Algoritmo de KMP

Bruno Monteiro

Universidade Federal de Minas Gerais

27 de Maio de 2020

Introdução

- **String Matching**: dado um texto `t` com `m` caracteres e um padrão `s` com `n` caracteres, em quais posições de `t` `s` ocorre como substring?

```
t = aabaacaadaabaaba
s = aaba
```

- Matching nas posições 0, 9 e 12

Introdução

- **String Matching**: dado um texto `t` com `m` caracteres e um padrão `s` com `n` caracteres, em quais posições de `t` `s` ocorre como substring?

```
t = aabaacaadaabaaba
s = aaba
```

- Matching nas posições 0, 9 e 12

Introdução

- **String Matching**: dado um texto `t` com `m` caracteres e um padrão `s` com `n` caracteres, em quais posições de `t` `s` ocorre como substring?

```
t = aabaacaadaabaaba
s = aaba
```

- Matching nas posições 0, 9 e 12

Alguns conceitos

- **Prefixo** de uma string s é a string obtida após a remoção de 0 o mais caracteres do fim de s.

Alguns conceitos

- **Prefixo** de uma string s é a string obtida após a remoção de 0 o mais caracteres do fim de s.
  - “abc” e “abca” são prefixos de “abca”

Alguns conceitos

- **Prefixo** de uma string s é a string obtida após a remoção de 0 o mais caracteres do fim de s.
  - "abc" e "abca" são prefixos de "abca"
- **Sufixo** de uma string s é a string obtida após a remoção de 0 o mais caracteres do início de s.

Alguns conceitos

- **Prefixo** de uma string s é a string obtida após a remoção de 0 o mais caracteres do fim de s.
  - "abc" e "abca" são prefixos de "abca"
- **Sufixo** de uma string s é a string obtida após a remoção de 0 o mais caracteres do início de s.
  - "ca" e "abca" são sufixos de "abca"

Alguns conceitos

- **Prefixo** de uma string s é a string obtida após a remoção de 0 o mais caracteres do fim de s.
  - "abc" e "abca" são prefixos de "abca"
- **Sufixo** de uma string s é a string obtida após a remoção de 0 o mais caracteres do início de s.
  - "ca" e "abca" são sufixos de "abca"
- Prefixo/sufixo **próprio** de umas string s é um prefixo/sufixo diferente de s.

Alguns conceitos

- **Prefixo** de uma string s é a string obtida após a remoção de 0 o mais caracteres do fim de s.
  - "abc" e "abca" são prefixos de "abca"
- **Sufixo** de uma string s é a string obtida após a remoção de 0 o mais caracteres do início de s.
  - "ca" e "abca" são sufixos de "abca"
- Prefixo/sufixo **próprio** de umas string s é um prefixo/sufixo diferente de s.
- **Substring** de uma string s é uma string obtida após a remoção de 0 ou mais caracteres no inicio ou fim de s.

Alguns conceitos

- **Prefixo** de uma string s é a string obtida após a remoção de 0 o mais caracteres do fim de s.
  - "abc" e "abca" são prefixos de "abca"
- **Sufixo** de uma string s é a string obtida após a remoção de 0 o mais caracteres do início de s.
  - "ca" e "abca" são sufixos de "abca"
- Prefixo/sufixo **próprio** de umas string s é um prefixo/sufixo diferente de s.
- **Substring** de uma string s é uma string obtida após a remoção de 0 ou mais caracteres no inicio ou fim de s.
  - "bc", "c", "abc" e "abca" são substrings de "abca"

String Matching

- **String Matching**: dado um texto `t` com `m` caracteres e um padrão `s` com `n` caracteres, em quais posições de `t` `s` ocorre como substring?

String Matching

- **String Matching**: dado um texto `t` com `m` caracteres e um padrão `s` com `n` caracteres, em quais posições de `t` `s` ocorre como substring?
- Algoritmo *naive*: testa se tem um *matching* em cada posição do texto.

String Matching

- **String Matching**: dado um texto `t` com `m` caracteres e um padrão `s` com `n` caracteres, em quais posições de `t` `s` ocorre como substring?
- Algoritmo *naive*: testa se tem um *matching* em cada posição do texto.

```
t = aabaacaadaabaaba
s = aaba
```

String Matching

- **String Matching**: dado um texto `t` com `m` caracteres e um padrão `s` com `n` caracteres, em quais posições de `t` `s` ocorre como substring?
- Algoritmo *naive*: testa se tem um *matching* em cada posição do texto.

```
t = aabaacaadaabaaba
s = aaba
```

String Matching

- **String Matching**: dado um texto `t` com `m` caracteres e um padrão `s` com `n` caracteres, em quais posições de `t` `s` ocorre como substring?
- Algoritmo *naive*: testa se tem um *matching* em cada posição do texto.

```
t = aabaacaadaabaaba
s =  aaba
```

String Matching

- **String Matching**: dado um texto `t` com `m` caracteres e um padrão `s` com `n` caracteres, em quais posições de `t` `s` ocorre como substring?
- Algoritmo *naive*: testa se tem um *matching* em cada posição do texto.

```
t = aabaacaadaabaaba
s =  aaba
```

String Matching

- **String Matching**: dado um texto `t` com `m` caracteres e um padrão `s` com `n` caracteres, em quais posições de `t` `s` ocorre como substring?
- Algoritmo *naive*: testa se tem um *matching* em cada posição do texto.

```
t = aabaacaadaabaaba
s =   aaba
```

String Matching

- **String Matching**: dado um texto `t` com `m` caracteres e um padrão `s` com `n` caracteres, em quais posições de `t` `s` ocorre como substring?
- Algoritmo *naive*: testa se tem um *matching* em cada posição do texto.

```
t = aabaacaadaabaaba
s =    aaba
```

String Matching

- **String Matching**: dado um texto `t` com `m` caracteres e um padrão `s` com `n` caracteres, em quais posições de `t` `s` ocorre como substring?
- Algoritmo *naive*: testa se tem um *matching* em cada posição do texto.

```
t = aabaacaadaabaaba
s =   aaba
```

String Matching

- **String Matching**: dado um texto `t` com `m` caracteres e um padrão `s` com `n` caracteres, em quais posições de `t` `s` ocorre como substring?
- Algoritmo *naive*: testa se tem um *matching* em cada posição do texto.

```
t = aabaacaadaabaaba
s =     aaba
```

String Matching

- **String Matching**: dado um texto `t` com `m` caracteres e um padrão `s` com `n` caracteres, em quais posições de `t` `s` ocorre como substring?
- Algoritmo *naive*: testa se tem um *matching* em cada posição do texto.

```
t = aabaacaadaabaaba
s =     aaba
```

String Matching

- **String Matching**: dado um texto `t` com `m` caracteres e um padrão `s` com `n` caracteres, em quais posições de `t` `s` ocorre como substring?
- Algoritmo *naive*: testa se tem um *matching* em cada posição do texto.

```
t = aabaacaadaabaaba
s =       aaba
```

