Aula 12 · Strings: KMP, trie, hashing e vetor de sufixos

Desafios de Programação

Fernando Kiotheka Vinícius Tikara Date

UFPR

07/06/2023

Strings e Buscas

O que são strings?

- Aqui vamos tratar basicamente de autômatos finitos determinísticos (lembra deles?).
- Não deve ser surpresa o porquê se você já fez Introdução a Teoria da Computação.
- Em geral, vamos tratar de contar, encontrar e reconhecer padrões em strings de maneira eficiente.
- Uma string é definida como uma sequência de caracteres de um alfabeto (geralmente representado por Σ).
- A string vazia é $\emptyset$ ou também ϵ.

Exemplos de string:

- `01010001011` com $\Sigma = \{0, 1\}$
- `ACTACAGAACT` com $\Sigma = \{A, C, T, G\}$
- `banana` com Σ_{ASCII}

Definições iniciais

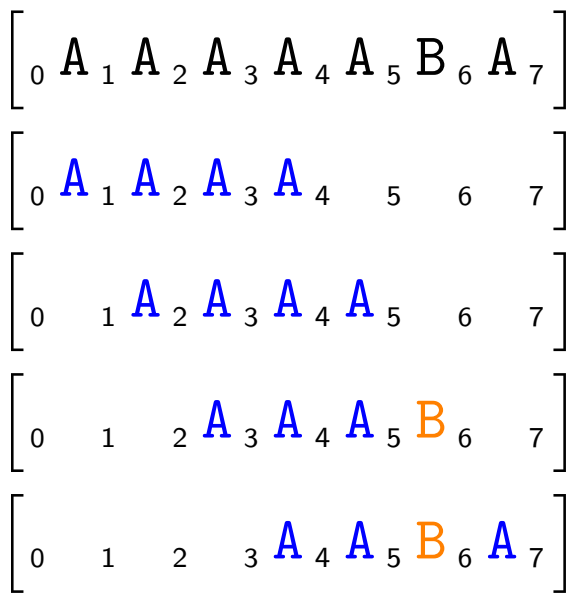
Para uma sequência de caracteres ou string S, definimos:

- **Substring:** Sequência de caracteres consecutivos dentro de S.
- **Subsequência:** Sequência de caracteres não necessariamente consecutivos, mas ordenados, dentro de S.
- **Prefixo:** Substring que começa no primeiro caractere de S.
- **Prefixo próprio:** Prefixo de S diferente de S.
- **Sufixo:** Substring que termina no último caractere de S.
- **Sufixo próprio:** Sufixo de S diferente de S.
- **Borda:** Substring que é prefixo e sufixo de S.

O Problema de Contar Padrões

Vamos formular o problema em contar as agulhas em um palheiro. O palheiro é a string em que fazemos as buscas, e os padrões são as agulhas, que serão buscadas. Queremos achar padrões com sobreposição, isto é, aa aparece duas vezes em aaa.

- Podemos tentar casar a agulha com o palheiro deslizando a agulha de um em um.



Complexidade disso é $\mathcal{O}(M(N - M + 1))$.

Podemos fazer melhor!

- O método que acabamos de ver é aquele usado por padrão no C++, do `search`.
- Desde o C++17, porém, é possível escolher também o algoritmo de Boyer-Moore e o algoritmo de Boyer-Moore-Horspool.
- Eles são algoritmos melhores em strings aleatórias, mas também são, infelizmente, $\mathcal{O}(NM)$ no pior caso.
- Como os problemas de maratona exploram os piores casos, é recomendado não usá-los.
- Assim, não veremos o seu funcionamento, vamos ver um algoritmo que é melhor daqui a pouco.
- Outro problema com o uso do `search` é que ele é terrível para encontrar sobreposições: Você precisa basicamente ficar deslizando a agulha de um em um no pior caso.

Buscando usando search

Código search.cpp

```
#include <bits/stdc++.h>
using namespace std;
#define all(X) X.begin(), X.end()
int main () {
    string hay, ne; while (cin >> hay >> ne) {
        auto it = hay.begin();
        int c = 0;
        while (it = search(it, hay.end(),
                    boyer_moore_horspool_searcher(all(ne))),
                it != hay.end()) {
            cout << hay.substr(0, it - hay.begin())
                 << "[" << ne << "]"
                 << hay.substr(it - hay.begin() + ne.size())
                 << "\n";
            c++; it++; }
        cout << c << "\n";
    }
}
```

Busca usando `search` (continuado)

Entrada	Saída
banana	b[a]nana
a	ban[a]na
	banan[a]
abbba	3
bb	a[bb]ba
	ab[bb]a
abacbabacababa	2
ca	abacbaba[ca]baba
	1
aybabtu	[aybabtu]
aybabtu	1

Hashing

O que é *hashing*?

