Aula 12 · Strings

Desafios de Programação

Fernando Kiotheka Victor Alflen

UFPR

24/08/2022

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

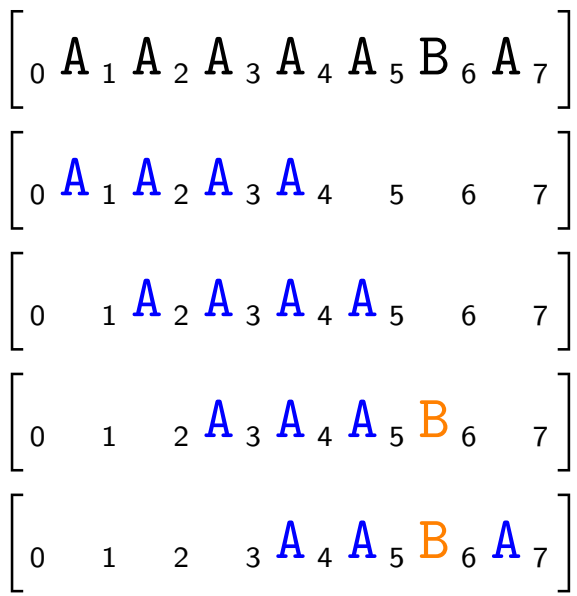
Para uma sequência de caracteres ou string S, definimos:

- **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

Possível solução: Expressões regulares?

- Pra quem já é familiarizado com algumas linguagens de programação, pode achar que os regexes são a solução.
- Regexes (expressões regulares) codificam padrões de casamento de strings criando um atômato.
- Útil para buscas, filtragens e validações.
- Se fala um pouco mais dos detalhes em Teoria da Computação (pois muitos autômatos podem ser expressos com uma expressão dessas).
- Você vai encontrar coisas parecidas em problemas de maratona (* como wildcard, ? simbolizando qualquer caractere, etc).

Algumas expressões regulares

- `a`: caractere “a”.
- `.`: qualquer caractere.
- `a|b`: caractere “a” ou “b”.
- `a(a|b)a`: “aaa” ou “aba”.
- `?`: token anterior aparece uma vez ou nenhuma.
- `+`: token anterior aparece uma ou mais vezes.
- `*`: token anterior aparece zero ou mais vezes.
- `{10}`: token anterior aparece 10 vezes.
- `{10,}`: token anterior aparece 10 ou mais vezes.
- `{10,12}`: token anterior aparece de 10 a 12. vezes

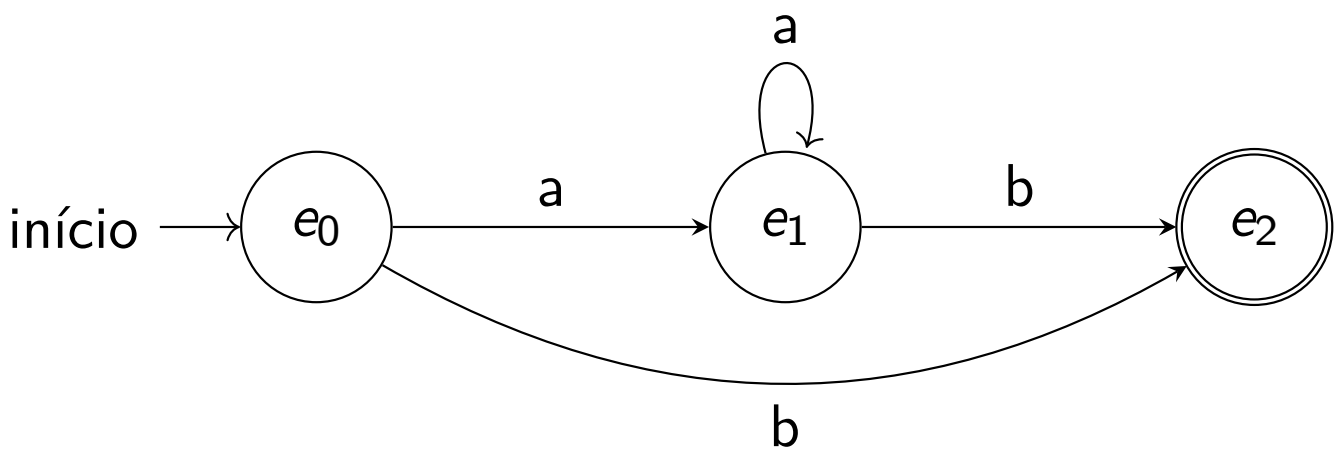
Exemplos de casamentos de expressões regulares

- `a?b`: `b`, `ab`.
- `a+b`: `ab`, `aab`, `aaab`, ...
- `a*b`: `b`, `ab`, `aab`, ...
- `(ab){2,3}`: `abab`, `ababab`.