String Matching

- **String Matching**: dado um texto `t` com `m` caracteres e um padrão `s` com `n` caracteres, em quais posições de `t` `s` ocorre como substring?
- Algoritmo *naive*: testa se tem um *matching* em cada posição do texto.

```
t = aabaacaadaabaaba
s =        aaba
```

String Matching

- **String Matching**: dado um texto `t` com `m` caracteres e um padrão `s` com `n` caracteres, em quais posições de `t` `s` ocorre como substring?
- Algoritmo *naive*: testa se tem um *matching* em cada posição do texto.

```
t = aabaacaadaabaaba
s =       aaba
```

String Matching

- **String Matching**: dado um texto `t` com `m` caracteres e um padrão `s` com `n` caracteres, em quais posições de `t` `s` ocorre como substring?
- Algoritmo *naive*: testa se tem um *matching* em cada posição do texto.

```
t = aabaacaadaabaaba
s =         aaba
```

String Matching

- **String Matching**: dado um texto `t` com `m` caracteres e um padrão `s` com `n` caracteres, em quais posições de `t` `s` ocorre como substring?
- Algoritmo *naive*: testa se tem um *matching* em cada posição do texto.

```
t = aabaacaadaabaaba
s =         aaba
```

String Matching

- **String Matching**: dado um texto `t` com `m` caracteres e um padrão `s` com `n` caracteres, em quais posições de `t` `s` ocorre como substring?
- Algoritmo *naive*: testa se tem um *matching* em cada posição do texto.

```
t = aabaacaadaabaaba
s =          aaba
```

String Matching

- **String Matching**: dado um texto `t` com `m` caracteres e um padrão `s` com `n` caracteres, em quais posições de `t` `s` ocorre como substring?
- Algoritmo *naive*: testa se tem um *matching* em cada posição do texto.

```
t = aabaacaadaabaaba
s =           aaba
```

String Matching

- **String Matching**: dado um texto `t` com `m` caracteres e um padrão `s` com `n` caracteres, em quais posições de `t` `s` ocorre como substring?
- Algoritmo *naive*: testa se tem um *matching* em cada posição do texto.

```
t = aabaacaadaabaaba
s =           aaba
```

String Matching

- **String Matching**: dado um texto `t` com `m` caracteres e um padrão `s` com `n` caracteres, em quais posições de `t` `s` ocorre como substring?
- Algoritmo *naive*: testa se tem um *matching* em cada posição do texto.

```
t = aabaacaadaabaaba
s =           aaba
```

String Matching

- **String Matching**: dado um texto `t` com `m` caracteres e um padrão `s` com `n` caracteres, em quais posições de `t` `s` ocorre como substring?
- Algoritmo *naive*: testa se tem um *matching* em cada posição do texto.

```
t = aabaacaadaabaaba
s =           aaba
```

String Matching

- **String Matching**: dado um texto `t` com `m` caracteres e um padrão `s` com `n` caracteres, em quais posições de `t` `s` ocorre como substring?
- Algoritmo *naive*: testa se tem um *matching* em cada posição do texto.

```
t = aabaacaadaabaaba
s =             aaba
```

String Matching

- **String Matching**: dado um texto `t` com `m` caracteres e um padrão `s` com `n` caracteres, em quais posições de `t` `s` ocorre como substring?
- Algoritmo *naive*: testa se tem um *matching* em cada posição do texto.

```
t = aabaacaadaabaaba
s =             aaba
```

String Matching

- **String Matching**: dado um texto `t` com `m` caracteres e um padrão `s` com `n` caracteres, em quais posições de `t` `s` ocorre como substring?
- Algoritmo *naive*: testa se tem um *matching* em cada posição do texto.

```
t = aabaacaadaabaaba
s =              aaba
```

Algoritmo *naive*

- **String Matching**: dado um texto t de m caracteres e um padrão s de n caracteres, em quais posições t s ocorre como substring?
- Algoritmo *naive*: testa se tem um *matching* em cada posição do texto.
- O algoritmo *naive* tem complexidade $\mathcal{O}(nm)$ no pior caso.

Algoritmo *naive*

- **String Matching**: dado um texto t de m caracteres e um padrão s de n caracteres, em quais posições t s ocorre como substring?
- Algoritmo *naive*: testa se tem um *matching* em cada posição do texto.
- O algoritmo *naive* tem complexidade $\mathcal{O}(nm)$ no pior caso.
- Queremos fazer melhor que isso. Vamos analisar com cuidado o que está acontecendo.

Observação importante

```
t = abababababcd
s = abababcd
```

Observação importante

```
t = abababcd
s = ababcd
```

Observação importante

```
t = abababcd
s = ababab
      ababcd
```

Observação importante

```
t = abababababcd
s = abababcd
      abababcd
```

Observação importante

```
t = abababababcd
s = abababcd
      ababababcd
```

A função de prefixo

- A função de prefixo π nos fala, para cada prefixo $\overline{s}$ de s, o tamanho do maior prefixo **próprio** de $\overline{s}$ que também é sufixo de $\overline{s}$.

A função de prefixo

- A função de prefixo π nos fala, para cada prefixo $\overline{s}$ de s, o tamanho do maior prefixo **próprio** de $\overline{s}$ que também é sufixo de $\overline{s}$.

$$\pi[i] = \max_{0 \leq k \leq i} \{k : s[0 : k-1] = s[i-k+1 : i]\}$$

A função de prefixo

- A função de prefixo π nos fala, para cada prefixo $\overline{s}$ de s, o tamanho do maior prefixo **próprio** de $\overline{s}$ que também é sufixo de $\overline{s}$.

$$\pi[i] = \max_{0 \leq k \leq i} \{k : s[0 : k-1] = s[i-k+1 : i]\}$$

i	0	1	2	3	4	5	6
$\pi[i]$	0	0	0	1	2	3	0
s	a	b	c	a	b	c	d

A função de prefixo

- A função de prefixo π nos fala, para cada prefixo $\overline{s}$ de s, o tamanho do maior prefixo **próprio** de $\overline{s}$ que também é sufixo de $\overline{s}$.

$$\pi[i] = \max_{0 \leq k \leq i} \{k : s[0 : k-1] = s[i-k+1 : i]\}$$

i	0	1	2	3	4	5	6
$\pi[i]$	0	0	0	1	**2**	3	0
s	a	b	c	a	b	c	d

O algoritmo de KMP

i	0	1	2	3	4	5	6	7
$\pi[i]$	0	0	1	2	3	4	0	0
s	a	b	a	b	a	b	c	d

t = ababababcd

s = ababab cd

$i = 0\,;\, j = 0$

O algoritmo de KMP

i	0	1	2	3	4	5	6	7
$\pi[i]$	0	0	1	2	3	4	0	0
s	a	b	a	b	a	b	c	d

t = ababababcd

s = abababcd

$i = 6\,;\, j = 6$

O algoritmo de KMP

i	0	1	2	3	4	5	6	7
$\pi[i]$	0	0	1	2	3	**4**	0	0
s	a	b	a	b	a	b	c	d

t = ababababcd

s = abababcd

$i = 6\,;\, j = 6$

O algoritmo de KMP

i	0	1	2	3	4	5	6	7
$\pi[i]$	0	0	1	2	3	4	0	0
s	a	b	a	b	a	b	c	d

```
t = ababababcd
s =    ababacbcd
```