- *Hashing* pode ser traduzido como trituração.
- Uma função de *hashing* é uma função que tem como
  - **Entrada:** Uma sequência de bytes de tamanho variável.
  - **Saída:** uma sequência de bytes de tamanho fixo, o *hash*.
- O resultado pode ser chamado de resumo criptográfico, e é uma "impressão digital" da entrada.
- Existe um limite para a direcionalidade dessa função: Podemos "comprimir" sem perdas até 13 caracteres do alfabeto latino minúsculo por exemplo em um inteiro de 64 bits: $26^{13} < 2^{64}$.
- Uma propriedade boa para usos criptográficos é que seja praticamente impossível conseguir qualquer tipo de pista sobre os dados originais dado o *hash*.
- Porém, estamos preocupados apenas que tenhamos colisões mínimas para nossos algoritmos, a qualidade do *hash* é desprezível.

Mas pra que?

- Comparar inteiros é mais rápido que comparar vários bytes.
- Podemos ter alguma certeza de que $h(x) = h(y) \implies x = y$.
- Assim, podemos resolver alguns problemas:
  - Achar padrões.
  - Calcular o número de substrings diferentes.
  - Calcular o número de substrings palíndromes.
  - Entre outros.

Função de *hashing* polinomial móvel

$$\begin{aligned} h(s) &= s[0] + s[1]p + s[2]p^2 + \ldots + s[n-1]p^{n-1} \pmod{m} \\ &= \sum_{i=0}^{n-1} s[i]p^i \pmod{m}, \end{aligned}$$

onde p e m são números escolhidos.

- É razoável que p seja um número primo próximo ao tamanho do alfabeto. Para 26 caracteres, vamos usar $p = 31$.
- m deve ser grande pois a probabilidade de duas strings colidirem vai ser por volta de $\approx \frac{1}{m}$.
- Seria legal usar $m = 2^{64}$ pois com `unsigned long long`, você obtém esse módulo "de graça". Porém existe um método para gerar colisões para $m = 2^{64}$, então não é recomendado.
- Ao invés disso vamos usar $m = 10^9 + 9$ que permite fazer multiplicações em `long long`.

Implementação da função de *hashing* polinomial móvel

Código `hash.cpp`

```
#include <bits/stdc++.h>
using namespace std;
using ll = long long;
const int P = 31; const int M = 1e9 + 9;
ll polyhash(string const& s) {
    ll h = 0, p = 1;
    for (char c : s) {
        h += (c - 'a' + 1) * p; h %= M;
        p *= P; p %= M;
    }
    return h;
}
int main() {
    string s; while (cin >> s) {
        cout << polyhash(s) << "\n";
    }
}
```

Achando padrões: Algoritmo de Rabin-Karp

- Podemos usar a mesma ideia para achar correspondências usando *hashing*.
- Esse é o algoritmo de Rabin-Karp, e tem complexidade linear igual o KMP.
- A ideia é comparar a hash de cada prefixo no palheiro e comparar com a hash da agulha.

Implementação do Algoritmo de Rabin-Karp

Código `rabinkarp.h`

```
int search(string hay, string ne) {
    int n = hay.size(), m = ne.size();
    vector<ll> ps (max(n, m), 1);
    for (int i = 1; i < max(n, m); i++)
        ps[i] = (ps[i-1] * P) % M;
    vector<ll> h (n+1, 0);
    for (int i = 0; i < n; i++)
        h[i+1] = (h[i] + (hay[i]-'a'+1) * ps[i]) % M;
    ll hne = 0;
    for (int i = 0; i < m; i++)
        hne = (hne + (ne[i]-'a'+1) * ps[i]) % M;
    int c = 0;
    for (int i = 0; i < n - m + 1; i++)
        if ((h[i+m] - h[i] + M) % M == hne * ps[i] % M) {
            c++; cout << hay.substr(0, i) << "[" << ne << "]"
                 << hay.substr(i+m) << "\n";
        }
    return c;
}
```

Implementação do Algoritmo de Rabin-Karp

Código `rabinkarp.cpp`

```
#include <bits/stdc++.h>
using namespace std; using ll = long long;
const int P = 31; const int M = 1e9 + 9;
#include "rabinkarp.h"
int main() {
    string hay, ne; while (cin >> hay >> ne)
        cout << search(hay, ne) << "\n";
}
```

Entrada	Saída
banana a abbba bb aybabtu aybabtu	b[a]nana ban[a]na banan[a] 3 a[bb]ba ab[bb]a 2 [aybabtu] 1

Árvore de Prefixos (Trie)

Trie, árvore digital ou árvore de prefixos

Árvore que representa um conjunto de strings (dicionário)

- Vértices representam prefixos.
- Cada vértice indica se é o final de uma palavra no dicionário.

