Departamento de Computação

Programação Competitiva unesp

Strings: KMP e String Hashing

Laboratório de Programação Competitiva I

Pedro Henrique Paiola

Rene Pegoraro

Wilson M Yonezawa

Arissa Yoshida

Nicolas Barbosa Gomes

Luis Henrique Morelli

Strings em Programação Competitiva

- Existem diversos problemas clássicos associados a Strings. Nesta aula trataremos sobre dois problemas específicos:
  - Busca em Strings / String *matching*
  - Substrings palindrômicas

Busca em strings

- O problema *substring search*/*pattern search*/*string matching* consiste em encontrar uma dada string dentro de outra.

- Exemplo:

```
S = “Que a Força esteja com você”
P = “Força”
```

Busca em strings

- O problema *substring search* ou *pattern search* consiste em encontrar uma dada string dentro de outra.

- Exemplo:

```
S = “Que a Força esteja com você”
P = “Força”
```

Ocorrências: 6 (posição)

Busca em strings

- O problema *substring search* ou *pattern search* consiste em encontrar uma dada string dentro de outra.

- Exemplo:

  S = “aabaacaadaabaaba”

  P = “aaba”

  Ocorrências: 0, 9 e 12

Busca em strings

- O problema *substring search* ou *pattern search* consiste em encontrar uma dada string dentro de outra.

- Exemplo:

  S = "aabaacaadaabaaba"

  P = "aaba"

  Ocorrências: 0, 9 e 12

Busca em strings

- O problema *substring search* ou *pattern search* consiste em encontrar uma dada string dentro de outra.

- Exemplo:

  S = “aabaacaadaabaaba”

  P = “aaba”

  Ocorrências: 0, 9 e 12

Busca em strings

- Algoritmo ingênuo

```
int search(string S, string P) {
    int i, j;
    for(i = 0; i <= S.size() - P.size(); i++) {
        for(j = 0; j < P.size(); j++)
            if (S[i+j] != P[j])
                break;
        if (j == P.size())
            return i;
    }
    return -1;
}
```

Busca em strings

- Esse algoritmo, no pior caso, tem complexidade $O(m.n)$, fazendo $m.n$ comparações. Porém, em geral, ele não chega a realizar tantas comparações.
- Usar esse algoritmo é bastante razoável para vários casos, principalmente quando as strings não são muito grandes.
- Mas, existem algoritmos de busca de substrings mais eficientes, que podem ser necessários em algumas situações, como exemplo temos o KMP.

Alguns conceitos

- **Prefixo** de uma string S é a string obtida após a remoção de 0 ou mais caracteres do fim de S.
  - “a”, “adc”, “adcbaa” são prefixos de “adcbaa”
- **Sufixo** de uma string S é a string obtida após a remoção de 0 ou mais carateres do início de S.
  - “a”, “baa”, “adcbaa” são sufixos de “adcbaa”
- **Prefixo/sufixo próprio** de S é um prefixo/sufixo de S que é diferente de S.
- **Substring** de uma string S é uma string obtida após a remoção de 0 ou mais caracteres no início ou no fim de S.
  - “a”, “cba”, “adc”, “dcba”, “adcbaa” são substrings de “adcbaa”

KMP

- Knuth Morrit Pratt
- Complexidade: $O(n + m)$ no pior caso

- No algoritmo ingênuo, sempre que detectamos caracteres diferentes, avançávamos um caracter na string principal (i++) e testamos toda a substring, desde o começo (começando sempre com j = 0).
- O KMP, porém, aproveita as comparações que foram feitas antes de encontrar dois caracteres diferentes, evitando comparar novamente caracteres que já sabemos que são compatíveis.

KMP

- A principal ideia deste algoritmo é pré-processar o padrão P, de modo a obter um vetor de inteiros lps, que conta o número de caracteres que podem ser "ignorados" em uma nova comparação.
- O nome lps refere-se à "*longest proper prefix and suffix*", ou seja, o maior prefixo próprio (não pode ser a própria palavra) que também é sufixo.
  - Conhecido também como função de prefixo.