$i = 6\,;\, j = 4$

O algoritmo de KMP

i	0	1	2	3	4	5	6	7
$\pi[i]$	0	0	1	2	3	4	0	0
s	a	b	a	b	a	b	c	d

```
t = ababababcd_
s =   ababababcd_
```

$i = 10\,;\, j = 8$

O algoritmo de KMP

i	0	1	2	3	4	5
$\pi[i]$	0	0	1	0	1	2
s	a	b	a	c	a	b

```
t = abadaabaccabacabacab
s = abacab
```

$$i = 0\,;\, j = 0$$

O algoritmo de KMP

i	0	1	2	3	4	5
$\pi[i]$	0	0	1	0	1	2
s	a	b	a	c	a	b

t = abadaabaccabacabacab

s = abacab

$i = 1\,;\, j = 1$

O algoritmo de KMP

i	0	1	2	3	4	5
$\pi[i]$	0	0	1	0	1	2
s	a	b	a	c	a	b

t = abadaabaccabacabacab

s = abacab

$i = 2\,;\, j = 2$

O algoritmo de KMP

i	0	1	2	3	4	5
$\pi[i]$	0	0	1	0	1	2
s	a	b	a	c	a	b

```
t = abadaabaccabacabacab
s = abacab
```

$$i = 3\,;\, j = 3$$

O algoritmo de KMP

i	0	1	2	3	4	5
$\pi[i]$	0	0	**1**	0	1	2
s	a	b	a	c	a	b

```
t = abadaabaccabacabacab
s = abacab
```

$$i = 3\,;\, j = 3$$

O algoritmo de KMP

i	0	1	2	3	4	5
$\pi[i]$	0	0	1	0	1	2
s	a	b	a	c	a	b

```
t = abadaabaccabacabacab
s =   abacab
```

$$i = 3\,;\, j = 1$$

O algoritmo de KMP

i	0	1	2	3	4	5
$\pi[i]$	**0**	0	1	0	1	2
s	a	b	a	c	a	b

```
t = abadaabaccabacabacab
s =   abacab
```

$$i = 3\,;\, j = 1$$

O algoritmo de KMP

i	0	1	2	3	4	5
$\pi[i]$	0	0	1	0	1	2
s	a	b	a	c	a	b

```
t = abadaabaccabacabacab
s =    abacab
```

$$i = 3\,;\, j = 0$$

O algoritmo de KMP

i	0	1	2	3	4	5
$\pi[i]$	0	0	1	0	1	2
s	a	b	a	c	a	b

```
t = abadaabaccabacabacab
s =     abacab
```

$$i = 4\,;\, j = 0$$

O algoritmo de KMP

i	0	1	2	3	4	5
$\pi[i]$	0	0	1	0	1	2
s	a	b	a	c	a	b

```
t = abadaabaccabacabacab
s =     abacab
```

$$i = 5\,;\, j = 1$$

O algoritmo de KMP

i	0	1	2	3	4	5
$\pi[i]$	**0**	0	1	0	1	2
s	a	b	a	c	a	b

```
t = abadaabaccabacabacab
s =     abacab
```

$$i = 5\,;\, j = 1$$

O algoritmo de KMP

i	0	1	2	3	4	5
$\pi[i]$	0	0	1	0	1	2
s	a	b	a	c	a	b

```
t = abadaabaccabacabacab
s =     abacab
```

$$i = 5\,;\, j = 0$$

O algoritmo de KMP

i	0	1	2	3	4	5
$\pi[i]$	0	0	1	0	1	2
s	a	b	a	c	a	b

```
t = abadaabaccabacabacab
s =     abacab
```

$i = 6\,;\, j = 1$

O algoritmo de KMP

i	0	1	2	3	4	5
$\pi[i]$	0	0	1	0	1	2
s	a	b	a	c	a	b

```
t = abadaabaccabacabacab
s =     abacab
```

$i = 7\,;\, j = 2$

O algoritmo de KMP

i	0	1	2	3	4	5
$\pi[i]$	0	0	1	0	1	2
s	a	b	a	c	a	b

```
t = abadaabaccabacabacab
s =     abacab
```

$i = 8\,;\, j = 3$

O algoritmo de KMP

i	0	1	2	3	4	5
$\pi[i]$	0	0	1	0	1	2
s	a	b	a	c	a	b

```
t = abadaabaccabacabacab
s =      abacab
```

$i = 9\,;\, j = 4$

O algoritmo de KMP

i	0	1	2	3	4	5
$\pi[i]$	0	0	1	**0**	1	2
s	a	b	a	c	a	b

```
t = abadaabaccabacabacab
s =     abacab
```

$i = 9\,;\, j = 4$

O algoritmo de KMP

i	0	1	2	3	4	5
$\pi[i]$	0	0	1	0	1	2
s	a	b	a	c	a	b

```
t = abadaabaccabacabacab
s =          abacab
```

$i = 9\,;\, j = 0$

O algoritmo de KMP

i	0	1	2	3	4	5
$\pi[i]$	0	0	1	0	1	2
s	a	b	a	c	a	b

```
t = abadaabaccabacabacab
s =           abacab
```

$i = 10\,;\, j = 0$

O algoritmo de KMP

i	0	1	2	3	4	5
$\pi[i]$	0	0	1	0	1	2
s	a	b	a	c	a	b

```
t = abadaabaccabacabacab
s =          abacab
```

$i = 11\,;\, j = 1$

O algoritmo de KMP

i	0	1	2	3	4	5
$\pi[i]$	0	0	1	0	1	2
s	a	b	a	c	a	b

```
t = abadaabaccabacabacab
s =           abacab
```

$i = 12\,;\, j = 2$

O algoritmo de KMP

i	0	1	2	3	4	5
$\pi[i]$	0	0	1	0	1	2
s	a	b	a	c	a	b

```
t = abadaabaccabacabacab
s =             abacab
```

$i = 13\,;\, j = 3$

O algoritmo de KMP

i	0	1	2	3	4	5
$\pi[i]$	0	0	1	0	1	2
s	a	b	a	c	a	b

```
t = abadaabaccabacabacab
s =             abacab
```

$i = 14\,;\, j = 4$

O algoritmo de KMP

i	0	1	2	3	4	5
$\pi[i]$	0	0	1	0	1	2
s	a	b	a	c	a	b

```
t = abadaabaccabacabacab
s =             abacab
```

$i = 15\,;\, j = 5$

O algoritmo de KMP

i	0	1	2	3	4	5
$\pi[i]$	0	0	1	0	1	2
s	a	b	a	c	a	b

```
t = abadaabaccabacabacab
s =             abacab_
```

$i = 16\,;\, j = 6$

O algoritmo de KMP

i	0	1	2	3	4	5
$\pi[i]$	0	0	1	0	1	**2**
s	a	b	a	c	a	b

```
t = abadaabaccabacabacab
s =             abacab_
```

