	a									
15													a	b	a	b								
16													a	b	a	b	b							

Ideia básica do algoritmo

$P =$ a b a b b a b a b b a

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	
	a	b	a	a	b	a	b	a	b	b	a	b	a	b	a	b	b	a	b	a	b	b	a	T
0	a																							
1	a	b																						
2	a	b	a																					
3				a																				
4				a	b																			
5				a	b	a																		
6				a	b	a	b																	
7						a	b	a																
8						a	b	a	b															
9						a	b	a	b	b														
10						a	b	a	b	b	a													
11						a	b	a	b	b	a	b												
12						a	b	a	b	b	a	b	a											
13						a	b	a	b	b	a	b	a	b										
14													a	b	a									
15													a	b	a	b								
16													a	b	a	b	b							
17													a	b	a	b	b	a						

Ideia básica do algoritmo

$P =$ a b a b b a b a b b a

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	
	a	b	a	a	b	a	b	a	b	b	a	b	a	b	a	b	b	a	b	a	b	b	a	T
0	a																							
1	a	b																						
2	a	b	a																					
3				a																				
4				a	b																			
5				a	b	a																		
6				a	b	a	b																	
7						a	b	a																
8						a	b	a	b															
9						a	b	a	b	b														
10						a	b	a	b	b	a													
11						a	b	a	b	b	a	b												
12						a	b	a	b	b	a	b	a											
13						a	b	a	b	b	a	b	a	b										
14													a	b	a									
15													a	b	a	b								
16													a	b	a	b	b							
17													a	b	a	b	b	a						

Ideia básica do algoritmo

$P =$ a b a b b a b a b b a

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	
	a	b	a	a	b	a	b	a	b	b	a	b	a	b	a	b	b	a	b	a	b	b	a	T
0	a																							
1	a	b																						
2	a	b	a																					
3				a																				
4				a	b																			
5				a	b	a																		
6				a	b	a	b																	
7						a	b	a																
8						a	b	a	b															
9						a	b	a	b	b														
10						a	b	a	b	b	a													
11						a	b	a	b	b	a	b												
12						a	b	a	b	b	a	b	a											
13						a	b	a	b	b	a	b	a	b										
14													a	b	a									
15													a	b	a	b								
16													a	b	a	b	b							
17													a	b	a	b	b	a						

Ideia básica do algoritmo

$P =$ a b a b b a b a b b a

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	
	a	b	a	a	b	a	b	a	b	b	a	b	a	b	a	b	b	a	b	a	b	b	a	T
0	a																							
1	a	b																						
2	a	b	a																					
3				a																				
4				a	b																			
5				a	b	a																		
6				a	b	a	b																	
7						a	b	a																
8						a	b	a	b															
9						a	b	a	b	b														
10						a	b	a	b	b	a													
11						a	b	a	b	b	a	b												
12						a	b	a	b	b	a	b	a											
13						a	b	a	b	b	a	b	a	b										
14													a	b	a									
15													a	b	a	b								
16													a	b	a	b	b							
17													a	b	a	b	b	a						

Ideia básica do algoritmo

$P =$ a b a b b a b a b b a

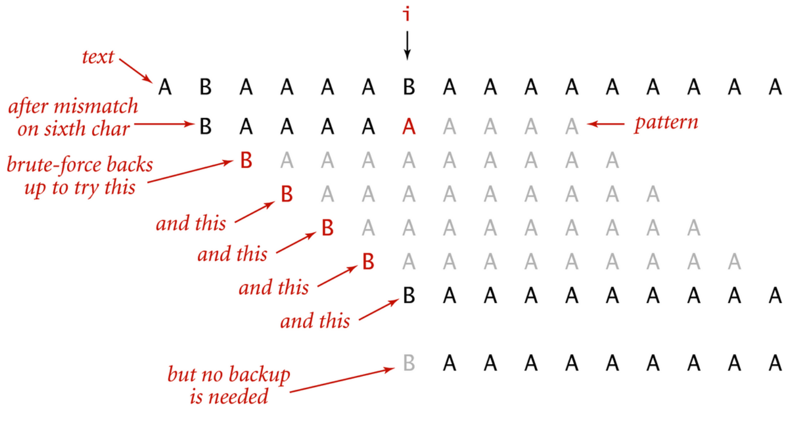
	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	
	a	b	a	a	b	a	b	a	b	b	a	b	a	b	a	b	b	a	b	a	b	b	a	T
0	a																							
1	a	b																						
2	a	b	a																					
3				a																				
4				a	b																			
5				a	b	a																		
6				a	b	a	b																	
7						a	b	a																
8						a	b	a	b															
9						a	b	a	b	b														
10						a	b	a	b	b	a													
11						a	b	a	b	b	a	b												
12						a	b	a	b	b	a	b	a											
13						a	b	a	b	b	a	b	a	b										
14													a	b	a									
15													a	b	a	b								
16													a	b	a	b	b							
17													a	b	a	b	b	a						

Ideia básica do algoritmo

$P =$ a b a b b a b a b b a

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	
	a	b	a	a	b	a	b	a	b	b	a	b	a	b	a	b	b	a	b	a	b	b	a	T
0	a																							
1	a	b																						
2	a	b	a																					
3				a																				
4				a	b																			
5				a	b	a																		
6				a	b	a	b																	
7						a	b	a																
8						a	b	a	b															
9						a	b	a	b	b														
10						a	b	a	b	b	a													
11						a	b	a	b	b	a	b												
12						a	b	a	b	b	a	b	a											
13						a	b	a	b	b	a	b	a	b										
14													a	b	a									
15													a	b	a	b								
16													a	b	a	b	b							
17													a	b	a	b	b	a						

Ideia básica do algoritmo



Text pointer backup in substring searching

Ideia geral

Quando encontramos um conflito entre `txt[i]` e `pat[j]`, **não** é necessário retroceder `i` e passar a comparar `txt[i-j+1..]` com `pat[0..]`.

Basta:

encontrar o comprimento do maior prefixo de `pat[0..]` que é sufixo de `txt[..i]`,

ou seja,

encontrar o maior `k` tal que `pat[0..k-1]` é igual a `txt[i-k+1..i]` que é igual a `pat[j-k+1..j-1]+txt[i]`,

e passar a comparar `txt[i+1..]` com `pat[k..]`.

Ideia geral

Exemplo: texto `CAABAABAAAA` e padrão `AABAAA`: depois do conflito entre `txt[6]` e `pat[5]`, não precisamos retroceder no texto: podemos continuar e comparar `txt[7..]` com `pat[3..]`:

	C	A	A	B	A	A	B	A	A	A	A
uma tentativa:		A	A	B	A	A	A				
não precisa tentar:			A	A	B	A	A	A			
não precisa tentar:				A	A	B	A	A	A		
próxima tentativa:					A	A	B	A	A	A	

Algoritmo KMP

Examina os caracteres de `txt` um a um, da esquerda para a direita, sem nunca retroceder.

Em cada iteração, o algoritmo sabe qual posição `k` de `pat` deve ser emparelhada com a próxima posição `i+1` de `txt`.

Ou seja, no fim de cada iteração, o algoritmo sabe qual índice `k` deve fazer o papel de `j` na próxima iteração.

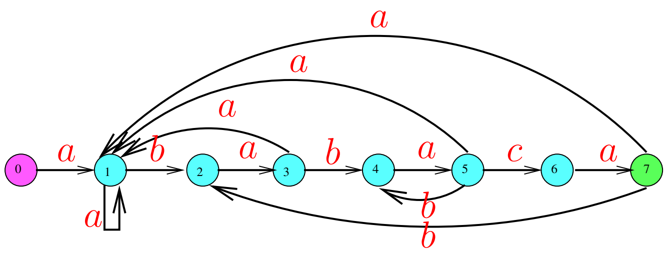
Algoritmo KMP

O algoritmo KMP usa uma tabela `dfa[][]` que armazena os `índices mágicos k`.

O nome da tabela deriva da expressão *deterministic finite-state automaton*.

As colunas da tabela são indexadas pelos índices `0..m-1` do padrão e as linhas são indexadas pelo alfabeto, que é o conjunto de todos os caracteres do texto e do padrão.

Autômato de estados determinístico (DFA)



$0\,..\,7$ = conjunto de estados

Σ = {a, b, c} = alfabeto

δ = função de transição

0 é estado inicial e 7 é estado final

Exemplo: pat = ABABAC

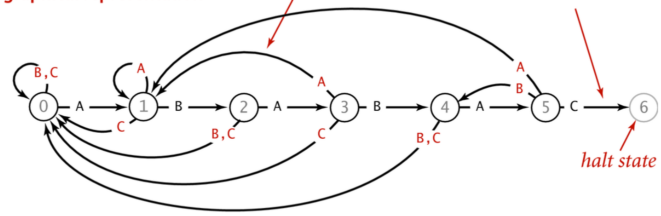
internal representation

		0	1	2	3	4	5
pat.charAt(j)		A	B	A	B	A	C
dfa[][j]	A	1	1	3	1	5	1
	B	0	2	0	4	0	4
	C	0	0	0	0	0	6

mismatch transition (back up)

match transition (increment)

graphical representation



halt state

Autômato finito determinístico (DFA)

O algoritmo KMP simula o funcionamento do autômato de estados.

O autômato começa no estado 0 e examina os caracteres do texto, um de cada vez, da esquerda para a direita, mudando para um novo estado cada vez que lê um caractere do texto.

Se atingir o estado m, dizemos que o autômato reconheceu ou aceitou o padrão.

Se chegar ao fim do texto sem atingir o estado m, sabemos que o padrão não ocorre no texto.

Autômato finito determinístico (DFA)

O autômato está no estado j se acabou de casar os j primeiros caracteres do padrão com um segmento do texto, ou seja, se acabou de casar pat[0 . . j-1] com txt[i-j . . i-1].

Para cada estado j, a transição que corresponde ao caractere pat[j] é de casamento e leva ao estado j+1.

Todas as outras transições que começam no estado j são de conflito e levam a um estado $\leq$ j.

O autômato de estados é uma ideia **muito importante** em compilação, na teoria da computação, etc.

Autômato finito determinístico (DFA)

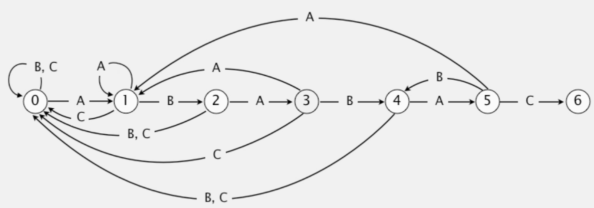
Um **autômato finito** é formado uma 5-upla (Q, Σ, δ, q_0, F), onde

- Q é um conjunto finitos de estados,
- Σ é um conjunto finito chamado alfabeto,
- δ : Q $\times$ $\Sigma \to$ Q é a função de transição,
- $q_0 \in$ Q é o estado inicial, e
- F $\subseteq$ Q é o conjunto de aceitação.