Autômatos?

- Máquinas abstratas representadas (geralmente) em um grafo direcionado.
- Vértices representam estados.
- Arestas representam a leitura de símbolos.
- Vértices com círculos representam estados que são considerados “finais” (a máquina achou um padrão).

Autômato para a expressão a*b



E serve?

- Por ser uma linguagem tão genérica, as expressões regulares geralmente codificam autômatos não muito eficientes.
- Muitas expressões podem ser **exponenciais**, e isso é muito ruim. Isso é um vetor de ataque: Negação de Serviço usando Expressão Regular!
- Exemplos de expressões regulares problemáticas: `(a+)+`, `(a|aa)+`.
- De toda forma, existe um limite na quantidade de estados, o que leva a ser impossível criar um padrão com mais de $\approx 10^4$ caracteres (o que é bem comum).
- Expressões regulares também são horríveis com contagem de padrões sobrepostos.
- Em geral, não resolvem nossos problemas.

Busca de substring usando RegEx

Código `regex.cpp`

```
#include <bits/stdc++.h>
using namespace std;
#define all(X) X.begin(), X.end()
int main () {
    string hay, ne; while (cin >> hay >> ne) {
        regex r ("(?=(" + ne + "))(.)");
        for (auto i = sregex_iterator(all(hay), r);
                i != sregex_iterator(); ++i) {
            cout << hay.substr(0, i->position())
                 << "[" << ne << "]"
                 << hay.substr(i->position() + ne.size())
                 << "\n";
        }
        cout << distance(sregex_iterator(all(hay), r),
                sregex_iterator()) << "\n";
    }
}
```

Busca de substring usando RegEx (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

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)

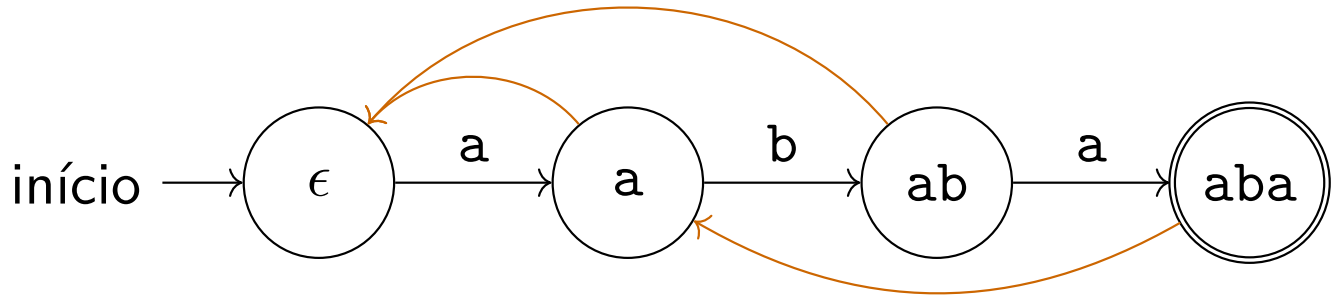
Código `pre.cpp`

```
#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";
    }
}
```