$i = 16\,;\, j = 6$

O algoritmo de KMP

i	0	1	2	3	4	5
$\pi[i]$	0	0	1	0	1	2
s	a	b	a	c	a	b

```
t = abadaabaccabacabacab
s =                 abacab
```

$i = 16\,;\, j = 2$

O algoritmo de KMP

i	0	1	2	3	4	5
$\pi[i]$	0	0	1	0	1	2
s	a	b	a	c	a	b

```
t = abadaabaccabacabacab
s =                 abacab
```

$i = 17\,;\, j = 3$

O algoritmo de KMP

i	0	1	2	3	4	5
$\pi[i]$	0	0	1	0	1	2
s	a	b	a	c	a	b

```
t = abadaabaccabacabacab
s =                 abacab
```

$i = 18\,;\, j = 4$

O algoritmo de KMP

i	0	1	2	3	4	5
$\pi[i]$	0	0	1	0	1	2
s	a	b	a	c	a	b

```
t = abadaabaccabacabacab
s =                 abacab
```

$i = 19\,;\, j = 5$

O algoritmo de KMP

i	0	1	2	3	4	5
$\pi[i]$	0	0	1	0	1	2
s	a	b	a	c	a	b

```
t = abadaabaccabacabacab_
s =                 abacab_
```

$i = 20\,;\, j = 6$

Implementação do algoritmo de KMP

```
1  vector<int> matching(string& t, string& s) {
2      vector<int> p = pi(s+'$'), match;
3      for (int i = 0, j = 0; i < t.size(); i++) {
4          while (j > 0 and s[j] != t[i]) j = p[j-1];
5          if (s[j] == t[i]) j++;
6          if (j == s.size()) match.push_back(i-j+1);
7      }
8      return match;
9  }
```

Implementação do algoritmo de KMP

```
1  vector<int> matching(string& t, string& s) {
2      vector<int> p = pi(s+'$'), match;
3      for (int i = 0, j = 0; i < t.size(); i++) {
4          while (j > 0 and s[j] != t[i]) j = p[j-1];
5          if (s[j] == t[i]) j++;
6          if (j == s.size()) match.push_back(i-j+1);
7      }
8      return match;
9  }
```

- Em cada iteração do *while*, o j diminui em pelo menos 1.

Implementação do algoritmo de KMP

```
1  vector<int> matching(string& t, string& s) {
2      vector<int> p = pi(s+'$'), match;
3      for (int i = 0, j = 0; i < t.size(); i++) {
4          while (j > 0 and s[j] != t[i]) j = p[j-1];
5          if (s[j] == t[i]) j++;
6          if (j == s.size()) match.push_back(i-j+1);
7      }
8      return match;
9  }
```

- Em cada iteração do *while*, o `j` diminui em pelo menos 1.
- O `j` só aumenta quando o `i` também aumenta.

Implementação do algoritmo de KMP

```
1  vector<int> matching(string& t, string& s) {
2      vector<int> p = pi(s+'$'), match;
3      for (int i = 0, j = 0; i < t.size(); i++) {
4          while (j > 0 and s[j] != t[i]) j = p[j-1];
5          if (s[j] == t[i]) j++;
6          if (j == s.size()) match.push_back(i-j+1);
7      }
8      return match;
9  }
```

- Em cada iteração do *while*, o `j` diminui em pelo menos 1.
- O `j` só aumenta quando o `i` também aumenta.
- Portanto, o *while* executa no máximo $m = |t|$ iterações.

Implementação do algoritmo de KMP

```
1  vector<int> matching(string& t, string& s) {
2      vector<int> p = pi(s+'$'), match;
3      for (int i = 0, j = 0; i < t.size(); i++) {
4          while (j > 0 and s[j] != t[i]) j = p[j-1];
5          if (s[j] == t[i]) j++;
6          if (j == s.size()) match.push_back(i-j+1);
7      }
8      return match;
9  }
```

- Em cada iteração do *while*, o `j` diminui em pelo menos 1.
- O `j` só aumenta quando o `i` também aumenta.
- Portanto, o *while* executa no máximo $m = |t|$ iterações.
- A complexidade do algoritmo é a complexidade de construir a função de prefixo $\pi + \mathcal{O}(m)$.

Como computar a função π

i	0	1	2	3	4	5	6
$\pi[i]$	0	0	1	0	1	2	3
s	a	b	a	c	a	b	a

Como computar a função π

i	0	1	2	3	4	5	6
$\pi[i]$	0	0	1	0	1	2	**3**
s	a	b	a	c	a	b	a

Como computar a função π

i	0	1	2	3	4	5	6
$\pi[i]$	0	0	1	0	1	**2**	3
s	a	b	a	c	a	b	a

Como computar a função π

i	0	1	2	3	4	5	6
$\pi[i]$	0	0	1	0	**1**	2	3
s	a	b	a	c	a	b	a

Como computar a função π

i	0	1	2	3	4	5	6	7
$\pi[i]$	0	0	1	0	1	2	3	
s	a	b	a	c	a	b	a	b

Como computar a função π

i	0	1	2	3	4	5	6	7
$\pi[i]$	0	0	1	0	1	2	**3**	
s	a	b	a	c	a	b	a	b

Como computar a função π

i	0	1	2	3	4	5	6	7
$\pi[i]$	0	0	**1**	0	1	2	3	
s	a	b	a	c	a	b	a	b

Como computar a função π

i	0	1	2	3	4	5	6	7
$\pi[i]$	0	0	1	0	1	2	3	**2**
s	a	b	a	c	a	b	a	b

Como computar a função π

i	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
$\pi[i]$	0															
s	a	b	c	a	b	a	b	c	a	b	c	a	a	b	a	c

Como computar a função π

i	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
$\pi[i]$	**0**															
s	a	b	c	a	b	a	b	c	a	b	c	a	a	b	a	c

Como computar a função π

i	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
$\pi[i]$	0	0														
s	a	b	c	a	b	a	b	c	a	b	c	a	a	b	a	c

Como computar a função π

i	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
$\pi[i]$	0	**0**														
s	a	b	c	a	b	a	b	c	a	b	c	a	a	b	a	c

Como computar a função π

i	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
$\pi[i]$	0	0	0													
s	a	b	c	a	b	a	b	c	a	b	c	a	a	b	a	c

Como computar a função π

i	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
$\pi[i]$	0	0	**0**													
s	a	b	c	a	b	a	b	c	a	b	c	a	a	b	a	c

Como computar a função π

i	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
$\pi[i]$	0	0	0	1												
s	a	b	c	a	b	a	b	c	a	b	c	a	a	b	a	c

Como computar a função π

i	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
$\pi[i]$	0	0	0	**1**												
s	a	b	c	a	b	a	b	c	a	b	c	a	a	b	a	c

Como computar a função π

i	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
$\pi[i]$	0	0	0	1	2											
s	a	b	c	a	b	a	b	c	a	b	c	a	a	b	a	c