Permite buscar todos os prefixos de S no dicionário em $\mathcal{O}(|S|)$.

- Preenchimento automático? Provavelmente usa isso.
- É fácil por exemplo de descobrir o maior prefixo comum entre duas strings.

Algumas outras variações:

- Trie comprimida: Árvore Patricia (Practical Algorithm To Retrieve Information Coded in Alphanumeric).
- Representando conjuntos usando Tries (`https://codeforces.com/blog/entry/83969`).

Implementação da Trie

Código `trie.h`

```
struct trie {
    vector<int> nxt; int cnt; bool leaf;
    trie() : nxt (S, -1), cnt (0), leaf (false) {}
};
vector<trie> t (1);
void ins(string ne) {
    int u = 0;
    for (char c : ne) {
        int ch = to_i(c);
        if (t[u].nxt[ch] == -1) {
            t[u].nxt[ch] = t.size();
            trie n; t.push_back(n);
        }
        u = t[u].nxt[ch];
        t[u].cnt++;
    }
    t[u].leaf = true;
}
```

Implementação da Trie (continuado)

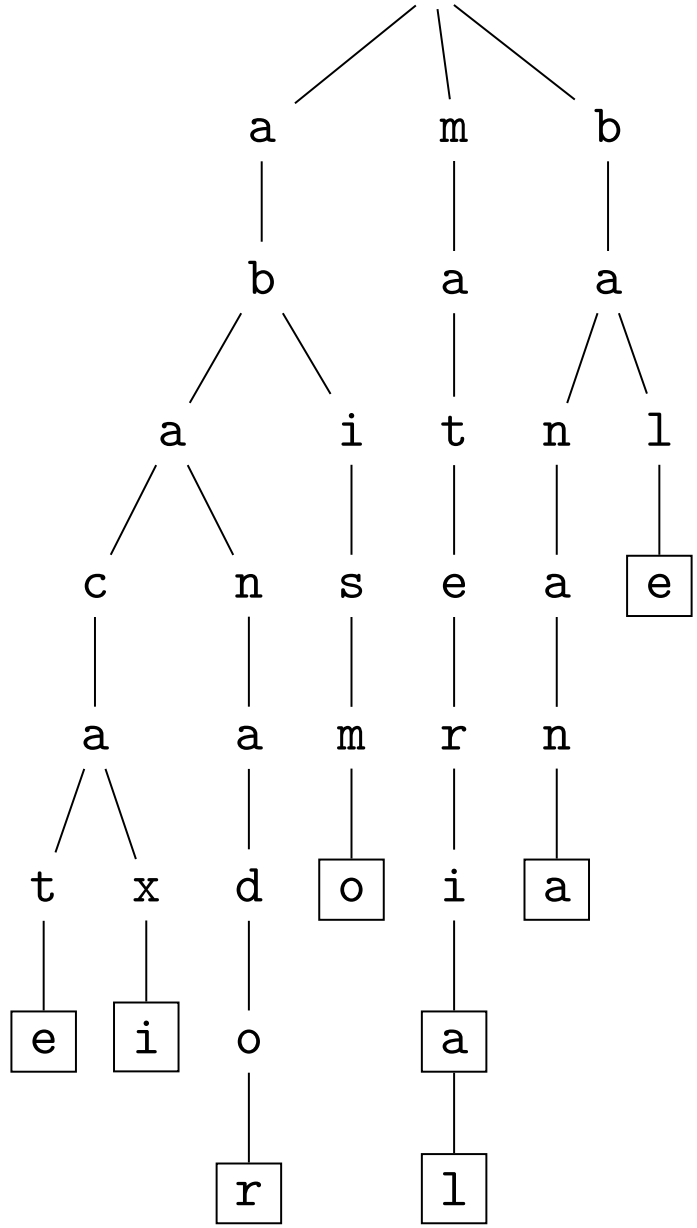
Código `trie.cpp`

```
#include <bits/stdc++.h>
using namespace std;
const int S = 26;
int to_i(char c) { return c - 'a'; }
#include "trie.h"
int main() {
    int n, q; cin >> n >> q;
    while (n--) { string s; cin >> s; ins(s); }
    while (q--) {
        string s; cin >> s;
        int cur = 0;
        for (char c : s) {
            cur = t[cur].nxt[to_i(c)];
            if (cur == -1) { cout << 0 << "\n"; break; }
        }
        if (cur != -1) { cout << t[cur].cnt << "\n"; }
    }
}
```

