

# Strings

## *Suffix Array* – Aplicações

---

Prof. Edson Alves - UnB/FGA

2019

1. Busca em *array* de sufixos
2. Comparação de substrings de mesmo tamanho
3. Maior prefixo comum entre duas substrings

## Busca em *array* de sufixos

---

## Busca em *array* de sufixos

- O problema de se determinar se a string  $P$ , de tamanho  $M$ , é ou não uma substring de  $S$ , de tamanho  $N$ , pode ser resolvido por meio de um *array* de sufixos
- Isto porque, se  $P$  é uma substring de  $S$ , ela será substring de algum dos prefixos  $S[i..N]$  de  $S$
- Assim, para localizar  $P$  em  $S$  basta fazer uma busca binária em  $s_A(S)$
- Em cada etapa da busca binária, a comparação de  $T$  com o prefixo em questão tem complexidade  $O(M)$
- Assim o algoritmo tem complexidade  $O(M \log N)$ , se  $s_A(S)$  já estiver construído
- O número de ocorrências de  $P$  pode ser determinado por meio de uma segunda busca binária, pois todas elas estarão adjacentes no *array* de sufixos

# Implementação da busca em *array* de sufixos em C++

```
84 int occurrences(const string& P, const string& S)
85 {
86     auto sa = suffix_array(S);
87
88     auto it = lower_bound(sa.begin(), sa.end(), P,
89         // retorna true S[sa[i]..N] de anteceder P na ordenação
90         [&](int i, const string& P) {
91             return S.compare(i, P.size(), P) < 0;
92         });
93
94     auto jt = upper_bound(sa.begin(), sa.end(), P,
95         // retorna true se P deve anteceder S[sa[i]..N] na ordenação
96         [&](const string& P, int i) {
97             return S.compare(i, P.size(), P) > 0;
98         });
99
100     return jt - it;
101 }
```

## **Comparação de substrings de mesmo tamanho**

---

## Comparação de strings de mesmo tamanho

- Seja  $S$  uma string de tamanho  $N$  e considere duas substrings de  $S$ :  
 $a = S[i..(i + M - 1)]$  e  $b = S[j..(j + M - 1)]$ , ambas de tamanho  $M$
- Uma função  $f(a, b)$  é uma função de comparação de strings se  
 $f(a, b) < 0$  se  $a < b$ ,  $f(a, b) = 0$  se  $a = b$  e  $f(a, b) > 0$  se  $a > b$
- Usando a comparação caractere a caractere, uma função de comparação pode ser implementada com complexidade  $O(M)$
- Contudo, uma vez construído o vetor de sufixos  $s_A(S)$ , esta função pode ser implementada em  $O(1)$
- Para tal, é preciso armazenar os valores das classes de equivalência  $cs$  obtidos em todas as iterações da construção de  $s_A(S)$
- Seja  $cs[k][i]$  a classe de equivalência da substring cíclica de  $S$ , de tamanho  $2^k$ , com início na posição  $i$

## Comparação de strings de mesmo tamanho

- Se  $M = 2^k$ , então a função  $f(a, b)$  corresponde à comparação direta entre  $cs[k][i]$  e  $cs[k][j]$
- Caso contrário,  $M$  pode ser decomposto em dois blocos de tamanho  $2^t$ , onde  $2^t \leq M$  e  $2^{t+1} > M$
- O primeiro bloco começa na posição inicial da substring ( $i$ , no caso da substring  $a$ )
- O segundo bloco começa  $2^t$  posições antes da última posição (no caso da substring  $a$ , na posição  $i + M - 2^t$ )
- Se as classes de ambas strings em relação ao primeiro bloco são distintas, a comparação entre eles é suficiente
- Caso contrário, basta finalizar a comparação utilizando as classes de equivalência dos respectivos segundos blocos
- Assim, o algoritmo tem complexidade  $O(N \log N)$ , por conta da construção de  $s_A(S)$ , e complexidade de memória  $O(N \log N)$ , por conta da tabela de classes de equivalência  $cs$





## Visualização da comparação entre duas substrings de mesmo tamanho

$a$  = programação

$b$  = programados

## Visualização da comparação entre duas substrings de mesmo tamanho

$a =$    $b =$  

The diagram illustrates the comparison of two substrings of equal length. String  $a$  is "programação" and string  $b$  is "programados". Both strings have their first 10 characters, "programa", highlighted in green. A bracket above the green part of  $a$  and a bracket below the green part of  $b$  are both labeled "1º bloco", indicating the first block of comparison.