Como computar a função π

i	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
$\pi[i]$	0	0	0	1	**2**											
s	a	b	c	a	b	a	b	c	a	b	c	a	a	b	a	c

Como computar a função π

i	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
$\pi[i]$	0	**0**	0	1	2											
s	a	b	c	a	b	a	b	c	a	b	c	a	a	b	a	c

Como computar a função π

i	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
$\pi[i]$	0	0	0	1	2	1										
s	a	b	c	a	b	a	b	c	a	b	c	a	a	b	a	c

Como computar a função π

i	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
$\pi[i]$	0	0	0	1	2	**1**										
s	a	b	c	a	b	a	b	c	a	b	c	a	a	b	a	c

Como computar a função π

i	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
$\pi[i]$	0	0	0	1	2	1	2									
s	a	b	c	a	b	a	b	c	a	b	c	a	a	b	a	c

Como computar a função π

i	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
$\pi[i]$	0	0	0	1	2	1	**2**									
s	a	b	c	a	b	a	b	c	a	b	c	a	a	b	a	c

Como computar a função π

i	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
$\pi[i]$	0	0	0	1	2	1	2	3								
s	a	b	c	a	b	a	b	c	a	b	c	a	a	b	a	c

Como computar a função π

i	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
$\pi[i]$	0	0	0	1	2	1	2	**3**								
s	a	b	c	a	b	a	b	c	a	b	c	a	a	b	a	c

Como computar a função π

i	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
$\pi[i]$	0	0	0	1	2	1	2	3	4							
s	a	b	c	a	b	a	b	c	a	b	c	a	a	b	a	c

Como computar a função π

i	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
$\pi[i]$	0	0	0	1	2	1	2	3	**4**							
s	a	b	c	a	b	a	b	c	a	b	c	a	a	b	a	c

Como computar a função π

i	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
$\pi[i]$	0	0	0	1	2	1	2	3	4	5						
s	a	b	c	a	b	a	b	c	a	b	c	a	a	b	a	c

Como computar a função π

i	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
$\pi[i]$	0	0	0	1	2	1	2	3	4	**5**						
s	a	b	c	a	b	a	b	c	a	b	c	a	a	b	a	c

Como computar a função π

i	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
$\pi[i]$	0	0	0	1	**2**	1	2	3	4	5						
s	a	b	c	a	b	a	b	c	a	b	c	a	a	b	a	c

Como computar a função π

i	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
$\pi[i]$	0	0	0	1	2	1	2	3	4	5	3					
s	a	b	c	a	b	a	b	c	a	b	c	a	a	b	a	c

Como computar a função π

i	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
$\pi[i]$	0	0	0	1	2	1	2	3	4	5	**3**					
s	a	b	c	a	b	a	b	c	a	b	c	a	a	b	a	c

Como computar a função π

i	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
$\pi[i]$	0	0	0	1	2	1	2	3	4	5	3	4				
s	a	b	c	a	b	a	b	c	a	b	c	a	a	b	a	c

Como computar a função π

i	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
$\pi[i]$	0	0	0	1	2	1	2	3	4	5	3	**4**				
s	a	b	c	a	b	a	b	c	a	b	c	a	a	b	a	c

Como computar a função π

i	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
$\pi[i]$	0	0	0	**1**	2	1	2	3	4	5	3	4				
s	a	b	c	a	b	a	b	c	a	b	c	a	a	b	a	c

Como computar a função π

i	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
$\pi[i]$	**0**	0	0	1	2	1	2	3	4	5	3	4				
s	a	b	c	a	b	a	b	c	a	b	c	a	a	b	a	c

Como computar a função π

i	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
$\pi[i]$	0	0	0	1	2	1	2	3	4	5	3	4	1			
s	a	b	c	a	b	a	b	c	a	b	c	a	a	b	a	c

Como computar a função π

i	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
$\pi[i]$	0	0	0	1	2	1	2	3	4	5	3	4	**1**			
s	a	b	c	a	b	a	b	c	a	b	c	a	a	b	a	c

Como computar a função π

i	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
$\pi[i]$	0	0	0	1	2	1	2	3	4	5	3	4	1	2		
s	a	b	c	a	b	a	b	c	a	b	c	a	a	b	a	c

Como computar a função π

i	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
$\pi[i]$	0	0	0	1	2	1	2	3	4	5	3	4	1	**2**		
s	a	b	c	a	b	a	b	c	a	b	c	a	a	b	a	c

Como computar a função π

i	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
$\pi[i]$	0	**0**	0	1	2	1	2	3	4	5	3	4	1	2		
s	a	b	c	a	b	a	b	c	a	b	c	a	a	b	a	c

Como computar a função π

i	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
$\pi[i]$	0	0	0	1	2	1	2	3	4	5	3	4	1	2	1	
s	a	b	c	a	b	a	b	c	a	b	c	a	a	b	a	c

Como computar a função π

i	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
$\pi[i]$	0	0	0	1	2	1	2	3	4	5	3	4	1	2	**1**	
s	a	b	c	a	b	a	b	c	a	b	c	a	a	b	a	c

Como computar a função π

i	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
$\pi[i]$	**0**	0	0	1	2	1	2	3	4	5	3	4	1	2	1	
s	a	b	c	a	b	a	b	c	a	b	c	a	a	b	a	c

Como computar a função π

i	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
$\pi[i]$	0	0	0	1	2	1	2	3	4	5	3	4	1	2	1	0
s	a	b	c	a	b	a	b	c	a	b	c	a	a	b	a	c

Implementação do algoritmo para computar a função π

```
1  vector<int> pi(string s) {
2      vector<int> p(s.size());
3      for (int i = 1, j = 0; i < s.size(); i++) {
4          while (j > 0 and s[j] != s[i]) j = p[j-1];
5          if (s[j] == s[i]) j++;
6          p[i] = j;
7      }
8      return p;
9  }
```

Implementação do algoritmo para computar a função π

```
1  vector<int> pi(string s) {
2      vector<int> p(s.size());
3      for (int i = 1, j = 0; i < s.size(); i++) {
4          while (j > 0 and s[j] != s[i]) j = p[j-1];
5          if (s[j] == s[i]) j++;
6          p[i] = j;
7      }
8      return p;
9  }
```

- Em cada iteração do *while*, o j diminui em pelo menos 1.

Implementação do algoritmo para computar a função π

```
1  vector<int> pi(string s) {
2      vector<int> p(s.size());
3      for (int i = 1, j = 0; i < s.size(); i++) {
4          while (j > 0 and s[j] != s[i]) j = p[j-1];
5          if (s[j] == s[i]) j++;
6          p[i] = j;
7      }
8      return p;
9  }
```

- Em cada iteração do *while*, o `j` diminui em pelo menos 1.
- O `j` só aumenta quando o `i` também aumenta.