Knuth-Morris-Pratt demo: DFA simulation

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

		0	1	2	3	4	5
pat.charAt(j)		A	B	A	B	A	C
dfa[][j]	A	1	1	3	1	5	1
	B	0	2	0	4	0	4
	C	0	0	0	0	0	6

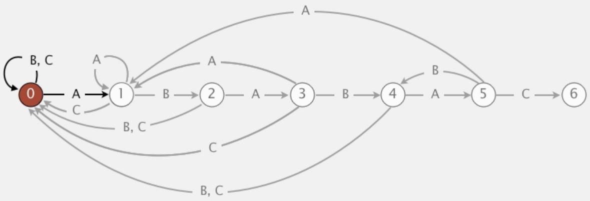


3

Knuth-Morris-Pratt demo: DFA simulation

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

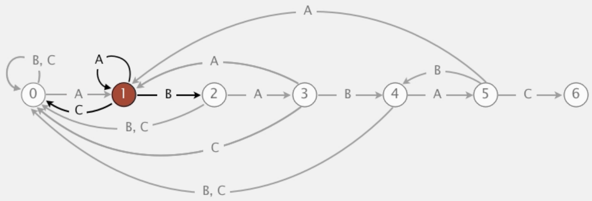
		0	1	2	3	4	5
pat.charAt(j)		A	B	A	B	A	C
dfa[][j]	A	1	1	3	1	5	1
	B	0	2	0	4	0	4
	C	0	0	0	0	0	6



Knuth-Morris-Pratt demo: DFA simulation

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

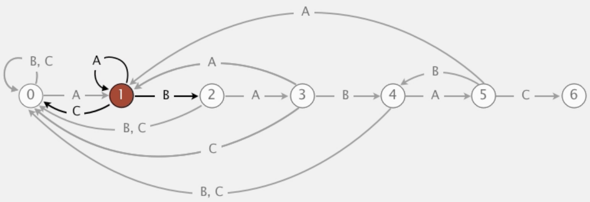
		0	1	2	3	4	5
pat.charAt(j)		A	B	A	B	A	C
dfa[][j]	A	1	1	3	1	5	1
	B	0	2	0	4	0	4
	C	0	0	0	0	0	6



Knuth-Morris-Pratt demo: DFA simulation

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

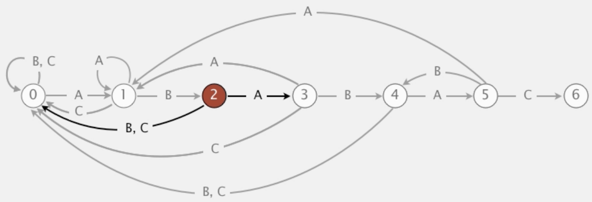
		0	1	2	3	4	5
pat.charAt(j)		A	B	A	B	A	C
dfa[][j]	A	1	1	3	1	5	1
	B	0	2	0	4	0	4
	C	0	0	0	0	0	6



Knuth-Morris-Pratt demo: DFA simulation

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

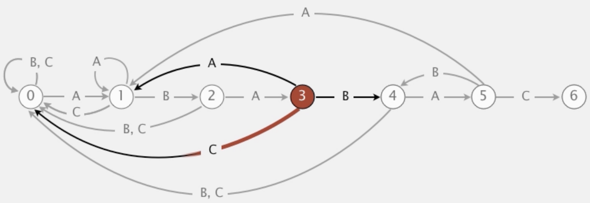
		0	1	2	3	4	5
pat.charAt(j)		A	B	A	B	A	C
dfa[][j]	A	1	1	3	1	5	1
	B	0	2	0	4	0	4
	C	0	0	0	0	0	6



Knuth-Morris-Pratt demo: DFA simulation

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

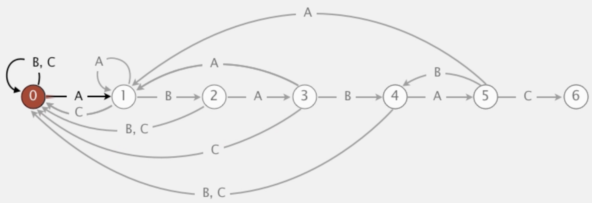
		0	1	2	3	4	5
pat.charAt(j)		A	B	A	B	A	C
dfa[][j]	A	1	1	3	1	5	1
	B	0	2	0	4	0	4
	C	0	0	0	0	0	6



Knuth-Morris-Pratt demo: DFA simulation

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

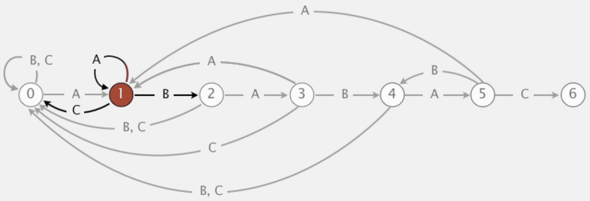
		0	1	2	3	4	5
pat.charAt(j)		A	B	A	B	A	C
dfa[][j]	A	1	1	3	1	5	1
	B	0	2	0	4	0	4
	C	0	0	0	0	0	6



Knuth-Morris-Pratt demo: DFA simulation

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

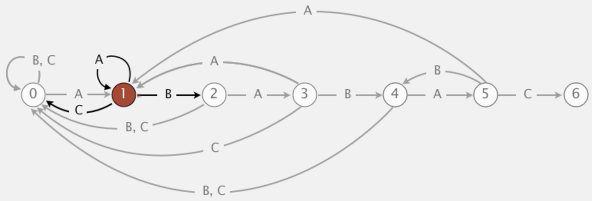
		0	1	2	3	4	5
pat.charAt(j)		A	B	A	B	A	C
dfa[][j]	A	1	1	3	1	5	1
	B	0	2	0	4	0	4
	C	0	0	0	0	0	6



Knuth-Morris-Pratt demo: DFA simulation

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

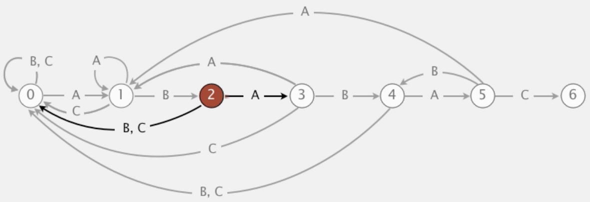
		0	1	2	3	4	5
pat.charAt(j)		A	B	A	B	A	C
dfa[][j]	A	1	1	3	1	5	1
	B	0	2	0	4	0	4
	C	0	0	0	0	0	6



Knuth-Morris-Pratt demo: DFA simulation

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

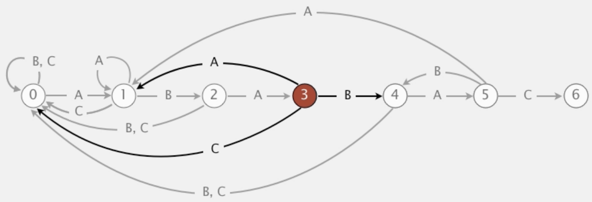
		0	1	2	3	4	5
pat.charAt(j)		A	B	A	B	A	C
dfa[][j]	A	1	1	3	1	5	1
	B	0	2	0	4	0	4
	C	0	0	0	0	0	6



Knuth-Morris-Pratt demo: DFA simulation

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

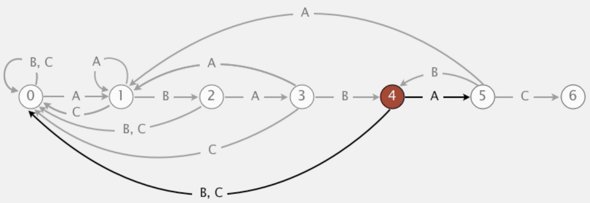
		0	1	2	3	4	5
pat.charAt(j)		A	B	A	B	A	C
dfa[][j]	A	1	1	3	1	5	1
	B	0	2	0	4	0	4
	C	0	0	0	0	0	6



Knuth-Morris-Pratt demo: DFA simulation

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

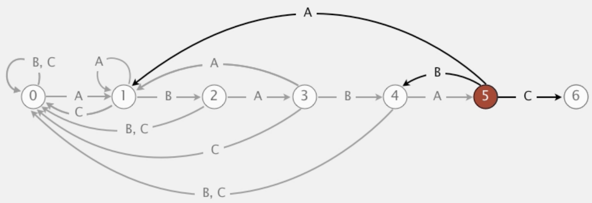
		0	1	2	3	4	5
pat.charAt(j)		A	B	A	B	A	C
dfa[][j]	A	1	1	3	1	5	1
	B	0	2	0	4	0	4
	C	0	0	0	0	0	6

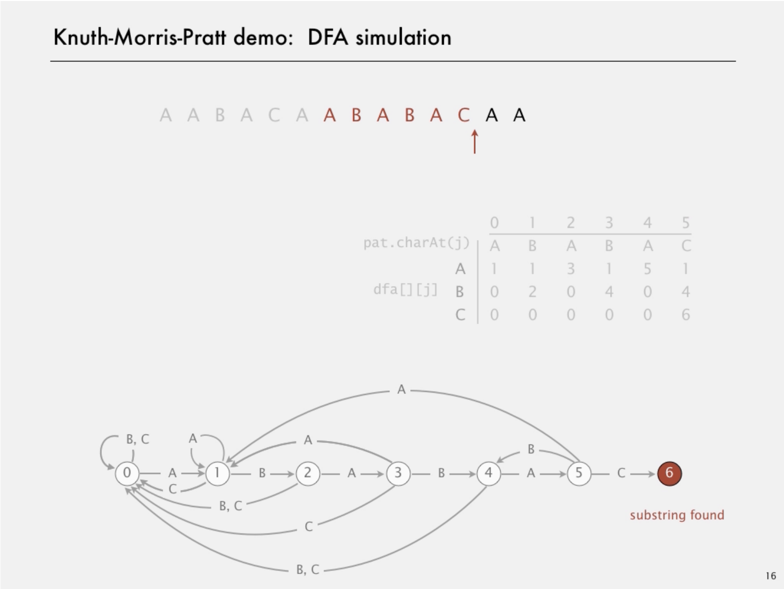


Knuth-Morris-Pratt demo: DFA simulation

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

		0	1	2	3	4	5
pat.charAt(j)		A	B	A	B	A	C
dfa[][j]	A	1	1	3	1	5	1
	B	0	2	0	4	0	4
	C	0	0	0	0	0	6