KMP

P = “ABABAC”
lps = {}

P = “ABABAC”
lps = {0}

P = “ABABAC”
lps = {0, 0}

P = “ABABAC”
lps = {0, 0, 1}

P = “ABABAC”
lps = {0, 0, 1, 2}

P = “ABABAC”
lps = {0, 0, 1, 2, 3}

P = “ABABAC”
lps = {0, 0, 1, 2, 3, 0}

KMP

- E como isto ajuda? Isso permite pular comparações desnecessárias, por exemplo:
- Pelo algoritmo ingênuo:

S = ABABABCABABABCABABABC

P = ABABAC

KMP

- E como isto ajuda? Isso permite pular comparações desnecessárias, por exemplo:
- Pelo algoritmo ingênuo:

S = ABABABCABABABCABABABC

P =   ABABAC

KMP

- E como isto ajuda? Isso permite pular comparações desnecessárias, por exemplo:
- Pelo algoritmo ingênuo:

S = ABABABCABABABCABABABC

P =   ABABAC

KMP

- E como isto ajuda? Isso permite pular comparações desnecessárias, por exemplo:
- Pelo algoritmo ingênuo:

S = ABABABCABABABCABABABC

P = ABABAC

KMP

- E como isto ajuda? Isso permite pular comparações desnecessárias, por exemplo:
- Pelo algoritmo ingênuo:

S = ABABABCABABABCABABABC

P = ABABAC

KMP

- E como isto ajuda? Isso permite pular comparações desnecessárias, por exemplo:
- Pelo algoritmo ingênuo:

S = ABABABCABABABCABABABC

P = ABABAC

KMP

- E como isto ajuda? Isso permite pular comparações desnecessárias, por exemplo:
- Pelo algoritmo ingênuo:

S = ABABABCABABABCABABABC

P = ABABAC

KMP

- E como isto ajuda? Isso permite pular comparações desnecessárias, por exemplo:
- Pelo KMP:

```
S = ABABABCABABABCABABABC
P = ABABAC
lps = {0, 0, 1, 2 , 3, 0}
```

KMP

- E como isto ajuda? Isso permite pular comparações desnecessárias, por exemplo:
- Pelo KMP:

i

```
S = ABABABCABABABCABABABC
P = ABABAC
lps = {0, 0, 1, 2 , 3, 0}
```

E agora?
mantemos o valor de i (ponteiro para posição de S)
j = lps[j - 1] = 3

KMP

- E como isto ajuda? Isso permite pular comparações desnecessárias, por exemplo:
- Pelo KMP:

i

```
S = ABABABCABABABCABABABC
P =   ABABAC
lps = {0, 0, 1, 2 , 3, 0}
```

KMP

```
int a[MAX], n, m;
char S[MAX], P[MAX];

void calculatePrefix(){
    int i = 0, j = -1;
    a[0] = -1;
    while(i < m){
        while(j >= 0 && P[i] != P[j])
            j = a[j];
        i++; j++;
        a[i] = j;
    }
}
```

KMP

```
vector<int> KMP2(){    //retorna todas as ocorrências da substring
    vector<int> resp;
    int i = 0, j = 0;
    calculatePrefix();
    while(i < n){
        while(j >= 0 && S[i] != P[j])
            j = a[j];
        i++; j++;
        if (j == m){
            resp.push_back(i - m);
            j = a[j];
        }
    }
    return resp;
}
```

KMP

- Sugestão para entender mais sobre o KMP e suas aplicações:
    - [Algoritmo de KMP | Vídeo do Bruno Monteiro]

String Hashing

- Uma técnica bastante interessante e relativamente simples de se utilizar é a de String Hashing.
- Primeiramente, vamos revisar, de forma muito intuitiva, o conceito de Hashing.

Hashing

- Podemos pensar em uma problema de busca da seguinte forma:
  - Considere um conjunto de chaves K e um conjunto de valores V, de forma que cada chave k está associada a um único valor v ($map[k] = v$).
  - Dado um valor c qualquer, encontrar a chave k a qual ele está associado (pensando em um vetor, encontrar a posição em que ele se encontra)

Hashing

- O Hashing (tabela de dispersão) consiste em um método de cálculo de endereço (de chave) a partir do valor, de forma que, no caso médio, a chave pode ser encontrada em tempo constante.