Implementação do algoritmo para computar a função π

```
1  vector<int> pi(string s) {
2      vector<int> p(s.size());
3      for (int i = 1, j = 0; i < s.size(); i++) {
4          while (j > 0 and s[j] != s[i]) j = p[j-1];
5          if (s[j] == s[i]) j++;
6          p[i] = j;
7      }
8      return p;
9  }
```

- Em cada iteração do *while*, o `j` diminui em pelo menos 1.
- O `j` só aumenta quando o `i` também aumenta.
- Portanto, o *while* executa no máximo $n = |s|$ iterações.

Implementação do algoritmo para computar a função π

```
1  vector<int> pi(string s) {
2      vector<int> p(s.size());
3      for (int i = 1, j = 0; i < s.size(); i++) {
4          while (j > 0 and s[j] != s[i]) j = p[j-1];
5          if (s[j] == s[i]) j++;
6          p[i] = j;
7      }
8      return p;
9  }
```

- Em cada iteração do *while*, o `j` diminui em pelo menos 1.
- O `j` só aumenta quando o `i` também aumenta.
- Portanto, o *while* executa no máximo $n = |s|$ iterações.
- A complexidade do algoritmo é $\mathcal{O}(n)$.

Implementação do algoritmo para computar a função π

```
1  vector<int> pi(string s) {
2      vector<int> p(s.size());
3      for (int i = 1, j = 0; i < s.size(); i++) {
4          while (j > 0 and s[j] != s[i]) j = p[j-1];
5          if (s[j] == s[i]) j++;
6          p[i] = j;
7      }
8      return p;
9  }
```

- Em cada iteração do *while*, o `j` diminui em pelo menos 1.
- O `j` só aumenta quando o `i` também aumenta.
- Portanto, o *while* executa no máximo $n = |\,s\,|$ iterações.
- A complexidade do algoritmo é $\mathcal{O}(n)$.
- A complexidade do algoritmo de KMP é $\mathcal{O}(n + m)$.

Revisando o KMP

```
1  vector<int> matching(string& t, string& s) {
2      vector<int> p = pi(s+'$'), match;
3      for (int i = 0, j = 0; i < t.size(); i++) {
4          while (j > 0 and s[j] != t[i]) j = p[j-1];
5          if (s[j] == t[i]) j++;
6          if (j == s.size()) match.push_back(i-j+1);
7      }
8      return match;
9  }
```

Revisando o KMP

```
1  vector<int> matching(string& t, string& s) {
2      vector<int> p = pi(s+'$'), match;
3      for (int i = 0, j = 0; i < t.size(); i++) {
4          while (j > 0 and s[j] != t[i]) j = p[j-1];
5          if (s[j] == t[i]) j++;
6          if (j == s.size()) match.push_back(i-j+1);
7      }
8      return match;
9  }
```

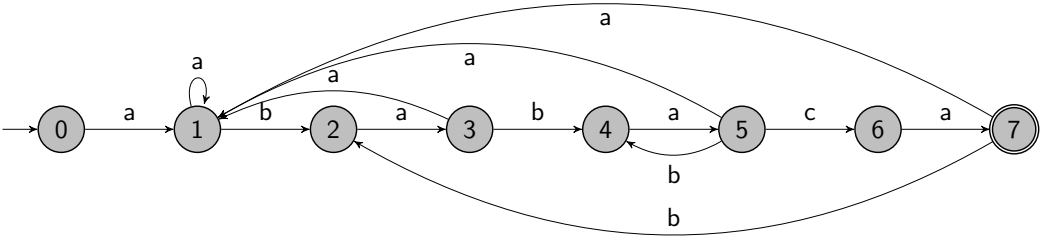
- Em cada iteração do `for`, olhamos a próxima letra do texto e atualizamos o `j`.

Revisando o KMP

```
1  vector<int> matching(string& t, string& s) {
2      vector<int> p = pi(s+'$'), match;
3      for (int i = 0, j = 0; i < t.size(); i++) {
4          while (j > 0 and s[j] != t[i]) j = p[j-1];
5          if (s[j] == t[i]) j++;
6          if (j == s.size()) match.push_back(i-j+1);
7      }
8      return match;
9  }
```

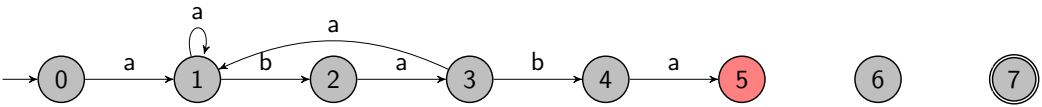
- Em cada iteração do `for`, olhamos a próxima letra do texto e atualizamos o `j`.
- Será que é possível construir alguma estrutura que nos permita atualizar o `j` em $\mathcal{O}(1)$?