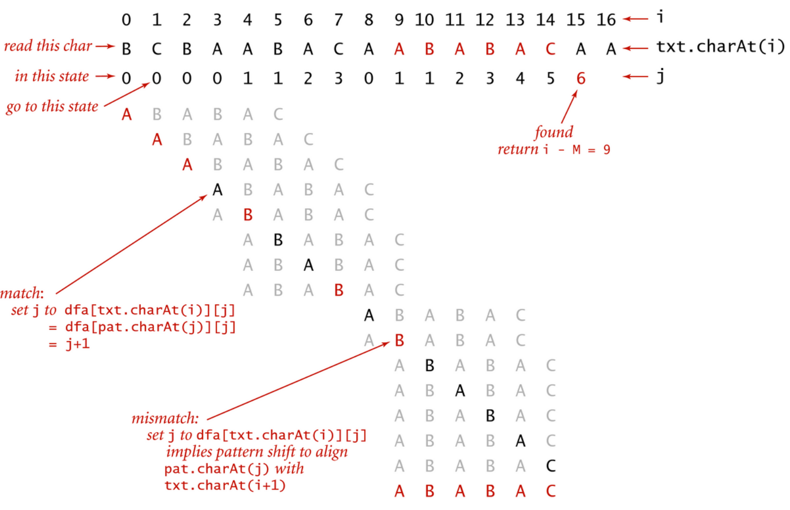
Algoritmo KMP

Retorna a posição a partir de onde `pat` ocorre em `txt` se `pat` não ocorre em `txt` retorna `n`.

```
public int search(String txt) {
   int i, n = txt.length();
   int j, m = pat.length();

   for (i = 0, j = 0; i < n && j < m; i++)
      j = dfa[txt.charAt(i)][j];

   if (j == m) return i - m;
   return n;
}
```



Trace of KMP substring search (DFA simulation) for A B A B A C

Invariantes

O método `search()` de KMP tem os seguintes invariantes.

Imediatamente antes do teste `i < n && j < m` vale que:

- `pat` não ocorre em `txt[0 . . i-1]`;
- `pat[0 . . k]` é diferente de `txt[i-k . . i]` para todo `k` no conjunto `j+1 . . m-1`; e
- `pat[0 . . j-1]` é igual a `txt[i-j . . i-1]`.

Autômato de estados determinístico (DFA)

A tabela `dfa[][]` representa uma máquina imaginária conhecida como **autômato de estados** (*deterministic finite-state automaton, DFA*).

Os estados do autômato correspondem aos índices `0 . . m-1` de `pat`.

Também há um estado *final* `m`.

Para cada estado e cada caractere do alfabeto, há uma transição que leva desse estado a um outro.

Construção do DFA

Para construir a tabela `dfa[][]` que representa o autômato podemos pré-processar o padrão `pat` desde que o alfabeto de `txt` seja conhecido.

Para qualquer caractere `c` do alfabeto e qualquer `j` em `0 . . m-1`, o valor de `dfa[c][j]` é

> *o comprimento do maior prefixo de `pat[0 . . j]` que é sufixo de `pat[0 . . j-1]+c`.*

Uma implementação literal dessa definição faria cerca de Rm^3 comparações entre caracteres para calcular a tabela `dfa[][]`, sendo `R` o número de caracteres do alfabeto.

Exemplo: padrão ABABAC e alfabeto A B C

1	A					
		0				
2	A	B				
		0	0			
3	A	B	A			
		0	0	1		
4	A	B	A	B		
		0	0	1	2	
5	A	B	A	B	A	
		0	0	1	2	3

restart states
X

DFA simulations to compute

Knuth-Morris-Pratt demo: DFA construction

Include one state for each character in pattern (plus accept state).

		0	1	2	3	4	5
pat.charAt(j)		A	B	A	B	A	C
dfa[][j]	A						
	B						
	C						

Constructing the DFA for KMP substring search for A B A B A C

Knuth-Morris-Pratt demo: DFA construction

Include one state for each character in pattern (plus accept state).

		0	1	2	3	4	5
pat.charAt(j)		A	B	A	B	A	C
dfa[][j]	A						
	B						
	C						

Constructing the DFA for KMP substring search for A B A B A C

0 1 2 3 4 5 6

Knuth-Morris-Pratt demo: DFA construction

Match transition. If in state j and next char c == pat.charAt(j), go to j+1.

first j characters of pattern have already been matched
next char matches
now first j+1 characters of pattern have been matched

		0	1	2	3	4	5
pat.charAt(j)		A	B	A	B	A	C
dfa[][j]	A	1		3		5	
	B		2		4		
	C						6

Constructing the DFA for KMP substring search for A B A B A C

$0 \xrightarrow{A} 1 \xrightarrow{B} 2 \xrightarrow{A} 3 \xrightarrow{B} 4 \xrightarrow{A} 5 \xrightarrow{C} 6$

Knuth-Morris-Pratt demo: DFA construction

Mismatch transition: back up if c != pat.charAt(j).

		0	1	2	3	4	5
pat.charAt(j)		A	B	A	B	A	C
dfa[][j]	A	1		3		5	
	B		2		4		
	C						6

Constructing the DFA for KMP substring search for A B A B A C

j

$0 \xrightarrow{A} 1 \xrightarrow{B} 2 \xrightarrow{A} 3 \xrightarrow{B} 4 \xrightarrow{A} 5 \xrightarrow{C} 6$

Knuth-Morris-Pratt demo: DFA construction

Mismatch transition: back up if c != pat.charAt(j).

		0	1	2	3	4	5
pat.charAt(j)		A	B	A	B	A	C
dfa[][j]	A	1		3		5	
	B	0	2		4		
	C	0					6

Constructing the DFA for KMP substring search for A B A B A C

B, C j

$0 \xrightarrow{A} 1 \xrightarrow{B} 2 \xrightarrow{A} 3 \xrightarrow{B} 4 \xrightarrow{A} 5 \xrightarrow{C} 6$

Knuth-Morris-Pratt demo: DFA construction

Mismatch transition: back up if `c != pat.charAt(j)`.

		0	1	2	3	4	5
pat.charAt(j)		A	B	A	B	A	C
dfa[][j]	A	1		3		5	
	B	0	2		4		
	C	0					6

Constructing the DFA for KMP substring search for A B A B A C

Knuth-Morris-Pratt demo: DFA construction

Mismatch transition: back up if `c != pat.charAt(j)`.

		0	1	2	3	4	5
pat.charAt(j)		A	B	A	B	A	C
dfa[][j]	A	1	1	3		5	
	B	0	2		4		
	C	0	0				6

Constructing the DFA for KMP substring search for A B A B A C

Knuth-Morris-Pratt demo: DFA construction

Mismatch transition: back up if `c != pat.charAt(j)`.

		0	1	2	3	4	5
pat.charAt(j)		A	B	A	B	A	C
dfa[][j]	A	1	1	3		5	
	B	0	2		4		
	C	0	0				6

Constructing the DFA for KMP substring search for A B A B A C

Knuth-Morris-Pratt demo: DFA construction

Mismatch transition: back up if `c != pat.charAt(j)`.

		0	1	2	3	4	5
pat.charAt(j)		A	B	A	B	A	C
dfa[][j]	A	1	1	3		5	
	B	0	2	0	4		
	C	0	0	0			6

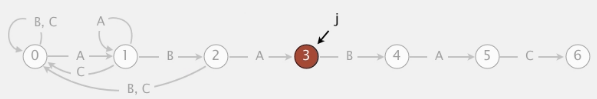
Constructing the DFA for KMP substring search for A B A B A C

Knuth-Morris-Pratt demo: DFA construction

Mismatch transition: back up if `c != pat.charAt(j)`.

		0	1	2	3	4	5
pat.charAt(j)		A	B	A	B	A	C
dfa[][j]	A	1	1	3		5	
	B	0	2	0	4		
	C	0	0	0			6

Constructing the DFA for KMP substring search for A B A B A C

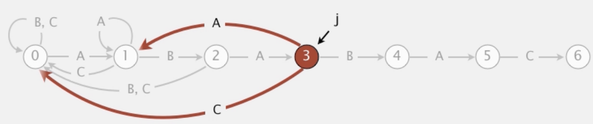


Knuth-Morris-Pratt demo: DFA construction

Mismatch transition: back up if `c != pat.charAt(j)`.

		0	1	2	3	4	5
pat.charAt(j)		A	B	A	B	A	C
dfa[][j]	A	1	1	3	1	5	
	B	0	2	0	4		
	C	0	0	0	0		6

Constructing the DFA for KMP substring search for A B A B A C

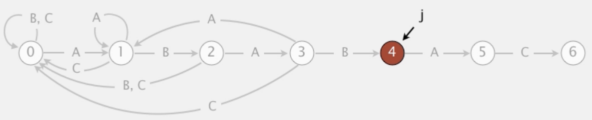


Knuth-Morris-Pratt demo: DFA construction

Mismatch transition: back up if `c != pat.charAt(j)`.

		0	1	2	3	4	5
pat.charAt(j)		A	B	A	B	A	C
dfa[][j]	A	1	1	3	1	5	
	B	0	2	0	4		
	C	0	0	0	0		6

Constructing the DFA for KMP substring search for A B A B A C

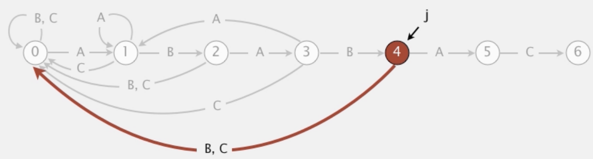


Knuth-Morris-Pratt demo: DFA construction

Mismatch transition: back up if `c != pat.charAt(j)`.

		0	1	2	3	4	5
pat.charAt(j)		A	B	A	B	A	C
dfa[][j]	A	1	1	3	1	5	
	B	0	2	0	4	0	
	C	0	0	0	0	0	6

Constructing the DFA for KMP substring search for A B A B A C

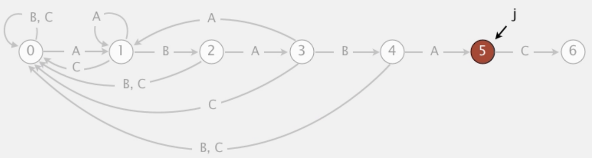


Knuth-Morris-Pratt demo: DFA construction

Mismatch transition: back up if `c != pat.charAt(j)`.