## Visualização da comparação entre duas substrings de mesmo tamanho

$a$  = programação

$b$  = programados

# Implementação da comparação de substrings de mesmo tamanho em $O(1)$

```
88 int compare(int i, int j, int M, const vector<vector<int>>& cs)
89 {
90     int k = 0;
91
92     while ((1 << (k + 1)) <= M)
93         ++k;
94
95     auto a = ii(cs[k][i], cs[k][i + M - (1 << k)]);
96     auto b = ii(cs[k][j], cs[k][j + M - (1 << k)]);
97
98     return a == b ? 0 : (a < b ? -1 : 1);
99 }
```

## **Maior prefixo comum entre duas substrings**

---

## Maior prefixo comum entre duas substrings

- O vetor de sufixos de  $s$  pode ser utilizado para computar o maior prefixo comum (*longest common prefix – LCP*) entre duas substrings de  $s$
- A ideia central é calcular o maior prefixo comum entre os pares de sufixos adjacentes em  $s_A(s)$
- Defina  $lcp(i)$  como o tamanho do maior prefixo comum entre os sufixos  $s_A[i]$  e  $s_A[i + 1]$ , com  $i = 1, 2, \dots, N - 1$
- Assim, o maior prefixo comum  $LPC(i, j)$  entre os sufixos  $s_A[i]$  e  $s_A[j]$  é dado por

$$LPC(i, j) = \min \{ lpc(i + 1), lpc(i + 2), \dots, lpc(j) \}$$

- Desde modo, o problema  $LPC(i, j)$  é reduzido a um problema de *range minimun query – RMQ*, o qual pode ser resolvido por meio de uma árvores de segmentos, por exemplo

# Algoritmo de Kasai

- Uma vez computado o vetor de sufixos  $s_A(s)$  de uma string  $s$  de tamanho  $N$ , o algoritmo de Kasai permite computar os valores  $lpc(i)$  em  $O(N)$
- Considere dois sufixos consecutivos no vetor de sufixos, que iniciem nas posições  $i$  e  $j$  da string  $s$ , cujo maior prefixo comum entre eles tenha tamanho  $k > 0$
- Se removidos os primeiros caracteres de cada um destes sufixos, serão obtidos os sufixos  $i + 1$  e  $j + 1$ , os quais não são, necessariamente, consecutivos no vetor de sufixos
- Contudo, estes novos sufixos tem, no mínimo,  $k - 1$  caracteres comuns entre seus prefixos
- Assim,  $k - 1$  dentre as comparações feitas podem ser reaproveitadas para computar o próximo valor de  $lcp$
- Um caso especial a ser considerado é o prefixo que ocupa a última posição do vetor de sufixos, que não terá um prefixo que o sucede

# Algoritmo de Kasai

- Para determinar em qual posição inicia o  $t$ -ésimo sufixo do vetor de sufixos, é utilizado um vetor auxiliar  $rank[t]$ , que indica tal posição
- A variável  $k$  que registrará o tamanho do maior prefixo comum deve iniciar com o valor zero
- No caso especial onde  $rank[t] = N$ , este valor  $k$  deve ser reiniciado para o valor zero
- Nos demais casos, para cada valor de  $i = 1, 2, \dots, N$ , o índice  $j$  onde inicia o prefixo que o sucede  $S[i..N]$  no vetor de sufixos será dado por  $j = s_A[rank[i] + 1]$
- O valor de  $k$  deve ser incrementando enquanto  $S[i + k]$  for igual a  $S[j + k]$ , respeitados os limites da string
- Assim,  $lcp(rank[i]) = k$ , e o valor  $k$  deve ser decrementado para o próximo índice