Implementação da Trie (continuado)

Entrada	Saída
8 8 abacate abacaxi abanador abismo materia material banana bale b a aba ab mate mater z bales	2 4 3 4 2 2 0 0

Implementação da Trie (continuado)



Algoritmo de Knuth-Morris-Pratt (KMP)

Função de prefixo

- A função de prefixo π de uma string S indica, para cada índice $0 \leq i < |S|$ de caractere em S, o tamanho do maior prefixo próprio da substring $S' = S[0..i]$ que é sufixo de S'. $\pi(0) = 0$ por definição.
- Ideia trivial: testar todos os possíveis tamanhos de prefixo ($\mathcal{O}(|S|^3)$).

Implementação da ideia trivial

Código `presimple.cpp`

```
vector<int> pre(string s) {
    int n = s.size();
    vector<int> pi (n);
    for (int i = 0; i < n; i++)
        for (int sz = 0; sz <= i; sz++)
            if (s.substr(0, sz) == s.substr(i-sz+1, sz))
                pi[i] = sz;
    return pi;
}
```

Função de prefixo: Melhorando o algoritmo

Duas observações são importantes para melhorarmos a complexidade do algoritmo:

- $\pi(i+1) \leq \pi(i) + 1 \; \forall \; i$ (caso contrário, poderíamos aumentar o valor de $\pi(i)$ a partir de $\pi(i+1)$[1]).
- Não precisamos comparar strings inteiras, apenas o caractere que "aumentaria" o valor de π para um índice anteriormente calculado.

Assim, inventamos a função de prefixo que é usada no algoritmo Knuth-Morris-Pratt que funciona em $O(m)$ no pior caso. Esse foi o primeiro algoritmo em tempo linear para casamento de padrões.

[1] Contemple os exemplos `abaa`, `abab`, `abad` e `aaad`

Implementação da função de prefixo

Código `pre.h`

```
// O(m)
vector<int> pre(string ne) {
    int n = ne.size();
    vector<int> pi (n, 0);
    for (int i = 1, j = 0; i < n; i++) {
        while (j > 0 && ne[i] != ne[j]) { j = pi[j-1]; }
        if (ne[i] == ne[j]) { j++; }
        pi[i] = j;
    }
    return pi;
}
```

Implementação da função de prefixo (continuado)

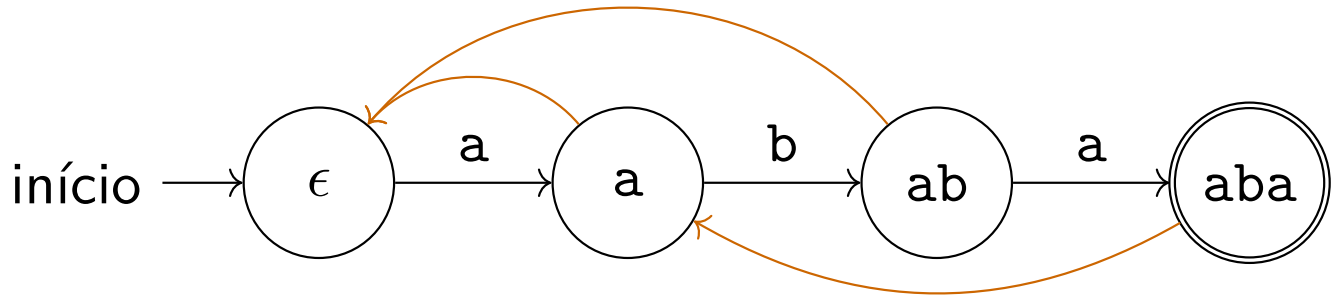
Código `pre.cpp`

```
#include <bits/stdc++.h>
using namespace std;
#include "pre.h"
int main() {
    string s; while (cin >> s) {
        for (int x : pre(s)) { cout << x << " "; }
        cout << "\n";
    }
}
```