		0	1	2	3	4	5
pat.charAt(j)		A	B	A	B	A	C
dfa[][j]	A	1	1	3	1	5	
	B	0	2	0	4	0	
	C	0	0	0	0	0	6

Constructing the DFA for KMP substring search for A B A B A C

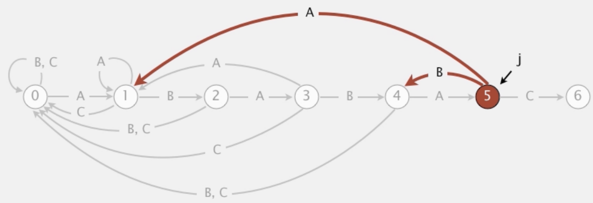


Knuth-Morris-Pratt demo: DFA construction

Mismatch transition: back up if `c != pat.charAt(j)`.

		0	1	2	3	4	5
pat.charAt(j)		A	B	A	B	A	C
dfa[][j]	A	1	1	3	1	5	1
	B	0	2	0	4	0	4
	C	0	0	0	0	0	6

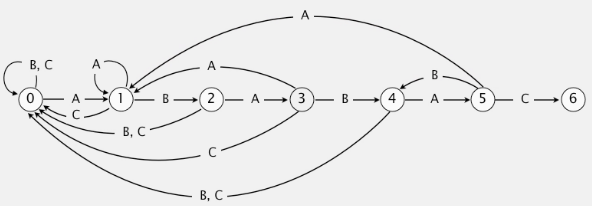
Constructing the DFA for KMP substring search for A B A B A C



Knuth-Morris-Pratt demo: DFA construction

		0	1	2	3	4	5
pat.charAt(j)		A	B	A	B	A	C
dfa[][j]	A	1	1	3	1	5	1
	B	0	2	0	4	0	4
	C	0	0	0	0	0	6

Constructing the DFA for KMP substring search for A B A B A C



Knuth-Morris-Pratt demo: DFA construction in linear time

Include one state for each character in pattern (plus accept state).

		0	1	2	3	4	5
pat.charAt(j)		A	B	A	B	A	C
dfa[][j]	A						
	B						
	C						

Constructing the DFA for KMP substring search for A B A B A C

Knuth-Morris-Pratt demo: DFA construction in linear time

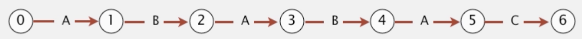
Match transition. For each state `j`, `dfa[pat.charAt(j)][j] = j+1`.

first j characters of pattern have already been matched

now first j+1 characters of pattern have been matched

		0	1	2	3	4	5
pat.charAt(j)		A	B	A	B	A	C
dfa[][j]	A	1		3		5	
	B		2		4		
	C						6

Constructing the DFA for KMP substring search for A B A B A C



Knuth-Morris-Pratt demo: DFA construction in linear time

Mismatch transition. For state `0` and char `c != pat.charAt(j)`, set `dfa[c][0] = 0`.

		0	1	2	3	4	5
pat.charAt(j)		A	B	A	B	A	C
dfa[][j]	A	1		3		5	
	B	0	2		4		
	C	0					6

Constructing the DFA for KMP substring search for A B A B A C

Knuth-Morris-Pratt demo: DFA construction in linear time

Mismatch transition. For state `0` and char `c != pat.charAt(j)`, set `dfa[c][0] = 0`.

		0	1	2	3	4	5
pat.charAt(j)		A	B	A	B	A	C
dfa[][j]	A	1		3		5	
	B	0	2		4		
	C	0					6

Constructing the DFA for KMP substring search for A B A B A C

Knuth-Morris-Pratt demo: DFA construction in linear time

Mismatch transition. For each state `j` and char `c != pat.charAt(j)`, set `dfa[c][j] = dfa[c][X]`; then update `X = dfa[pat.charAt(j)][X]`.

X = simulation of empty string

		0	1	2	3	4	5
pat.charAt(j)		A	B	A	B	A	C
dfa[][j]	A	1		3		5	
	B	0	2		4		
	C	0					6

Constructing the DFA for KMP substring search for A B A B A C

Knuth-Morris-Pratt demo: DFA construction in linear time

Mismatch transition. For each state `j` and char `c != pat.charAt(j)`, set `dfa[c][j] = dfa[c][X]`; then update `X = dfa[pat.charAt(j)][X]`.

X = simulation of empty string

		0	1	2	3	4	5
pat.charAt(j)		A	B	A	B	A	C
dfa[][j]	A	1	1	3		5	
	B	0	2		4		
	C	0	0				6

Constructing the DFA for KMP substring search for A B A B A C

Knuth-Morris-Pratt demo: DFA construction in linear time

Mismatch transition. For each state `j` and char `c != pat.charAt(j)`, set `dfa[c][j] = dfa[c][X]`; then update `X = dfa[pat.charAt(j)][X]`.

X = simulation of empty string

		0	1	2	3	4	5
pat.charAt(j)		A	B	A	B	A	C
dfa[][j]	A	1	1	3		5	
	B	0	2		4		
	C	0	0				6

Constructing the DFA for KMP substring search for A B A B A C

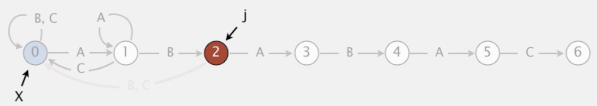
Knuth-Morris-Pratt demo: DFA construction in linear time

Mismatch transition. For each state `j` and char `c != pat.charAt(j)`, set `dfa[c][j] = dfa[c][X]`; then update `X = dfa[pat.charAt(j)][X]`.

X = simulation of B

		0	1	2	3	4	5
pat.charAt(j)		A	B	A	B	A	C
dfa[][j]	A	1	1	3		5	
	B	0	2		4		
	C	0	0				6

Constructing the DFA for KMP substring search for A B A B A C



Knuth-Morris-Pratt demo: DFA construction in linear time

Mismatch transition. For each state `j` and char `c != pat.charAt(j)`, set `dfa[c][j] = dfa[c][X]`; then update `X = dfa[pat.charAt(j)][X]`.

X = simulation of B

		0	1	2	3	4	5
pat.charAt(j)		A	B	A	B	A	C
dfa[][j]	A	1	1	3		5	
	B	0	2	0	4		
	C	0	0	0			6

Constructing the DFA for KMP substring search for A B A B A C

Knuth-Morris-Pratt demo: DFA construction in linear time

Mismatch transition. For each state `j` and char `c != pat.charAt(j)`, set `dfa[c][j] = dfa[c][X]`; then update `X = dfa[pat.charAt(j)][X]`.

X = simulation of B

		0	1	2	3	4	5
pat.charAt(j)		A	B	A	B	A	C
dfa[][j]	A	1	1	3		5	
	B	0	2	0	4		
	C	0	0	0			6

Constructing the DFA for KMP substring search for A B A B A C

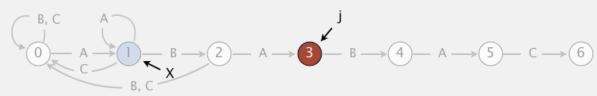
Knuth-Morris-Pratt demo: DFA construction in linear time

Mismatch transition. For each state `j` and char `c != pat.charAt(j)`, set `dfa[c][j] = dfa[c][X]`; then update `X = dfa[pat.charAt(j)][X]`.

X = simulation of B A

		0	1	2	3	4	5
pat.charAt(j)		A	B	A	B	A	C
dfa[][j]	A	1	1	3		5	
	B	0	2	0	4		
	C	0	0	0			6

Constructing the DFA for KMP substring search for A B A B A C



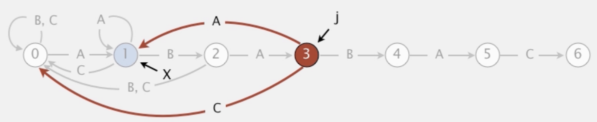
Knuth-Morris-Pratt demo: DFA construction in linear time

Mismatch transition. For each state `j` and char `c != pat.charAt(j)`, set `dfa[c][j] = dfa[c][X]`; then update `X = dfa[pat.charAt(j)][X]`.

X = simulation of B A

		0	1	2	3	4	5
pat.charAt(j)		A	B	A	B	A	C
dfa[][j]	A	1	1	3	1	5	
	B	0	2	0	4		
	C	0	0	0	0		6

Constructing the DFA for KMP substring search for A B A B A C



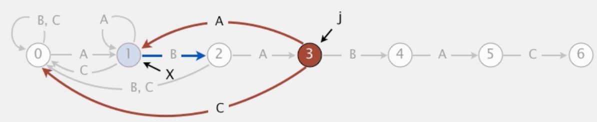
Knuth-Morris-Pratt demo: DFA construction in linear time

Mismatch transition. For each state `j` and char `c != pat.charAt(j)`, set `dfa[c][j] = dfa[c][X]`; then update `X = dfa[pat.charAt(j)][X]`.

X = simulation of B A

		0	1	2	3	4	5
pat.charAt(j)		A	B	A	B	A	C
dfa[][j]	A	1	1	3	1	5	
	B	0	2	0	4		
	C	0	0	0	0		6

Constructing the DFA for KMP substring search for A B A B A C



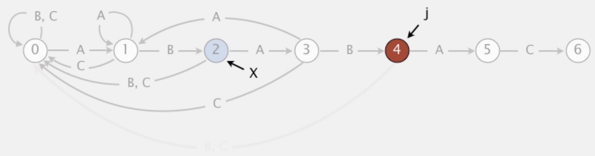
Knuth-Morris-Pratt demo: DFA construction in linear time

Mismatch transition. For each state `j` and char `c != pat.charAt(j)`, set `dfa[c][j] = dfa[c][X]`; then update `X = dfa[pat.charAt(j)][X]`.

X = simulation of B A B

		0	1	2	3	4	5
pat.charAt(j)		A	B	A	B	A	C
dfa[][j]	A	1	1	3	1	5	
	B	0	2	0	4		
	C	0	0	0	0		6

Constructing the DFA for KMP substring search for A B A B A C



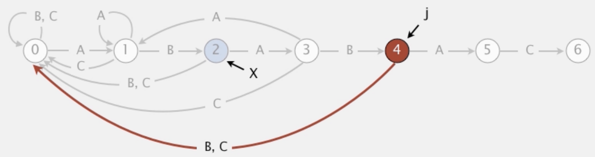
Knuth-Morris-Pratt demo: DFA construction in linear time

Mismatch transition. For each state `j` and char `c != pat.charAt(j)`, set `dfa[c][j] = dfa[c][X]`; then update `X = dfa[pat.charAt(j)][X]`.