$s = \text{banana}$

$s_A =$	6	4	2	1	5	3
$rank =$	4	3	6	2	5	1
$lcp =$	0	0	0	0	0	0

# Visualização da construção do vetor *lcp*

$s = \text{banana}$

$i$  ↓  
 $j$  ↑

$$k = 0, \quad i = 1, \quad j = 5$$

$s_A =$	6	4	2	1	5	3
$rank =$	4	3	6	2	5	1
$lcp =$	0	0	0	<b>0</b>	0	0

# Visualização da construção do vetor *lcp*

$s = \text{banana}$

$i \downarrow$

$\uparrow j$

$$k = 0, \quad i = 2, \quad j = 1$$

$s_A =$	6	4	2	1	5	3
$rank =$	4	3	6	2	5	1
$lcp =$	0	0	0	0	0	0

# Visualização da construção do vetor $lcp$

$s = \overset{i}{\downarrow} \text{banana}$

$$k = 0, \quad i = 3$$

$s_A =$	6	4	2	1	5	3
$rank =$	4	3	6	2	5	1
$lcp =$	0	0	0	0	0	0

# Visualização da construção do vetor *lcp*

$s = \text{banana}$

$i \downarrow$

$\uparrow j$

$$k = 3, \quad i = 4, \quad j = 2$$

$s_A =$	6	4	2	1	5	3
$rank =$	4	3	6	2	5	1
$lcp =$	0	<b>3</b>	0	0	0	0

# Visualização da construção do vetor $lcp$

$s = \text{banana}$

$i \downarrow$

$\uparrow j$

$$k = 2, \quad i = 5, \quad j = 3$$

$s_A =$	6	4	2	1	5	3
$rank =$	4	3	6	2	5	1
$lcp =$	0	3	0	0	2	0

# Visualização da construção do vetor $lcp$

$s = \text{banana}$

$i \downarrow$

$\uparrow j$

$$k = 1, \quad i = 6, \quad j = 4$$

$s_A =$	6	4	2	1	5	3
$rank =$	4	3	6	2	5	1
$lcp =$	1	3	0	0	2	0

## Implementação da construção do vetor *lpc*

```
84 vector<int> build_lcp(const string& S)
85 {
86     auto sa = suffix_array(S);
87     int N = S.size(), k = 0;
88
89     vector<int> rank(N, 0);
90
91     for (int i = 0; i < N; ++i)
92         rank[sa[i]] = i;
93
94     vector<int> lcp(N - 1, 0);
95
96     for (int i = 0; i < N; ++i)
97     {
98         if (rank[i] == N - 1)
99         {
100             k = 0;
101             continue;
102         }
103
104         auto j = sa[rank[i] + 1];
```



# Implementação da construção do vetor $lpc$

```
105
106     while (i + k < N and j + k < N and S[i + k] == S[j + k])
107         ++k;
108
109     lcp[rank[i]] = k;
110
111     if (k)
112         --k;
113 }
114
115 return lcp;
116 }
```

## Número de substrings distintas

- Uma string  $S$  de tamanho  $N$  tem  $N(N + 1)/2$  substrings no total
- Isto porque a substring  $S[i..j]$  de  $S$  pode ser caracterizada pelo par de índices  $(i, j)$ , com  $1 \leq i \leq j \leq N$
- O início de cada substring coincide com o início de algum sufixo de  $S$
- Deste modo, se  $b = S[i..j] = S[r..s]$ , isto significa que  $b$  é prefixo comum entre os sufixos  $S[i..N]$  e  $S[j..N]$
- Logo, todos os índices relacionados os maiores prefixos comuns entre dois prefixos consecutivos no vetor de prefixos geram substrings duplicadas
- Assim, se removidas as duplicatas do total de strings, resta o número de substrings distintas  $D(S)$  de  $S$
- Logo,

$$D(S) = \frac{N(N + 1)}{2} - \sum_{i=1}^{N-1} lcp[i]$$

# Implementação da contagem de substrings distintas usando $lcp$

```
118 long long distinct_substrings(const string& S)
119 {
120     auto lcp = build_lcp(S);
121     long long N = S.size();
122     long long ans = N*(N + 1)/2;
123
124     for (auto x : lcp)
125         ans -= x;
126
127     return ans;
128 }
```

1. CP Algorithms. [Suffix Array](#), acesso em 06/09/2019.
2. **CROCHEMORE**, Maxime; **RYTTER**, Wojciech. *Jewels of Stringology: Text Algorithms*, WSPC, 2002.
3. **HALIM**, Steve; **HALIM**, Felix. *Competitive Programming 3*, Lulu, 2013.