Autômato finito



i	0	1	2	3	4	5	6
$\pi[i]$	0	0	1	2	3	0	1
s	a	b	a	b	a	c	a

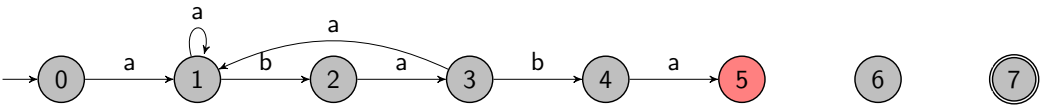
Autômato finito



i	0	1	2	3	4	5	6
$\pi[i]$	0	0	1	2	3	0	1
s	a	b	a	b	a	c	a

...ababa_
ababaca

Autômato finito

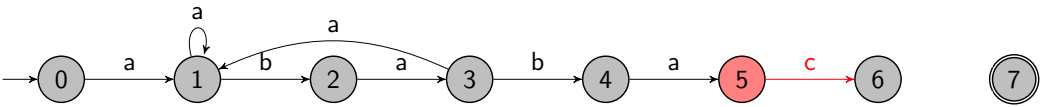


i	0	1	2	3	4	5	6
$\pi[i]$	0	0	1	2	3	0	1
s	a	b	a	b	a	c	a

...ababac

ababaca

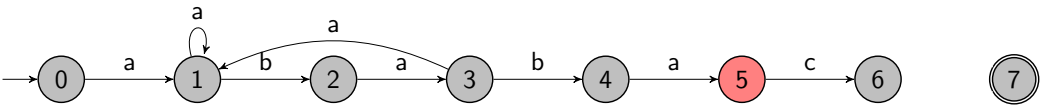
Autômato finito



i	0	1	2	3	4	5	6
$\pi[i]$	0	0	1	2	3	0	1
s	a	b	a	b	a	c	a

...ababac_
ababaca

Autômato finito

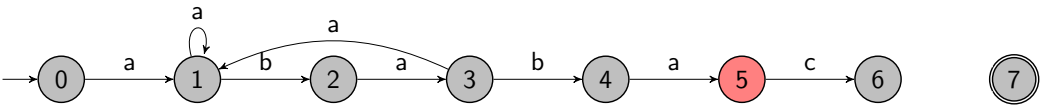


i	0	1	2	3	4	5	6
$\pi[i]$	0	0	1	2	3	0	1
s	a	b	a	b	a	c	a

...ababab

ababaca

Autômato finito

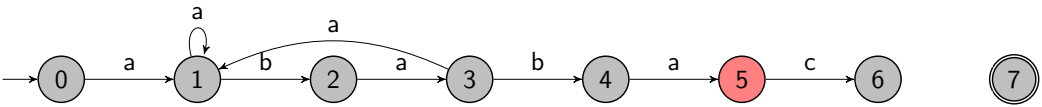


i	0	1	2	3	4	5	6
$\pi[i]$	0	0	1	2	**3**	0	1
s	a	b	a	b	a	c	a

...ababab

ababaca

Autômato finito

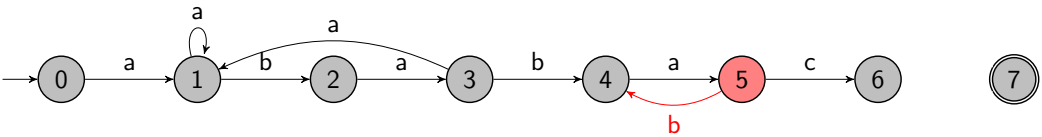


i	0	1	2	3	4	5	6
$\pi[i]$	0	0	1	2	3	0	1
s	a	b	a	b	a	c	a

...abab

ababaca

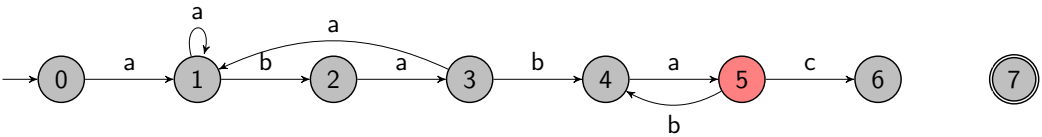
Autômato finito



i	0	1	2	3	4	5	6
$\pi[i]$	0	0	1	2	3	0	1
s	a	b	a	b	a	c	a

...abab_
ababaca

Autômato finito

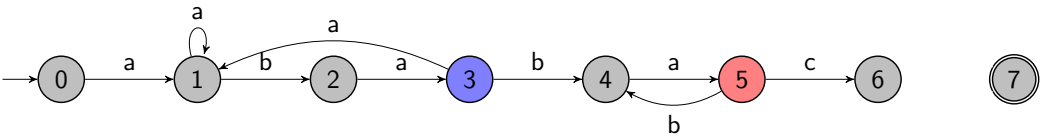


i	0	1	2	3	4	5	6
$\pi[i]$	0	0	1	2	3	0	1
s	a	b	a	b	a	c	a

...ababa<u>a</u>

ababa<u>c</u>a

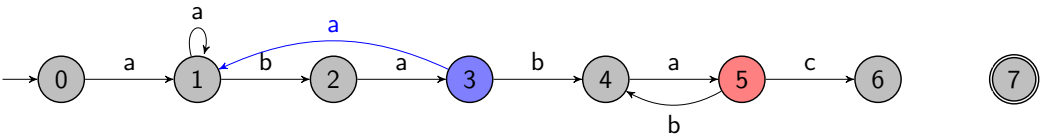
Autômato finito



i	0	1	2	3	4	5	6
$\pi[i]$	0	0	1	2	**3**	0	1
s	a	b	a	b	a	c	a

...ababa<u>a</u>
ababa<u>c</u>a

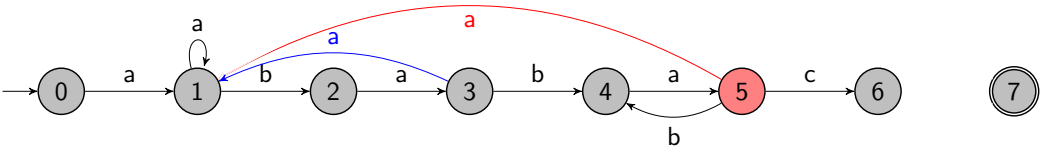
Autômato finito



i	0	1	2	3	4	5	6
$\pi[i]$	0	0	1	2	**3**	0	1
s	a	b	a	b	a	c	a

...ababa<u>a</u>
ababa<u>c</u>a

Autômato finito



i	0	1	2	3	4	5	6
$\pi[i]$	0	0	1	2	**3**	0	1
s	a	b	a	b	a	c	a

...a_
ababaca

Construção do autômato do KMP

```
1  struct autKMP {
2      vector<vector<int>> nxt;
3
4      autKMP(string& s) : nxt(26, vector<int>(s.size()+1)) {
5          vector<int> p = pi(s);
6          nxt[s[0]-'a'][0] = 1;
7          for (char c = 0; c < 26; c++)
8              for (int i = 1; i <= s.size(); i++)
9                  nxt[c][i] = c == s[i]-'a' ? i+1 : nxt[c][p[i-1]];
10     }
11 };
```

Construção do autômato do KMP

```
1  struct autKMP {
2      vector<vector<int>> nxt;
3
4      autKMP(string& s) : nxt(26, vector<int>(s.size()+1)) {
5          vector<int> p = pi(s);
6          nxt[s[0]-'a'][0] = 1;
7          for (char c = 0; c < 26; c++)
8              for (int i = 1; i <= s.size(); i++)
9                  nxt[c][i] = c == s[i]-'a' ? i+1 : nxt[c][p[i-1]];
10     }
11 };
```

- A construção é feita em $\mathcal{O}(n|\Sigma|)$ (linear no tamanho do autômato).

Outras formas de resolver string matching

- Para resolver string matching, podemos construir o autômato do padrão e passar o texto pelo autômato.

Outras formas de resolver string matching

- Para resolver string matching, podemos construir o autômato do padrão e passar o texto pelo autômato.

```
1     vector<int> matching(string& t, string& s) {
2         auto aut = autKMP(s);
3         vector<int> match;
4         int at = 0;
5         for (int i = 0; i < t.size(); i++) {
6             at = aut.nxt[t[i]-'a'][at];
7             if (at == s.size()) match.push_back(i-at+1);
8         }
9         return match;
10    }
```

Outras formas de resolver string matching

- Outra forma é concatenar o padrão com o texto (com um separador entre eles). Olhando para a função de prefixo em cada posição, podemos detectar os matchings.

Outras formas de resolver string matching

- Outra forma é concatenar o padrão com o texto (com um separador entre eles). Olhando para a função de prefixo em cada posição, podemos detectar os matchings.

abc$cbabca

Outras formas de resolver string matching

- Outra forma é concatenar o padrão com o texto (com um separador entre eles). Olhando para a função de prefixo em cada posição, podemos detectar os matchings.

abc$cbabca

```
vector<int> matching(string& t, string& s) {
    vector<int> p = pi(s+'$'+t), match;
    for (int i = s.size()+1; i < p.size(); i++)
        if (p[i] == s.size()) match.push_back(i - 2*s.size());
    return match;
}
```

Outras aplicações da função de prefixo

- Dado uma string s tal que $|s| \leq 10^3$, quantas substrings distintas s possui?

Outras aplicações da função de prefixo

- Dado uma string s tal que $|s| \leq 10^3$, quantas substrings distintas s possui?
- Resolvemos o problema iterativamente (para cada sufixo).