X = simulation of B A B

		0	1	2	3	4	5
pat.charAt(j)		A	B	A	B	A	C
dfa[][j]	A	1	1	3	1	5	
	B	0	2	0	4	0	
	C	0	0	0	0	0	6

Constructing the DFA for KMP substring search for A B A B A C



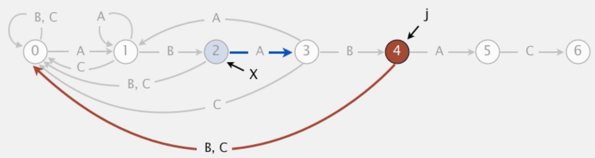
Knuth-Morris-Pratt demo: DFA construction in linear time

Mismatch transition. For each state `j` and char `c != pat.charAt(j)`, set `dfa[c][j] = dfa[c][X]`; then update `X = dfa[pat.charAt(j)][X]`.

X = simulation of B A B

		0	1	2	3	4	5
pat.charAt(j)		A	B	A	B	A	C
dfa[][j]	A	1	1	3	1	5	
	B	0	2	0	4	0	
	C	0	0	0	0	0	6

Constructing the DFA for KMP substring search for A B A B A C



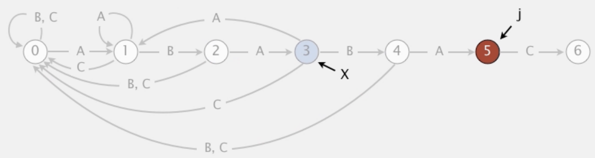
Knuth-Morris-Pratt demo: DFA construction in linear time

Mismatch transition. For each state `j` and char `c != pat.charAt(j)`, set `dfa[c][j] = dfa[c][X]`; then update `X = dfa[pat.charAt(j)][X]`.

X = simulation of B A B A

		0	1	2	3	4	5
pat.charAt(j)		A	B	A	B	A	C
dfa[][j]	A	1	1	3	1	5	
	B	0	2	0	4	0	
	C	0	0	0	0	0	6

Constructing the DFA for KMP substring search for A B A B A C



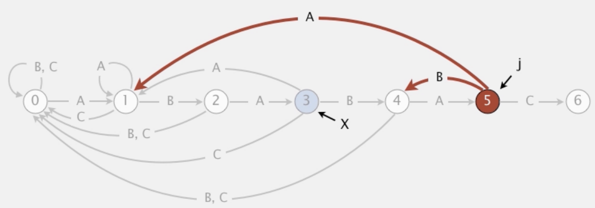
Knuth-Morris-Pratt demo: DFA construction in linear time

Mismatch transition. For each state `j` and char `c != pat.charAt(j)`, set `dfa[c][j] = dfa[c][X]`; then update `X = dfa[pat.charAt(j)][X]`.

X = simulation of B A B A

		0	1	2	3	4	5
pat.charAt(j)		A	B	A	B	A	C
dfa[][j]	A	1	1	3	1	5	1
	B	0	2	0	4	0	4
	C	0	0	0	0	0	6

Constructing the DFA for KMP substring search for A B A B A C



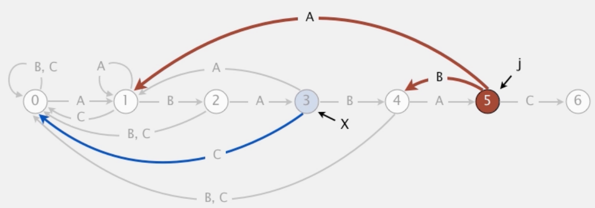
Knuth-Morris-Pratt demo: DFA construction in linear time

Mismatch transition. For each state `j` and char `c != pat.charAt(j)`, set `dfa[c][j] = dfa[c][X]`; then update `X = dfa[pat.charAt(j)][X]`.

X = simulation of B A B A

		0	1	2	3	4	5
pat.charAt(j)		A	B	A	B	A	C
dfa[][j]	A	1	1	3	1	5	1
	B	0	2	0	4	0	4
	C	0	0	0	0	0	6

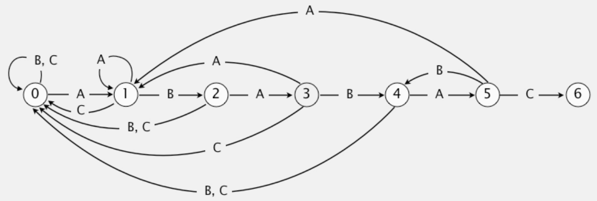
Constructing the DFA for KMP substring search for A B A B A C



Knuth-Morris-Pratt demo: DFA construction in linear time

		0	1	2	3	4	5
pat.charAt(j)		A	B	A	B	A	C
dfa[][j]	A	1	1	3	1	5	1
	B	0	2	0	4	0	4
	C	0	0	0	0	0	6

Constructing the DFA for KMP substring search for A B A B A C



37

Construção da DFA

Trecho de código do KMP que constrói a DFA.

```
dfa[pat.charAt(0)][0] = 1;
for (int j = 1, X = 0; j < m; j++) {
    for (int c = 0; c < R; c++)
        // copie casos de conflito
        dfa[c][j] = dfa[c][X];
    // defina casos de casamento
    dfa[pat.charAt(j)][j] = j+1;
    // atualize estado de reinício
    X = dfa[pat.charAt(j)][X];
}
```

Construção da DFA: programação dinâmica

dfa[c][j] = maior k tal que
pat[0..k-1] = pat[j-k+1..j-1]+c

Para j = 0:
dfa[c][0] = 1, se pat[0] = c
0, se pat[0] $\neq$ c

Para j > 0:
dfa[c][j] = dfa[c][j-1]+1, se pat[j] = c
dfa[c][X], se pat[j] $\neq$ c,
onde X é o maior valor tal que
pat[0..X]=pat[..j-1]+c.

Classe KMP: esqueleto

```
public class KMP {
   private final int R = 256;
   private String pat;
   // dfa[][] representa o autômato
   private int[][] dfa;
   public KMP(String pat) { ...}
   public int search(String txt) {...}
}
```

KMP: construtor

```
public KMP(String pat) {
    this.pat = pat;
    int m = pat.length();
    dfa = new int[R][m];
    dfa[pat.charAt(0)][0] = 1;
    for (int j = 1, X = 0; j < m; j++){
        // calcule dfa[][j]
        for (int c = 0; c < R; c++)
            dfa[c][j] = dfa[c][X];
        dfa[pat.charAt(j)][j] = j+1;
        X = dfa[pat.charAt(j)][X];
    }
}
```

KMP: search()

```
public int search(String txt) {
   inti, n = txt.length();
   int j, m = pat.length();

   for (i = 0, j = 0; i < n && j < m; i++)
       j = dfa[txt.charAt(i)][j];

   if (j == m) return i - m;
   returnn;
}
```

Consumo de tempo

O consumo de tempo do algoritmo KMP é $O(m + n)$.

Proposição. O algoritmo KMP examina não mais que m + n caracteres.

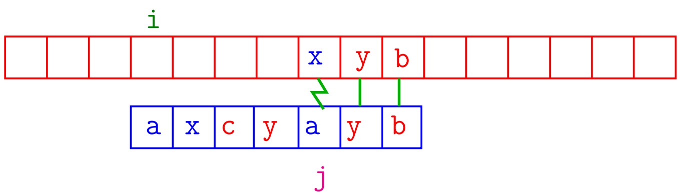
Se levarmos em conta o tamanho do alfabeto, R, o consumo de tempo para construir o DFA é mR.

Boyer-Moore

Fonte: ADS: Boyer Moore String Search

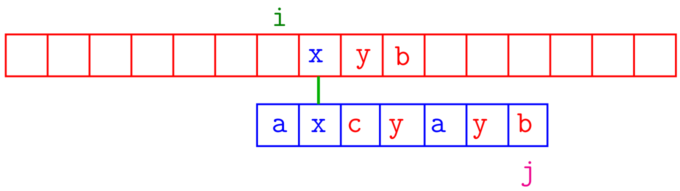
Primeiro algoritmo de Boyer-Moore

O primeiro algoritmo de R.S. Boyer e J.S. Moore (1977) é baseado na seguinte heurística.



Primeiro algoritmo de Boyer-Moore

O primeiro algoritmo de R.S. Boyer e J.S. Moore (1977) é baseado na seguinte heurística.



Boyer-Moore

pat = a n d a n d o

```
 0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30 31
 a s   a n d o r i n h a s   a n d a m   a n d a n d o   a l t o  txt
1a n d a n d o
```

Boyer-Moore

pat = a n d a n d o

```
 0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30 31
 a s   a n d o r i n h a s   a n d a m   a n d a n d o   a l t o  txt
1a n d a n d o
2      a n d a n d o
```

Boyer-Moore

pat = a n d a n d o

```
  0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30 31
  as andorinhas andam andando alto txt
1 andando
2    andando
3      andando
```

Boyer-Moore

pat = a n d a n d o

```
  0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30 31
  as andorinhas andam andando alto txt
1 andando
2    andando
3      andando
4         andando
```

Boyer-Moore

pat = a n d a n d o

```
  0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30 31
  as andorinhas andam andando alto txt
1 andando
2    andando
3      andando
4         andando
5            andando
```

Boyer-Moore

pat = a n d a n d o

```
  0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30 31
  as andorinhas andam andando alto txt
1 andando
2    andando
3      andando
4         andando
5            andando
6               andando
```

Boyer-Moore

pat = a n d a n d o

```
  0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30 31
  as andorinhas andam andando alto txt
1 andando
2    andando
3      andando
4         andando
5            andando
6               andando
7                  andando
```

Boyer-Moore

pat = a n d a n d o

```
  0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30 31
  as andorinhas andam andando alto txt
1 andando
2    andando
3      andando
4         andando
5            andando
6               andando
7                  andando
8                     andando
```

Boyer-Moore

pat = a b a b b a b a b b a

```
   0  1  2  3  4  5  6  7  8  9 10 11 12 13 14 15 16 17 18 19 20 21 22
   a  b  a  a  b  a  b  a  b  b  a  b  a  b  a  b  b  a  b  a  b  b  a  txt
1  a  b  a  b  b  a  b  a  b  b  a
```

Boyer-Moore

pat = a b a b b a b a b b a

```
   0  1  2  3  4  5  6  7  8  9 10 11 12 13 14 15 16 17 18 19 20 21 22
   a  b  a  a  b  a  b  a  b  b  a  b  a  b  a  b  b  a  b  a  b  b  a  txt
1  a  b  a  b  b  a  b  a  b  b  a
2     a  b  a  b  b  a  b  a  b  b  a
```