Hashing

- Exemplo: encontrar a posição em que um certo nome está armazenado.
  - Complexidade:
    - $O(n)$, se o vetor não estiver ordenado
    - $O(\log n)$, usando busca binária em vetor ordenado

0	1	2	3	4	5
Wilson	Arissa	Luis	Nicolas	Rene	Pedro

Hashing

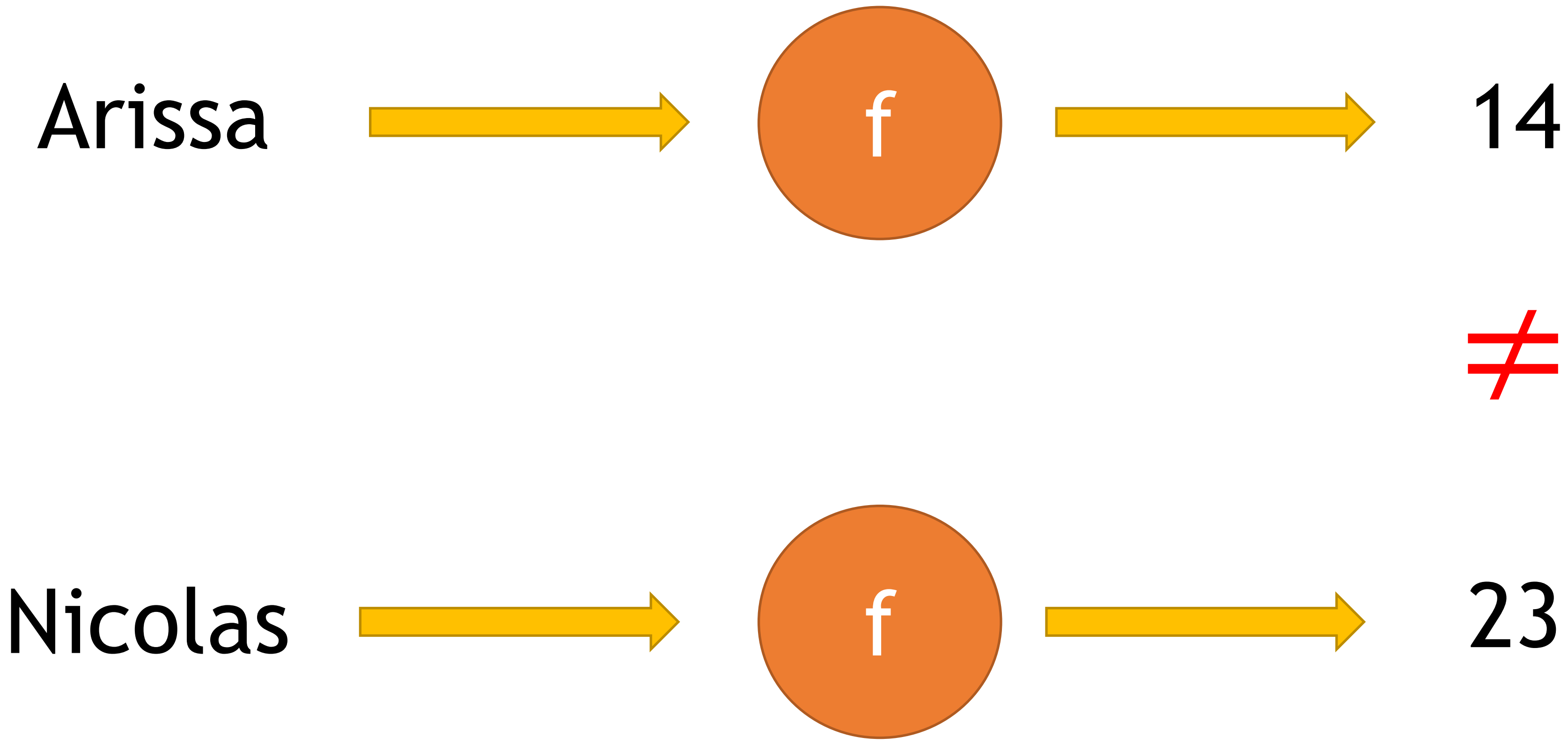
- Agora, suponha que tivéssemos uma “função mágica” que, dado um nome, calcule em tempo constante exatamente a posição que ele deveria ocupar nesse vetor.
  - Essa é a ideia da função *hash*. Claro que na prática isto não é tão simples, mas o nosso foco aqui é mais específico.