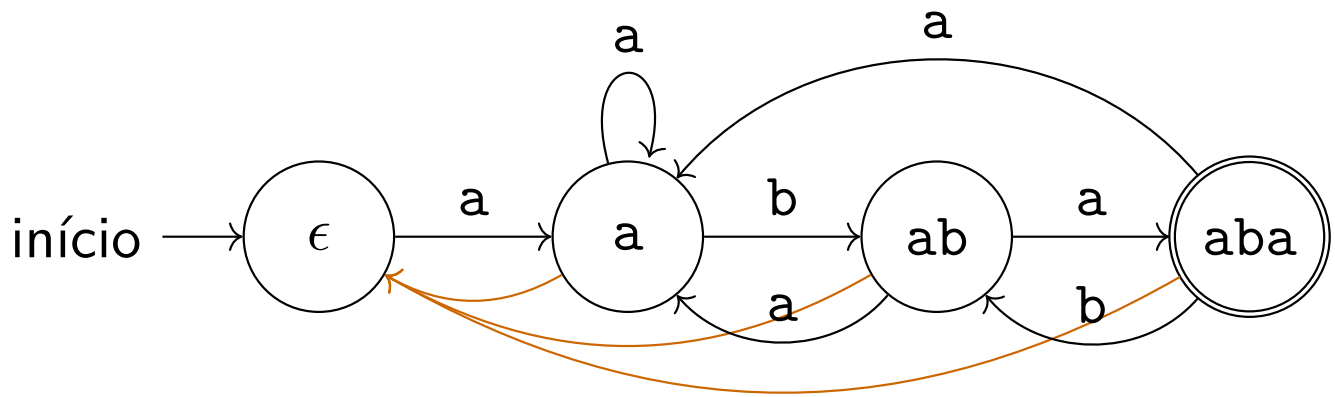
Entrada	Saída
aba	0 0 1
aaa	0 1 2
banana	0 0 0 0 0 0
abaabca	0 0 1 1 2 0 1
abaaczabaac	0 0 1 1 0 0 1 2 3 4 5
aabccaabccaabc	0 1 0 0 0 1 2 3 4 5 6 7 8 9

Cadê o autômato?

A ideia: Ao iterar sobre o texto, tenta ir do estado atual para o próximo. Se não der certo, volta pelo maior prefixo próprio e tenta novamente até conseguir.



Autômato de KMP: Pré-calcular as possibilidades.



Mas a ideia inicial já é boa o suficiente para muitos problemas.

Busca de substring

Código `kmp.cpp`

```
#include <bits/stdc++.h>
using namespace std;
#include "pre.h"
// O(n+m)
void search(string hay, string ne) {
    vector<int> pi = pre(ne); int c = 0;
    for (int i = 0, j = 0; i < hay.size(); i++) {
        while (j > 0 && hay[i] != ne[j]) { j = pi[j-1]; }
        if (hay[i] == ne[j]) { j++; }
        if (j == ne.size()) { c++;
            cout << hay.substr(0, i-j+1) << "[" << ne << "]"
                 << hay.substr(i-j+ne.size()+1) << "\n";
            j = pi[j-1]; }
    }
    cout << c << "\n";
}
int main() {
    string hay, ne;
    while (cin >> hay >> ne) { search(hay, ne); }
}
```

Busca de substring (continuado)

Entrada	Saída
banana	b[a]nana
a	ban[a]na
	banan[a]
abbba	3
bb	a[bb]ba
	ab[bb]a
abacbabacababa	2
ca	abacbaba[ca]baba
	1
aybabtu	[aybabtu]
aybabtu	1

Vetor de Sufixos

Definição

Sendo S uma string,

- O i-ésimo sufixo de S é a substring $S[i..|S| - 1]$.
- O vetor de sufixos de S contém todos os índices dos sufixos de S ordenados lexicograficamente.

Nosso problema é construir o vetor de sufixos. O jeito ingênuo é obtendo todos eles e ordenando-os com uma função pronta ($\mathcal{O}(n^2 \log n)$), mas queremos algo melhor!

Construção elegante: Ideia

- Estenderemos a string S para $S + t$, sendo t um caractere lexicograficamente menor que qualquer caractere em S (normalmente \$).
- Ordenaremos as substrings de S cujos tamanhos são potências de 2.
- Para todo $0 \leq k \leq \lceil \log n \rceil$, ordenamos as substrings de tamanho 2^k com base na ordem das substrings de tamanho 2^{k-1}.