Boyer-Moore

pat = a b a b b a b a b b a

```
   0  1  2  3  4  5  6  7  8  9 10 11 12 13 14 15 16 17 18 19 20 21 22
   a  b  a  a  b  a  b  a  b  b  a  b  a  b  a  b  b  a  b  a  b  b  a  txt
1  a  b  a  b  b  a  b  a  b  b  a
2     a  b  a  b  b  a  b  a  b  b  a
3        a  b  a  b  b  a  b  a  b  b  a
```

Boyer-Moore

pat = a b a b b a b a b b a

```
   0  1  2  3  4  5  6  7  8  9 10 11 12 13 14 15 16 17 18 19 20 21 22
   a  b  a  a  b  a  b  a  b  b  a  b  a  b  a  b  b  a  b  a  b  b  a  txt
1  a  b  a  b  b  a  b  a  b  b  a
2     a  b  a  b  b  a  b  a  b  b  a
3        a  b  a  b  b  a  b  a  b  b  a
4           a  b  a  b  b  a  b  a  b  b  a
```

Boyer-Moore

pat = a b a b b a b a b b a

```
   0  1  2  3  4  5  6  7  8  9 10 11 12 13 14 15 16 17 18 19 20 21 22
   a  b  a  a  b  a  b  a  b  b  a  b  a  b  a  b  b  a  b  a  b  b  a  txt
1  a  b  a  b  b  a  b  a  b  b  a
2     a  b  a  b  b  a  b  a  b  b  a
3        a  b  a  b  b  a  b  a  b  b  a
4           a  b  a  b  b  a  b  a  b  b  a
5              a  b  a  b  b  a  b  a  b  b  a
```

Boyer-Moore

pat = a b a b b a b a b b a

```
   0  1  2  3  4  5  6  7  8  9 10 11 12 13 14 15 16 17 18 19 20 21 22
   a  b  a  a  b  a  b  a  b  b  a  b  a  b  a  b  b  a  b  a  b  b  a  txt
1  a  b  a  b  b  a  b  a  b  b  a
2     a  b  a  b  b  a  b  a  b  b  a
3        a  b  a  b  b  a  b  a  b  b  a
4           a  b  a  b  b  a  b  a  b  b  a
5              a  b  a  b  b  a  b  a  b  b  a
6                 a  b  a  b  b  a  b  a  b  b  a
```

Boyer-Moore

pat = a b a b b a b a b b a

```
     0  1  2  3  4  5  6  7  8  9 10 11 12 13 14 15 16 17 18 19 20 21 22
     a  b  a  a  b  a  b  a  b  b  a  b  a  b  a  b  b  a  b  a  b  b  a  txt
1    a  b  a  b  b  a  b  a  b  b  a
2       a  b  a  b  b  a  b  a  b  b  a
3          a  b  a  b  b  a  b  a  b  b  a
4             a  b  a  b  b  a  b  a  b  b  a
5                a  b  a  b  b  a  b  a  b  b  a
6                   a  b  a  b  b  a  b  a  b  b  a
7                      a  b  a  b  b  a  b  a  b  b  a
```

Boyer-Moore

pat = a b a b b a b a b b a

```
     0  1  2  3  4  5  6  7  8  9 10 11 12 13 14 15 16 17 18 19 20 21 22
     a  b  a  a  b  a  b  a  b  b  a  b  a  b  a  b  b  a  b  a  b  b  a  txt
1    a  b  a  b  b  a  b  a  b  b  a
2       a  b  a  b  b  a  b  a  b  b  a
3          a  b  a  b  b  a  b  a  b  b  a
4             a  b  a  b  b  a  b  a  b  b  a
5                a  b  a  b  b  a  b  a  b  b  a
6                   a  b  a  b  b  a  b  a  b  b  a
7                      a  b  a  b  b  a  b  a  b  b  a
8                         a  b  a  b  b  a  b  a  b  b  a
```

Boyer-Moore

pat = a b a b b a b a b b a

```
     0  1  2  3  4  5  6  7  8  9 10 11 12 13 14 15 16 17 18 19 20 21 22
     a  b  a  a  b  a  b  a  b  b  a  b  a  b  a  b  b  a  b  a  b  b  a  txt
1    a  b  a  b  b  a  b  a  b  b  a
2       a  b  a  b  b  a  b  a  b  b  a
3          a  b  a  b  b  a  b  a  b  b  a
4             a  b  a  b  b  a  b  a  b  b  a
5                a  b  a  b  b  a  b  a  b  b  a
6                   a  b  a  b  b  a  b  a  b  b  a
7                      a  b  a  b  b  a  b  a  b  b  a
8                         a  b  a  b  b  a  b  a  b  b  a
9                            a  b  a  b  b  a  b  a  b  b  a
```

Boyer-Moore

pat = a b a b b a b a b b a

```
     0  1  2  3  4  5  6  7  8  9 10 11 12 13 14 15 16 17 18 19 20 21 22
     a  b  a  a  b  a  b  a  b  b  a  b  a  b  a  b  b  a  b  a  b  b  a  txt
1    a  b  a  b  b  a  b  a  b  b  a
2       a  b  a  b  b  a  b  a  b  b  a
3          a  b  a  b  b  a  b  a  b  b  a
4             a  b  a  b  b  a  b  a  b  b  a
5                a  b  a  b  b  a  b  a  b  b  a
6                   a  b  a  b  b  a  b  a  b  b  a
7                      a  b  a  b  b  a  b  a  b  b  a
8                         a  b  a  b  b  a  b  a  b  b  a
9                            a  b  a  b  b  a  b  a  b  b  a
10                              a  b  a  b  b  a  b  a  b  b  a
```

Boyer-Moore

pat = a b a b b a b a b b a

```
     0  1  2  3  4  5  6  7  8  9 10 11 12 13 14 15 16 17 18 19 20 21 22
     a  b  a  a  b  a  b  a  b  b  a  b  a  b  a  b  b  a  b  a  b  b  a  txt
1    a  b  a  b  b  a  b  a  b  b  a
2       a  b  a  b  b  a  b  a  b  b  a
3          a  b  a  b  b  a  b  a  b  b  a
4             a  b  a  b  b  a  b  a  b  b  a
5                a  b  a  b  b  a  b  a  b  b  a
6                   a  b  a  b  b  a  b  a  b  b  a
7                      a  b  a  b  b  a  b  a  b  b  a
8                         a  b  a  b  b  a  b  a  b  b  a
9                            a  b  a  b  b  a  b  a  b  b  a
10                              a  b  a  b  b  a  b  a  b  b  a
11                                 a  b  a  b  b  a  b  a  b  b  a
```

Boyer-Moore

pat = a b a b b a b a b b a

```
     0  1  2  3  4  5  6  7  8  9 10 11 12 13 14 15 16 17 18 19 20 21 22
     a  b  a  a  b  a  b  a  b  b  a  b  a  b  a  b  b  a  b  a  b  b  a  txt
1    a  b  a  b  b  a  b  a  b  b  a
2       a  b  a  b  b  a  b  a  b  b  a
3          a  b  a  b  b  a  b  a  b  b  a
4             a  b  a  b  b  a  b  a  b  b  a
5                a  b  a  b  b  a  b  a  b  b  a
6                   a  b  a  b  b  a  b  a  b  b  a
7                      a  b  a  b  b  a  b  a  b  b  a
8                         a  b  a  b  b  a  b  a  b  b  a
9                            a  b  a  b  b  a  b  a  b  b  a
10                              a  b  a  b  b  a  b  a  b  b  a
11                                 a  b  a  b  b  a  b  a  b  b  a
12                                    a  b  a  b  b  a  b  a  b  b  a
```

Boyer-Moore

pat = a b a b b a b a b b a

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	
	a	b	a	a	b	a	b	a	b	b	a	b	a	b	a	b	b	a	b	a	b	b	a	txt
1	a	b	a	b	b	a	b	a	b	b	a													
2		a	b	a	b	b	a	b	a	b	b	a												
3			a	b	a	b	b	a	b	a	b	b	a											
4				a	b	a	b	b	a	b	a	b	b	a										
5					a	b	a	b	b	a	b	a	b	b	a									
6						a	b	a	b	b	a	b	a	b	b	a								
7							a	b	a	b	b	a	b	a	b	b	a							
8								a	b	a	b	b	a	b	a	b	b	a						
9									a	b	a	b	b	a	b	a	b	b	a					
10										a	b	a	b	b	a	b	a	b	b	a				
11											a	b	a	b	b	a	b	a	b	b	a			
12												a	b	a	b	b	a	b	a	b	b	a		
13													a	b	a	b	b	a	b	a	b	b	a	

Bad-character heuristic

Ideia ("*bad-character heuristic*"): calcular um deslocamento de modo que txt[j] fique emparelhado com a última ocorrência do caractere txt[j] em pat.

Suponha que o conjunto a que pertencem todos os elementos de pat e de txt é conhecido de antemão. Este conjunto é o **alfabeto** do problema.

Suponha que o alfabeto é o conjunto de todos os 256 caracteres.

Bad-character heuristic

Para implementar essa ideia fazemos um pré-processamento de pat, determinando para cada símbolo x do alfabeto a posição de sua última ocorrência em pat.

	0	1	2	3	4	5	6
pat	a	n	d	a	n	d	o

	0	...	'a'	'b'	'c'	'd'	...	...	'n'	'o'	'p'	...	255
right	−1	...	3	−1	−1	5	...	...	4	6	−1	...	...