0	1	2	3	4	5
Wilson	Arissa	Luis	Nicolas	Rene	Pedro

String Hashing

- De forma muito simplista, uma função *hash* nos gera um número que identifica um dado qualquer (outro número, uma string, uma struct...)
  - Idealmente, identifica unicamente, de forma que cada chave está associada a apenas um valor. Na prática, podemos ter problema de **colisões.**
- E o que isto nos ajuda com strings?
  - Comparar duas substrings tem complexidade $O(n)$, sendo n o número de caracteres.
  - Mas, se calcularmos um **hash** dessas substrings, obteremos seus "números de identificação", que podem ser comparados em $O(1)$.

String Hashing



String Hashing

- Na prática:
  - Dada a(s) string(s) de entrada, realizaremos um pré-processamento para o cálculo do hash ($O(n)$).
  - A partir deste pré-processamento, podemos obter o hash de qualquer substring em $O(1)$.
  - Com isso, a resolução de uma série de problemas terá uma grande queda de complexidade, comparada com a solução por força bruta.

Polynomial Rolling Hash

- Para calcular o hash de uma string qualquer, utilizaremos a técnica de *polynomial rolling*. De forma que, dada uma string s, o $hash(s)$ é calculado da seguinte forma:

$$\begin{aligned}\text{hash}(s) &= s[0]\cdot b^{n-1} + s[1]\cdot b^{n-2} + \ldots + s[n-2]\cdot b^{1} + s[n-1] \mod P \\ &= \sum_{i=0}^{n-1} s[i]\cdot b^{n-i-1} \mod P,\end{aligned}$$

- em que P é um número primo muito grande e b uma constante aleatória (normalmente um primo de valor próximo ao tamanho do alfabeto)
  - A ideia é evitar colisões, mas não entraremos a fundo na fundamentação probabilística deste problema.

Polynomial Rolling Hash

- Exemplo: seja $s = \text{“}ALLEY\text{”}$, $\mathrm{b} = 3$ e $\mathrm{P} = 97$:

Caractere	A	L	L	E	Y
ASCII	65	76	76	69	89

$$(65 \times 3^4 + 76 \times 3^3 + 76 \times 3^2 + 69 \times 3^1 + 89 * 3^0)\ mod\ 97 = 52$$
$$hash(\text{"ALLEY"}) = 52$$

Pré-processamento

- Durante o pré-processamento de nossa substring, construiremos dois vetores que serão importantes para o cálculo do hash de qualquer substring:
  - $h[i]$ = armazena o hash do prefixo s[0...i]
    - $h[0] \; = \; s[0]$
    - $h[i] \; = \; (h[i-1] * b \; + \; s[i]) \; mod \; P$
  - $p[i]$ = armazena o coeficiente polinomial $b^i \; mod \; P$
    - $p[0] \; = \; 1$
    - $p[i] \; = \; (p[i-1] \; * \; b) \; mod \; P$