Construção elegante: implementação

Código `sa.h`

```
#define kk first
#define ii second
// O(n lg^2 n)
pair<vector<int>, vector<int>> build_sa(string s) {
    int n = s.size();
    vector<int> sk (s.begin(), s.end());
    vector<pair<pair<int, int>, int>> a(n);
    for (int k = 1; k < n; k *= 2) {
        for (int i = 0; i < n; i++)
            a[i] = { { sk[i], sk[(i+k)%n] }, i };
        sort(a.begin(), a.end());
        sk[a[0].ii] = 0;
        for (int i = 1, r = 0; i < n; i++)
            sk[a[i].ii] = a[i-1].kk == a[i].kk ? r : ++r;
    }
    vector<int> sa (n);
    for (int i = 0; i < n; i++) { sa[i] = a[i].ii; }
    return make_pair(sa, sk);
}
```

Vamos usar a ordenação por contagem!

Código `sar.h`

```
// O(n lg n)
pair<vector<int>, vector<int>> build_sa(string s, int n) {
    vector<int> sk (n), sa (n);
    vector<pair<int, int>> a (n);
    for (int i = 0; i < n; i++) { a[i] = { s[i], i }; }
    sort(a.begin(), a.end());
    for (int i = 0; i < n; i++) { tie(sk[i], sa[i]) = a[i]; }
    vector<int> nsk(n);
    for (int i = 1, r = 0; i < n; i++)
        nsk[sa[i]] = (sk[i-1] == sk[i] ? r : ++r);
    sk.swap(nsk);
    for (int k = 1; k < n; k *= 2) {
        for (int i = 0; i < n; i++) { sa[i] = (sa[i]-k+n)%n; }
        vector<int> nsa(n), cnt(n+1);
        for (int x : sk) { cnt[x+1]++; }
        for (int i = 1; i < n; i++) { cnt[i] += cnt[i-1]; }
        for (int x : sa) { nsa[cnt[sk[x]]++] = x; }
        sa.swap(nsa);
        vector<int> nsk(n);
        for (int i = 1, r = 0; i < n; i++)
            nsk[sa[i]] =
                make_pair(sk[sa[i-1]], sk[(sa[i-1]+k)%n]) ==
                make_pair(sk[sa[i-0]], sk[(sa[i-0]+k)%n]) ? r : ++r;
        sk.swap(nsk);
    }
    return make_pair(sa, sk);
}
```

Exemplo de vetor de sufixos

Código `sa.cpp`

```
#include <bits/stdc++.h>
using namespace std;
#include "sa.h"
int main() {
    string s; cin >> s; s += '$';
    auto [sa, sk] = build_sa(s);
    for (int ix : sa) { cout << s.substr(ix) << "\n"; }
}
```

Entrada	Saída
abaab	$ aab$ ab$ abaab$ b$ baab$

Algoritmo de Kasai

Código `lcp.h`

```
vector<int> compute_lcp (
        string s, vector<int> sa, vector<int> sk) {
    int n = s.size();
    vector<int> lcp (n);
    int k = 0;
    for (int i = 0; i < n-1; i++) {
        int pi = sk[i];
        int j = sa[pi-1];
        // lcp(i) = lcp(s[i..], s[j..])
        while (s[i+k] == s[j+k]) k++;
        lcp[pi] = k;
        k = max(k - 1, 0);
    }
    return lcp;
}
```

Exemplo do algoritmo de Kasai

Código `lcp.cpp`

```
#include <bits/stdc++.h>
using namespace std;
#include "sa.h"
#include "lcp.h"
int main() {
    string s; cin >> s; s += '$';
    auto [sa, sk] = build_sa(s);
    auto lcp = compute_lcp(s, sa, sk);
    for (int i = 0; i < s.size(); i++)
        cout << lcp[i] << " " << s.substr(sa[i]) << "\n";
}
```

Entrada	Saída
abaab	0 $ 0 aab$ 1 ab$ 2 abaab$ 0 b$ 1 baab$

E isso é tudo!

- Todo o conteúdo que vocês vão precisar está aqui!
- Recomendo o TCC do Yan Couto que é uma material didático de algumas estruturas de dados de string: `https://bcc.ime.usp.br/tccs/2016/yancouto/tcc.pdf`.
- No MaratonUSP tem vídeos dele sobre Árvore de Sufixos.
- O Naum tem um vídeo sobre KMP Automata: `https://www.youtube.com/watch?v=D1e3y8XTHg8`.
- Vale muito a leitura a seção de strings do CP-Algorithms.
- Curso sobre vetor de sufixos: `https://codeforces.com/edu/course/2/lesson/2`.
- Bons estudos!