Classe BoyerMoore: esqueleto

```
public class BoyerMoore {
   private final int R; // tam.  alfabeto
   // pulo bad-character
   private int[] right;
   // padrão como array ou String
   private char[] pattern;
   private String pat;
   public BoyerMoore(String pat) {...}
   public BoyerMoore(char[] pattern,
                     int R) {...}
   public int search(String txt) {...}
   public int search(char[] text) {...}
}
```

BoyerMoore: construtor

```
public BoyerMoore(String pat) {
   this.R = 256;
   this.pat = pat;
   // última ocorrência de c em pat
   right = new int[R];
   for (int c = 0; c < R; c++)
      right[c] = -1;
   for (int j = 0; j < pat.length(); j++)
      right[pat.charAt(j)] = j;
}
```

BoyerMoore: search()

Recebe strings pat e txt com $m \geq 1$ e $n \geq 0$, e retorna o índice da primeira ocorrências de pat em txt. Se pat não ocorre em txt, retorna n.

```
public int search(String txt) {
   int n = txt.length();
   int m = pat.length();
   int skip;
```

BoyerMoore: search()

```
for (int i = 0; i <= n-m; i += skip) {
    skip = 0;
    for (int j = m-1; j >= 0; j--) {
        if(pat.charAt(j)!=txt.charAt(i+j)){
           int r= right[txt.charAt(i+j)]
           skip = Math.max(1,j-r);
           break;
        }
    }
    if (skip == 0) return i; // achou
  }
  return n; // não achou
}
```

Pior caso

pat = b a a a a a a a a a a

```
    1  2  3  4  5  6  7  8  9 10 11 12 13 14 15 16 17 18 19 20 21 22 23
    a  a  a  a  a  a  a  a  a  a  a  a  a  a  a  a  a  a  a  a  a  a  a  txt
 1  b  a  a  a  a  a  a  a  a  a  a
 2     b  a  a  a  a  a  a  a  a  a  a
 3        b  a  a  a  a  a  a  a  a  a  a
 4           b  a  a  a  a  a  a  a  a  a  a
 5              b  a  a  a  a  a  a  a  a  a  a
 6                 b  a  a  a  a  a  a  a  a  a  a
 7                    b  a  a  a  a  a  a  a  a  a  a
 8                       b  a  a  a  a  a  a  a  a  a  a
 9                          b  a  a  a  a  a  a  a  a  a  a
10                             b  a  a  a  a  a  a  a  a  a  a
11                                b  a  a  a  a  a  a  a  a  a  a
12                                   b  a  a  a  a  a  a  a  a  a  a
13                                      a  b  a  a  a  a  a  a  a  a  a
```

Melhor caso

pat = a b c d e

```
    1  2  3  4  5  6  7  8  9 10 11 12 13 14 15 16 17 18 19 20 21 22 23
    ?  ?  ?  ?  x  ?  ?  ?  ?  x  ?  ?  ?  ?  x  ?  ?  ?  ?  x  ?  ?  ?  txt
 1  a  b  c  d  e
```

Melhor caso

pat = a b c d e

```
    1  2  3  4  5  6  7  8  9 10 11 12 13 14 15 16 17 18 19 20 21 22 23
    ?  ?  ?  ?  x  ?  ?  ?  ?  x  ?  ?  ?  ?  x  ?  ?  ?  ?  x  ?  ?  ?  txt
 1  a  b  c  d  e
 2                 a  b  c  d  e
```

Melhor caso

pat = a b c d e

```
    1  2  3  4  5  6  7  8  9 10 11 12 13 14 15 16 17 18 19 20 21 22 23
    ?  ?  ?  ?  x  ?  ?  ?  ?  x  ?  ?  ?  ?  x  ?  ?  ?  ?  x  ?  ?  ?  txt
 1  a  b  c  d  e
 2                 a  b  c  d  e
 3                                a  b  c  d  e
```

Melhor caso

pat = a b c d e

```
    1  2  3  4  5  6  7  8  9 10 11 12 13 14 15 16 17 18 19 20 21 22 23
    ?  ?  ?  ?  x  ?  ?  ?  ?  x  ?  ?  ?  ?  x  ?  ?  ?  ?  x  ?  ?  ?  txt
 1  a  b  c  d  e
 2                 a  b  c  d  e
 3                                a  b  c  d  e
 4                                               a  b  c  d  e
```

Melhor caso

$\mathtt{pat} = \mathtt{a\ b\ c\ d\ e}$

```
   1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23
   ? ? ? ? x ? ? ? ? x  ?  ?  ?  ?  x  ?  ?  ?  ?  x  ?  ?  ?  txt
1  a b c d e
2            a b c d e
3                     a  b  c  d  e
4                                    a  b  c  d  e
5                                                   a  b  c  ...
```

Conclusões

O consumo de tempo do algoritmo BoyerMoore no pior caso é $\mathrm{O}((\mathtt{n}-\mathtt{m}+1)\mathtt{m})$.

O consumo de tempo do algoritmo BoyerMoore no melhor caso é $\mathrm{O}(\mathtt{n}/\mathtt{m})$.

Isto significa que no pior caso o consumo de tempo é essencialmente proporcional a $\mathtt{mn}$ e no melhor caso o algoritmo é **sublinear**.

Bad-character heuristic: variante

O primeiro algoritmo de R.S. Boyer e J.S. Moore (1977) é baseado na seguinte heurística.

```
                              k
| | | | | | | | | | |x| | | | | |
         |x|a|x|y|a|y|b|
```

Bad-character heuristic: variante

O primeiro algoritmo de R.S. Boyer e J.S. Moore (1977) é baseado na seguinte heurística.

```
                                  k
| | | | | | | | | | |x| | | | | |
                        |x|a|x|y|a|y|b|
```

Bad-character heuristic: variante

$\mathtt{pat} = \mathtt{a\ n\ d\ a\ n\ d\ o}$

```
  0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30 31
  a s   a n d o r i n h a  s     a  n  d  a  m     a  n  d  a  n  d  o     a  l  t  o  txt
1 a n d a n d o
```

Bad-character heuristic: variante

$\mathtt{pat} = \mathtt{a\ n\ d\ a\ n\ d\ o}$

```
  0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30 31
  a s   a n d o r i n h a  s     a  n  d  a  m     a  n  d  a  n  d  o     a  l  t  o  txt
1 a n d a n d o
2               a n d a n  d  o
```

Bad-character heuristic: variante

pat = a n d a n d o

```
  0123456789...                  31
  as andorinhas andam andando alto   txt
1 andando
2         andando
3            andando
```

Bad-character heuristic: variante

pat = a n d a n d o

```
  0123456789...                  31
  as andorinhas andam andando alto   txt
1 andando
2         andando
3            andando
4                     andando
```

Bad-character heuristic: variante

pat = a n d a n d o

```
  0123456789...                  31
  as andorinhas andam andando alto   txt
1 andando
2         andando
3            andando
4                     andando
5                      andando
```

Bad-character heuristic: variante

pat = a n d a n d o

```
  0123456789...                  31
  as andorinhas andam andando alto   txt
1 andando
2         andando
3            andando
4                     andando
5                      andando
6                              anda...
```

Bad-character heuristic: variante

pat = a b a b b a b a b b a

```
  0 1 2 3 4 5 6 7 8 9 10 ...        22
  a b a a b a b a b b a b a b a b b a b a b b a   txt
1 a b a b b a b a b b a
```

Bad-character heuristic: variante

pat = a b a b b a b a b b a

```
  0 1 2 3 4 5 6 7 8 9 10 ...        22
  a b a a b a b a b b a b a b a b b a b a b b a   txt
1 a b a b b a b a b b a
2     a b a b b a b a b b a
```

Bad-character heuristic: variante

pat = a b a b b a b a b b a

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	
	a	b	a	a	b	a	b	a	b	b	a	b	a	b	a	b	b	a	b	a	b	b	a	txt
1	a	b	a	b	b	a	b	a	b	b	a													
2			a	b	a	b	b	a	b	a	b	b	a											
3					a	b	a	b	b	a	b	a	b	b	a									

Bad-character heuristic: variante

pat = a b a b b a b a b b a

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	
	a	b	a	a	b	a	b	a	b	b	a	b	a	b	a	b	b	a	b	a	b	b	a	txt
1	a	b	a	b	b	a	b	a	b	b	a													
2			a	b	a	b	b	a	b	a	b	b	a											
3					a	b	a	b	b	a	b	a	b	b	a									
4							a	b	a	b	b	a	b	a	b	b	a							

Bad-character heuristic: variante

pat = a b a b b a b a b b a

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	
	a	b	a	a	b	a	b	a	b	b	a	b	a	b	a	b	b	a	b	a	b	b	a	txt
1	a	b	a	b	b	a	b	a	b	b	a													
2			a	b	a	b	b	a	b	a	b	b	a											
3					a	b	a	b	b	a	b	a	b	b	a									
4							a	b	a	b	b	a	b	a	b	b	a							
5								a	b	a	b	b	a	b	a	b	b	a						

Bad-character heuristic: variante

pat = a b a b b a b a b b a

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	
	a	b	a	a	b	a	b	a	b	b	a	b	a	b	a	b	b	a	b	a	b	b	a	txt
1	a	b	a	b	b	a	b	a	b	b	a													
2			a	b	a	b	b	a	b	a	b	b	a											
3					a	b	a	b	b	a	b	a	b	b	a									
4							a	b	a	b	b	a	b	a	b	b	a							
5								a	b	a	b	b	a	b	a	b	b	a						
6										a	b	a	b	b	a	b	a	b	b	a				

Bad-character heuristic: variante

pat = a b a b b a b a b b a

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	
	a	b	a	a	b	a	b	a	b	b	a	b	a	b	a	b	b	a	b	a	b	b	a	txt
1	a	b	a	b	b	a	b	a	b	b	a													
2			a	b	a	b	b	a	b	a	b	b	a											
3					a	b	a	b	b	a	b	a	b	b	a									
4							a	b	a	b	b	a	b	a	b	b	a							
5								a	b	a	b	b	a	b	a	b	b	a						
6										a	b	a	b	b	a	b	a	b	b	a				
7												a	b	a	b	b	a	b	a	b	b	a		

Bad-character heuristic: variante

pat = a b a b b a b a b b a

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	
	a	b	a	a	b	a	b	a	b	b	a	b	a	b	a	b	b	a	b	a	b	b	a	txt
1	a	b	a	b	b	a	b	a	b	b	a													
2			a	b	a	b	b	a	b	a	b	b	a											
3					a	b	a	b	b	a	b	a	b	b	a									
4							a	b	a	b	b	a	b	a	b	b	a							
5								a	b	a	b	b	a	b	a	b	b	a						
6										a	b	a	b	b	a	b	a	b	b	a				
7												a	b	a	b	b	a	b	a	b	b	a		
8													a	b	a	b	b	a	b	a	b	b	a	

Segundo algoritmo de Boyer-Moore

O segundo algoritmo de R.S. Boyer e J.S. Moore (1977) é baseado na seguinte heurística.

i

							x	y	b						

a	x	y	b	a	y	b

j

Segundo algoritmo de Boyer-Moore

O segundo algoritmo de R.S. Boyer e J.S. Moore (1977) é baseado na seguinte heurística.

i

							x	y	b						

a	x	y	b	a	y	b

j

Good suffix heuristic

Não precisa conhecer o alfabeto explicitamente.

A implementação deve começar com um pré-processamento de pat: para cada j em $0, 1, \ldots, \mathtt{m}-1$ devemos calcular o maior k em $0, 1, \ldots, \mathtt{m}-2$ tal que:

- pat[j..m-1] é sufixo de pat[0..k] ou
- pat[0..k] é sufixo de pat[j..m-1]

Chamemos de bm[j] esse valor k.