Pré-processamento

s	ASCII	h	p
A	65		
L	76		
L	76		
E	69		
Y	89		

$b = 3$ e $P = 97$

Pré-processamento

s	ASCII	h	p
A	65	65	1
L	76		
L	76		
E	69		
Y	89		

$b = 3$ e $P = 97$

Pré-processamento

s	ASCII	h	p
A	65	65	1
L	76		
L	76		
E	69		
Y	89		

$$h[1] = (h[0] * b + s[1]) \% P \quad e \quad p[1] = (p[0] * b) \% P$$

$b = 3$ e $P = 97$

Pré-processamento

s	ASCII	h	p
A	65	65	1
L	76	77	3
L	76		
E	69		
Y	89		

$h[1] = (\ 65 * 3 + 76\ )\ \%\ 97 \ e \ p[1] = (\ 1 \ * 3)\ \%\ 97$

$\mathrm{b} = 3 \text{ e } \mathrm{P} = 97$

Pré-processamento

s	ASCII	h	p
A	65	65	1
L	76	77	3
L	76		9
E	69		
Y	89		

$$h[2] = (h[1] * b + s[2])\ \%\ P$$

$b = 3$ e $P = 97$

Pré-processamento

s	ASCII	h	p
A	65	65	1
L	76	77	3
L	76		9
E	69		
Y	89		

$$h[2] = ((h[0] * b + s[1]) * b + s[2]) \ \% \ P$$

$\mathrm{b} = 3$ e $\mathrm{P} = 97$

Pré-processamento

s	ASCII	h	p
A	65	65	1
L	76	77	3
L	76		9
E	69		
Y	89		

$$h[2] = ((s[0] * b + s[1]) * b + s[2]) \ \% \ P$$

$\mathrm{b} = 3$ e $\mathrm{P} = 97$

Departamento de Computação

Programação Competitiva unesp

Pré-processamento

s	ASCII	h	p
A	65	65	1
L	76	77	3
L	76		9
E	69		
Y	89		

$$h[2] = (s[0] * b^2 + s[1] * b + s[2]) \ \% \ P$$

b = 3 e P = 97

Pré-processamento

s	ASCII	h	p
A	65	65	1
L	76	77	3
L	76	16	9
E	69		
Y	89		

$$h[2] = (77 * 3 + 76) \ \% \ 97$$

$\mathrm{b} = 3$ e $\mathrm{P} = 97$

Pré-processamento

s	ASCII	h	p
A	65	65	1
L	76	77	3
L	76	16	9
E	69		27
Y	89		

$$h[3] = (16 * 3 + 69)\ \%\ 97$$

$\mathrm{b} = 3$ e $\mathrm{P} = 97$

Pré-processamento

s	ASCII	h	p
A	65	65	1
L	76	77	3
L	76	16	9
E	69	20	27
Y	89		

$$h[3] = (16 * 3 + 69) \ \% \ 97$$

$\mathrm{b} = 3$ e $\mathrm{P} = 97$

Pré-processamento

s	ASCII	h	p
A	65	65	1
L	76	77	3
L	76	16	9
E	69	20	27
Y	89		81

$h[4] = (20 * 3 + 89) \;\%\; 97$

$\mathrm{b} = 3 \text{ e } \mathrm{P} = 97$

Pré-processamento

s	ASCII	h	p
A	65	65	1
L	76	77	3
L	76	16	9
E	69	20	27
Y	89	52	81

$h[4] = (20 * 3 + 89)\ \%\ 97$

$\mathrm{b} = 3$ e $\mathrm{P} = 97$

Hash de substring

- A partir das estruturas criadas no pré-processamento podemos obter o hash de qualquer substring em tempo constante.
  - Por exemplo, suponha que queremos o hash de “LLE” da substring anterior, dado por $hash(\text{“}LLE\text{”}) = L.b^2 + L.b + E$.
  - Nós já temos calculados os seguintes hashs:
    - $h[3] = hash(\text{“}ALLE\text{”}) = A.b^3 + L.b^2 + L.b + E$
    - $h[0] = hash(\text{“}A\text{”}) = A$
  - A partir destes podemos fazer a seguinte operação:
    - $hash(\text{"LLE"}) = \text{hash}(\text{"ALLE"}) - \text{hash}(\text{"A"}).b^3$
      $= A.b^3 + L.b^2 + L.b + E - A.b^3$

Pré-processamento

s	ASCII	h	p
A	65	65	1
L	76	77	3
L	76	16	9
E	69	20	27
Y	89	52	81

$$hash(\text{"LLE"}) = \text{hash}(\text{"ALLE"}) - \text{hash}(\text{"A"}).b^3$$

$$hash(\text{"LLE"}) = \big(20 \; - (65 \; * 27 \; \% \; 97)\big) \; \% \; 97$$

$$hash(\text{"LLE"}) = 11$$

$$hash(\text{"LLE"}) = (76 * 3^2 + 76 * 3 + 69) \; \% \; 97$$

$$hash(\text{"LLE"}) = 11$$

$\text{b} = 3$ e $\text{P} = 97$

Obs: importante a utilização de aritmética modular

Hash de substring

- A partir das estruturas criadas no pré-processamento podemos obter o hash de qualquer substring em tempo constante.
  - Generalizando:

$$hash(S[l \ldots r]) = (h[r] - h[l-1] * p[r-l+1]\ )\ mod\ P$$

Complexidade do String Hashing

- Pré-processamento: $O(n)$
- Consulta: $O(1)$

Implementação

```
mt19937 rng((int) chrono::steady_clock::now().time_since_epoch().count());
const ll P = 1e18+9;
const ll b = uniform_int_distribution<ll>(0, P-1)(rng);

inline ll mult(ll a, ll b, ll mod){
    return (a*b-(ll)((long double)a/mod*b)*mod + mod)%mod;
}
```