Outras aplicações da função de prefixo

- Dado uma string s tal que $|s| \leq 10^3$, quantas substrings distintas s possui?
- Resolvemos o problema iterativamente (para cada sufixo).

abacaba

Outras aplicações da função de prefixo

- Dado uma string s tal que $|s| \leq 10^3$, quantas substrings distintas s possui?
- Resolvemos o problema iterativamente (para cada sufixo).

abacaba

Outras aplicações da função de prefixo

- Dado uma string s tal que $|s| \leq 10^3$, quantas substrings distintas s possui?
- Resolvemos o problema iterativamente (para cada sufixo).

abacaba

Outras aplicações da função de prefixo

- Dado uma string s tal que $|s| \leq 10^3$, quantas substrings distintas s possui?
- Resolvemos o problema iterativamente (para cada sufixo).

abacaba

Outras aplicações da função de prefixo

- Dado uma string s tal que $|s| \leq 10^3$, quantas substrings distintas s possui?
- Resolvemos o problema iterativamente (para cada sufixo).

abacaba

i	0	1	2	3	4	5	6
$\pi[i]$	0	0	1	0	1	2	**3**
s	a	b	a	c	a	b	a

Outras aplicações da função de prefixo

- Dado `s` ($|s| \leq 10^6$), qual o tamanho da menor string `t` tal que `s = tttt..tt`?

Outras aplicações da função de prefixo

- Dado s ($|s| \leq 10^6$), qual o tamanho da menor string t tal que s = tttt..tt?

s = abababababab

Outras aplicações da função de prefixo

- Dado s ($|s| \leq 10^6$), qual o tamanho da menor string t tal que s = tttt..tt?

```
s = abababababab
t = ab
```

Outras aplicações da função de prefixo

- Dado `s` ($|s| \leq 10^6$), qual o tamanho da menor string `t` tal que `s = tttt..tt`?

```
s = abababababab
t = ab
```

- Seja `k = |s| - π[|s|-1]`. Se `k` divide `|s|`, então a resposta é `k`. Caso contrário, a resposta é `|s|`.

Outras aplicações da função de prefixo

- Seja `k = |s| - π[|s|-1]`. Se `k` divide `|s|`, então a resposta é `k`. Caso contrário, a resposta é `|s|`.

Outras aplicações da função de prefixo

- Seja `k = |s| - π[|s|-1]`. Se `k` divide `|s|`, então a resposta é `k`. Caso contrário, a resposta é `|s|`.
- Se `k` divide `|s|`:

Outras aplicações da função de prefixo

- Seja `k = |s| - π[|s|-1]`. Se `k` divide `|s|`, então a resposta é `k`. Caso contrário, a resposta é `|s|`.
- Se `k` divide `|s|`:

```
s = ab________
```

Outras aplicações da função de prefixo

- Seja `k = |s| - π[|s|-1]`. Se `k` divide `|s|`, então a resposta é `k`. Caso contrário, a resposta é `|s|`.
- Se `k` divide `|s|`:

```
s = abab______
```

Outras aplicações da função de prefixo

- Seja `k = |s| - π[|s|-1]`. Se `k` divide `|s|`, então a resposta é `k`. Caso contrário, a resposta é `|s|`.
- Se `k` divide `|s|`:

```
s = abababa____
```

Outras aplicações da função de prefixo

- Seja `k = |s| - π[|s|-1]`. Se `k` divide `|s|`, então a resposta é `k`. Caso contrário, a resposta é `|s|`.
- Se `k` divide `|s|`:

```
s = abababab__
```

Outras aplicações da função de prefixo

- Seja `k = |s| - π[|s|-1]`. Se `k` divide `|s|`, então a resposta é `k`. Caso contrário, a resposta é `|s|`.
- Se `k` divide `|s|`:

```
s = abababababab
```

Outras aplicações da função de prefixo

- Seja `k = |s| - π[|s|-1]`. Se `k` divide `|s|`, então a resposta é `k`. Caso contrário, a resposta é `|s|`.

Outras aplicações da função de prefixo

- Seja `k = |s| - π[|s|-1]`. Se `k` divide `|s|`, então a resposta é `k`. Caso contrário, a resposta é `|s|`.
- Se `k` não divide `|s|`:

Outras aplicações da função de prefixo

- Seja `k = |s| - π[|s|-1]`. Se `k` divide `|s|`, então a resposta é `k`. Caso contrário, a resposta é `|s|`.
- Se `k` não divide `|s|`:

```
s = abcabcabca
```

Outras aplicações da função de prefixo

- Seja `k = |s| - π[|s|-1]`. Se `k` divide `|s|`, então a resposta é `k`. Caso contrário, a resposta é `|s|`.
- Se `k` não divide `|s|`:

```
s = abcabcabca
```

- π`[|s|-1] = 7`.

Outras aplicações da função de prefixo

- Seja `k = |s| - π[|s|-1]`. Se `k` divide `|s|`, então a resposta é `k`. Caso contrário, a resposta é `|s|`.
- Se `k` não divide `|s|`:

`s = abcabcabca`

- `π[|s|-1] = 7`.

Outras aplicações da função de prefixo

- Seja `k = |s| - π[|s|-1]`. Se `k` divide `|s|`, então a resposta é `k`. Caso contrário, a resposta é `|s|`.
- Se `k` não divide `|s|`:

`s = abcabcabca`

- `π[|s|-1] = 7`.
- Por que não pode existir resposta de tamanho 2?

Outras aplicações da função de prefixo

- Seja `k = |s| - π[|s|-1]`. Se `k` divide `|s|`, então a resposta é `k`. Caso contrário, a resposta é `|s|`.
- Se `k` não divide `|s|`:

`s = abcabcabca`

- `π[|s|-1] = 7`.
- Por que não pode existir resposta de tamanho 2?

`s = ab________`

Outras aplicações da função de prefixo

- Seja `k = |s| - π[|s|-1]`. Se `k` divide `|s|`, então a resposta é `k`. Caso contrário, a resposta é `|s|`.
- Se `k` não divide `|s|`:

`s = abcabcabca`

- `π[|s|-1] = 7`.
- Por que não pode existir resposta de tamanho 2?

`s = abab______`

Outras aplicações da função de prefixo

- Seja `k = |s| - π[|s|-1]`. Se `k` divide `|s|`, então a resposta é `k`. Caso contrário, a resposta é `|s|`.
- Se `k` não divide `|s|`:

`s = abcabcabca`

- `π[|s|-1] = 7`.
- Por que não pode existir resposta de tamanho 2?

`s = aaaa______`

Outras aplicações da função de prefixo

- Seja `k = |s| - π[|s|-1]`. Se `k` divide `|s|`, então a resposta é `k`. Caso contrário, a resposta é `|s|`.
- Se `k` não divide `|s|`:

`s = abcabcabca`

- `π[|s|-1] = 7`.
- Por que não pode existir resposta de tamanho 2?

`s = aaaaaaaaaa`

Links

- Implementações.
- Alguns problemas: Pattern Find, Mysterious Code, Awesome Brother, String Compression.