Good suffix heuristic

Exemplo 1:

	0	1	2	3	4	5
pat	c	a	a	b	a	a
bm	−1	−1	−1	−1	2	4

Exemplo 2:

	0	1	2	3	4	5	6	7
pat	b	a	−	b	a	∗	b	a
bm	1	1	1	1	1	1	4	4

Good suffix heuristic

O vetor bm[] pode ser calculado facilmente em tempo $\mathrm{O}(\mathtt{m}^3)$.

Com mais trabalho, o vetor bm[] pode ser determinado em tempo $\mathrm{O}(\mathtt{m})$. Veja CLRS, seção 34.4.

Veja também a página do professor Paulo Feofiloff: Busca de palavras em um texto

Apêndice: Rabin-Karp

Fonte: ADS: Boyer Moore String Search

Referência: Algoritmo de Rabin-Karp para busca de substrings (PF)

Rabin-Karp

Criado por Richard M. Karp and Michael O. Rabin (1987).

O algoritmo também é conhecido como busca por impressão digital (*fingerprint search*).

Procura um segmento do texto que tenha o mesmo valor hash do padrão `pat`.

Usa hashing modular: módulo `Q`.

Se hash de `pat` é diferente do hash de todos os segmentos do texto então o padrão não ocorre no texto. A recíproca não vale: pode haver colisão.

Exemplo 1

```
    pat.charAt(j)
j   0  1  2  3  4
    2  6  5  3  5   % 997 = 613

                          txt.charAt(i)
i   0  1  2  3  4  5  6  7  8  9 10 11 12 13 14 15
    3  1  4  1  5  9  2  6  5  3  5  8  9  7  9  3
0   3  1  4  1  5  % 997 = 508
1      1  4  1  5  9  % 997 = 201
2         4  1  5  9  2  % 997 = 715
3            1  5  9  2  6  % 997 = 971
4               5  9  2  6  5  % 997 = 442
                                              match
5                  9  2  6  5  3  % 997 = 929
6 ← return i = 6      2  6  5  3  5  % 997 = 613
```

Basis for Rabin-Karp substring search

Exemplo 2

Procurar o padrão 12345 nos primeiros 100 mil dígitos da expansão decimal de π.

```
3141592653589793238462643383279502884197169399375105820974944592307
8164062862089986280348253421170679821480865132823066470938446095505
82231725359408128481117450284102701938521105559644622948954930381964
```

31415926535897932384626433832795028841971693993751058209749445923078
16406286208998628034825342117067982148086513282306647093844609550582
23172535940812848111745028410270193852110555964462294895493038196442
88109756659334461284756482337867831652712019091456485669234603486104
54326648213393607260249141273724587006606315588174881520920962829254
09171536436789259036001133053054882046652138414695194151160943305727
03657595919530921861173819326117931051185480744623799627495673518857
52724891227938183011949129833673362440656643086021394946395224737190
70217986094370277053921717629317675238467481846766940513200056812714
52635608277857713427577896091736371787214684409012249534301465495853
71050792279689258923542019956112129021960864034418159813629774771309
96051870721134999999837297804995105973173281609631859502445945534690
83026425223082533446850352619311881710100031378387528865875332083814
20617177669147303598253490428755468731159562863882353787593751957781
85778053217122680661300192787661119590921642019893809525720106548586
32788659361533818279682303019520353018529689957736225994138912497217
75283479131515574857242454150695950829533116861727855889075098381754
63746493931925506040092770167113900984882401285836160356370766010471
01819429555961989467678374494482553797747268471040475346462080466842
59069491293313677028989152104752162056966024058038150193511253382430
03558764024749647326391419927260426992279678235478163600934172164121
99245863150302861829745557067498385054945885869269956909272107975093
02955321165344987202755960236480665499119881834797753566369807426542
52786255181841757467289097777279380008164706001614524919217321721477
23501414419735685481613611573525521334757418494684385233239073941433
34547762416862518983569485562099219222184272550254256887671790494601
65346680498862723279178608578438382796797668145410095388378636095068
00642251252051173929848960841284886269456042419652850222106611863067

Algoritmo de Horner

Para calcular o valor de hash de um string usamos o algoritmo de Horner:

```java
private long hash(String key, int m) {
  long h = 0;
  for (int j = 0; j < m; j++)
     h = (h * R + key.charAt(j)) % Q;
  return h;
}
```

Exemplo

```
        pat.charAt(j)
i    0  1  2  3  4
     2  6  5  3  5
                        R              Q
0    2  % 997 = 2
1    2  6  % 997 = (2*10 + 6) % 997 = 26
2    2  6  5  % 997 = (26*10 + 5) % 997 = 265
3    2  6  5  3  % 997 = (265*10 + 3) % 997 = 659
4    2  6  5  3  5  % 997 = (659*10 + 5) % 997 = 613
```

Computing the hash value for the pattern with Horner's method

Ideia chave: hash de substrings consecutivos

Seja $t_i = txt[i]$ e x_i = o inteiro $t_i t_{i-1} \dots t_{i+m-1}$

Assim,

$$x_{i-1} = t_{i-1}R^{m-1} + t_i R^{m-2} + \cdots + t_{i+m-2}$$
$$x_i \quad = \qquad\qquad + t_i R^{m-1} + \cdots + t_{i+m-2}R + t_{i+m-1}$$

Logo, $x_i = (x_{i-1} - t_{i-1}R^{m-1})R + t_{i+m-1}$

Portanto, o valor $hash(x_i) = x_i \% Q$ pode ser obtido a partir do valor de $hash(x_{i-1}) = x_{i-1} \% Q$ em tempo constante.

$hash(x_i) = ((hash(x_i) - t_{i-1}R^{m-1})R + t_{i+m-1}) \% Q$

Exemplo

i	...	2	3	4	5	6	7	...	
current value	1	4	1	5	9	2	6	5	*text*
new value		4	1	5	9	2	6	5	

	4	1	5	9	2	*current value*
-	4	0	0	0	0	
		1	5	9	2	*subtract leading digit*
			*	1	0	*multiply by radix*
	1	5	9	2	0	
				+	6	*add new trailing digit*
	1	5	9	2	6	*new value*

Key computation in Rabin-Karp substring search

Implementação

Escolha Q igual a um primo grande para evitar a chance de colisão.

Evite *overflow* e números negativos:

- use um tipo-de-dados capaz de armazenar Q^2;
- na prática, escolha Q que caibe em um `int` ($2^{31} - 1$ é primo);
- faça as contas com `long`;
- tome o resto da divisão por Q depois de cada operação;
- some Q aos resultados intermediários quando necessário.

Exemplo: Q=997

$$
\begin{aligned}
(10000 + 535) \times 1000 &= (30 + 535) \times 3 \\
&= 565 \times 3 \\
&= 1695 \\
&= 698
\end{aligned}
$$

$$
\begin{aligned}
508 - 3 \times 10000 &= 508 - 3 \times (30) \\
&= 508 + 3 \times (-30) \\
&= 508 + 3 \times (997 - 30) \\
&= 508 + 3 \times 967 \\
&= 508 + 907 \\
&= 418
\end{aligned}
$$

Classe RabinKarp: esqueleto

```
public class RabinKarp {
  private String pat;
  private long patHash; // hash do padrão
  private int m;
  private long Q;
  private int R = 256;
  private long RM;
  public RabinKarp(String pat) {...}
  private long hash(String key, int m) {}
  private int search(String txt) {...}
  public boolean check(String txt, int i)
}
```

RabinKarp: construtor

```
public RabinKarp(String pat) {
  this.pat= pat;
  m = pat.length();
  Q = longRandomPrime();
  RM = 1;
  // calcula R^(m-1)%Q
  for (int i = 1; i <= m-1; i++)
     RM = (R * RM) % Q;
  patHash = hash(pat, m);
}
```

RabinKarp: search()

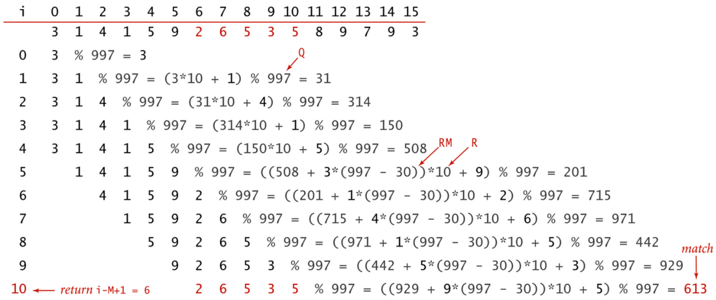
```
private int search(String txt) {
  int n = txt.length();
  long txtHash = hash(txt, m);
  if (patHash == txtHash
        && check(txt, 0)) return 0;
  for (int i= 1; i <= n- m; i++) {
     txtHash = (txtHash + Q -
           RM*txt.charAt(i-1) % Q) % Q;
     txtHash = (txtHash * R +
           txt.charAt(i+m-1)) % Q;
     if (patHash == txtHash
        && check(txt, i)) return i;
  }
  return n; } // não achou
```

RabinKarp: check()

```
// versão Las Vegas
private boolean check(String txt, int i){
   for (int j = 0; j < m; j++)
       if (pat.charAt(j)!=txt.charAt(i+j))
          return false;
       return true;
}

// versão Monte Carlo
private boolean check(String txt, int i){
   return true
}
```

Exemplo



Rabin-Karp substring search example

Monte Carlo *versus* Las Vegas

A versão Monte Carlo consome tempo linear, mas pode dar uma resposta errada, com baixíssima probabilidade.

A versão Las Vegas sempre dá a resposta certa, mas pode consumir tempo não linear, com baixíssima probabilidade.

Qual algoritmo é melhor

Força bruta é bom se o padrão e o texto não tiverem muitas auto-repetições.

KMP é rápido e tem a vantagem de nunca retroceder sobre o texto, o que é importante se o texto for dado como um fluxo contínuo (*streaming*).

BoyerMoore é provavelmente o mais rápido na prática.

RabinKarp é rápido mas pode dar resultados errados, com baixíssima probabilidade.

Implementações

Veja as implementações de busca de substrings:

- glibc: Implementation of strstr in glibc
- cpython: The stringlib Library
- Boyer-Moore-Horspool algorithm

Próximo passo

Que acontece se o padrão não é apenas uma string mas um conjunto de strings descrito por uma **expressão regular** como `A*|(A*BA*BA*)*` ou `((A*B|AC)D)`, por exemplo?

Essa generalização do problema de busca é muito importante. A solução envolve o conceito de autômato de estados não determinístico.