Implementação

```
struct hash_str
{

    vector<ll> h, p;

    hash_str(string s) : h(s.size()), p(s.size()) {
        int n = s.size();
        h[0] = s[0] + 128;
        p[0] = 1;
        for(int i = 1; i < n; i++){
            h[i] = (mult(h[i-1],b,P) + s[i] + 128) % P;
            p[i] = mult(p[i-1],b,P);
        }
    }
```

Implementação

```
    ll sub_hash(int l, int r){
        if (l == 0)
            return h[r];
        ll ans = (h[r] - mult(h[l-1], p[r-l+1],P)) % P;
        if (ans < 0)
            ans += P;
        return ans;
    }

};
```

Busca em strings com String Hashing

- Dada uma string S, de tamanho n, como determinados se a string P, de tamanho m, está presente em S?
- Calculamos o hash das duas strings, e então comparamos P com todas as substrings de tamanho m de S. A ideia é semelhante a força bruta, porém se torna eficiente devido ao uso do *hashing*.
- Algoritmo de Rabin-Karp

Busca em strings com String Hashing

```
hash_str hs(s), hp(p);
int ans = 0;
vector<int> pos;

int n = s.size(), m = p.size();

for(int i = 0; i <= n-m; i++){
    if (hs.sub_hash(i, i+m-1) == hp.sub_hash(0, m-1))
    {
            ans++;
            pos.push_back(i);
    }
}
```

Busca em strings com String Hashing

- Complexidades:
  - Força bruta: $O(n.m)$
  - KMP: $O(n + m)$
  - String Hashing:
    - Pré-processamento: $O(n + m)$
    - Consulta: $O(n - m)$

Exemplos de outros problemas

- Determinar a maior substring de P que ocorre em S
  - Busca binária no tamanho da substring. Procura todas as substrings de tamanho x de P em S.
  - $O(n^2 \log n)$
- Determinar a quantidade de diferentes substrings de S.
  - Para cada possível tamanho de substring cria um set e o povoe com o hash de todas as substrings possíveis. Somando o tamanho dos sets teremos a quantidade de diferentes substrings de S.
  - $O(n^2 \log n)$
- Determinar a maior substring palindrômica de S.
  - *Backward hash*: calcular o hash para a string invertida também.
  - $O(n^2)$
  - Utilizando algoritmo de Manacher (sem String Hashing): $O(n)$

Cuidados

- O maior problema da técnica de String Hashing é a possibilidade da ocorrência de **colisões**: quando duas strings diferentes resultam no mesmo hash.
- Formas de diminuir a probabilidade de ocorrência:
  - Utilização de valores adequados para os parâmetros b e P.
  - Duplo *hashing*.

Outras técnicas para lidar com Strings

- Existem diversas outras técnicas e estruturas que ajudam a lidar com problemas de Strings, por exemplo:
  - Para lidar com palíndromos:
    - Algoritmo de Manacher
    - Palindromic Tree
  - Z-function
  - String matching utilizando autômato finito
  - Algoritmo de Aho-Corasick
  - Trie
  - Suffix Array
  - Suffix Tree
  - Autômato de Sufixos
  - Fatorização de Lyndon / Algoritmo de Duval

Referências

S. Halim e F. Halim. Competitive Programming 2.

Fábio L. Usberti. Processamento de Cadeias de Caracteres. Summer School 2019.

Rafael Grandsire. String Hashing. Summer School 2022.

https://www.youtube.com/watch?v=RXISWaGmYW8

https://cp-algorithms-brasil.com/strings/prefixo.html

https://www.geeksforgeeks.org/kmp-algorithm-for-pattern-searching/

https://www.ime.usp.br/~pf/estruturas-de-dados/aulas/kmp.html

http://www2.ic.uff.br/~boeres/slides_ed/ed_TabelaHash.pdf

https://usaco.guide/CPH.pdf

https://cp-algorithms.com/string/string-hashing.html

https://www.geeksforgeeks.org/string-hashing-using-polynomial-rolling-hash-function/

https://usaco.guide/gold/string-hashing?lang=cpp