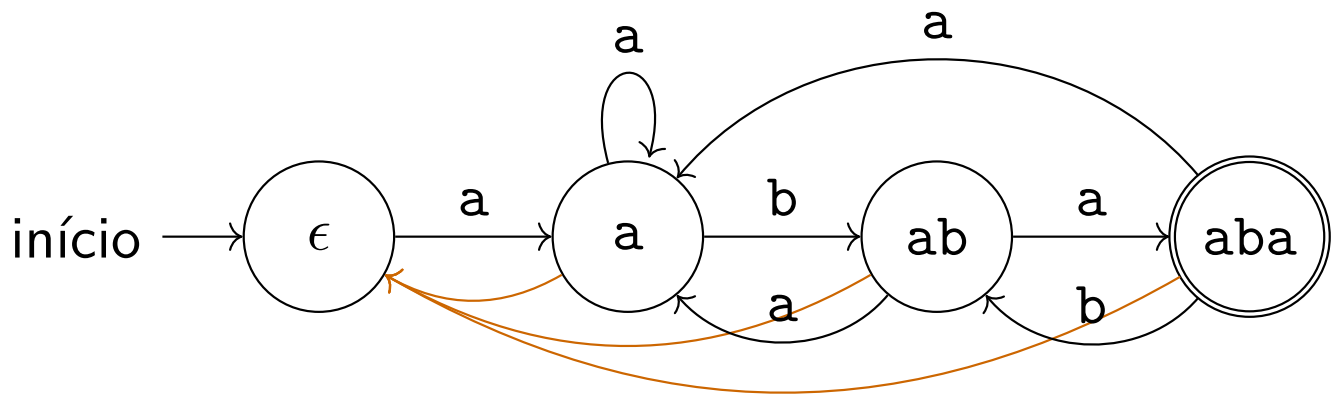
Entrada	Saída
aba	0 0 1
aaa	0 1 2
banana	0 0 0 0 0 0
abaabca	0 0 1 1 2 0 1
abaaczabaac	0 0 1 1 0 0 1 2 3 4 5
aabccaabccaabc	0 1 0 0 0 1 2 3 4 5 6 7 8 9

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

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)

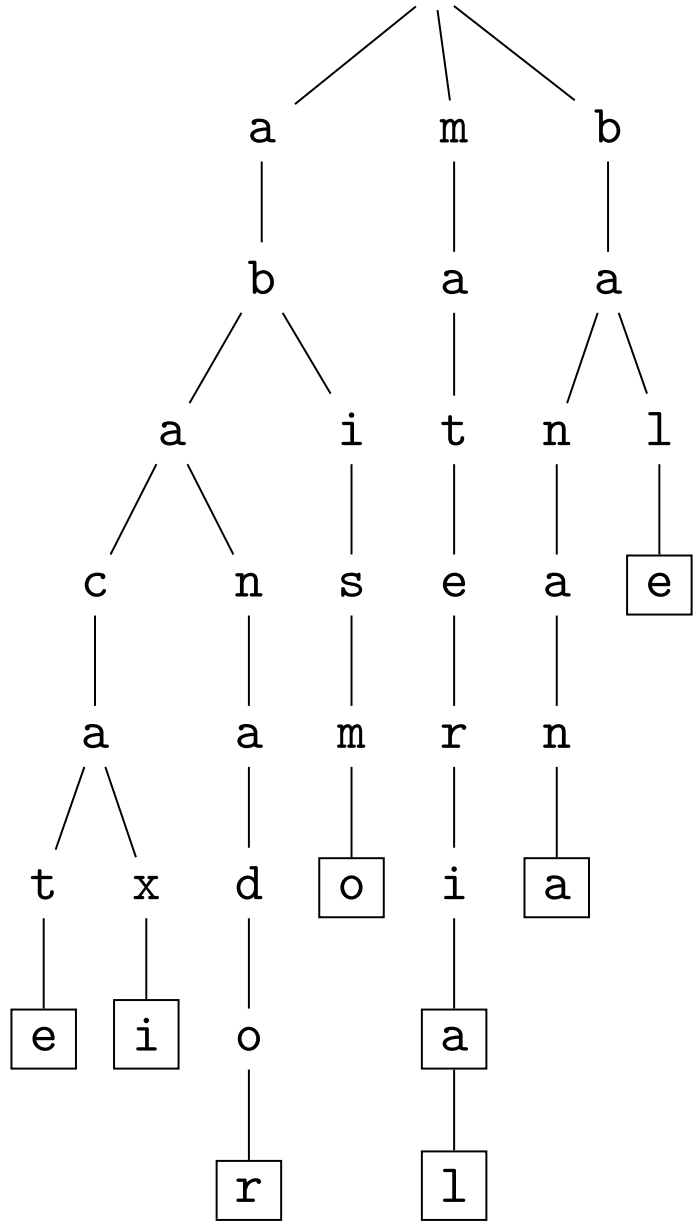
Código `trie.cpp`

```
#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)

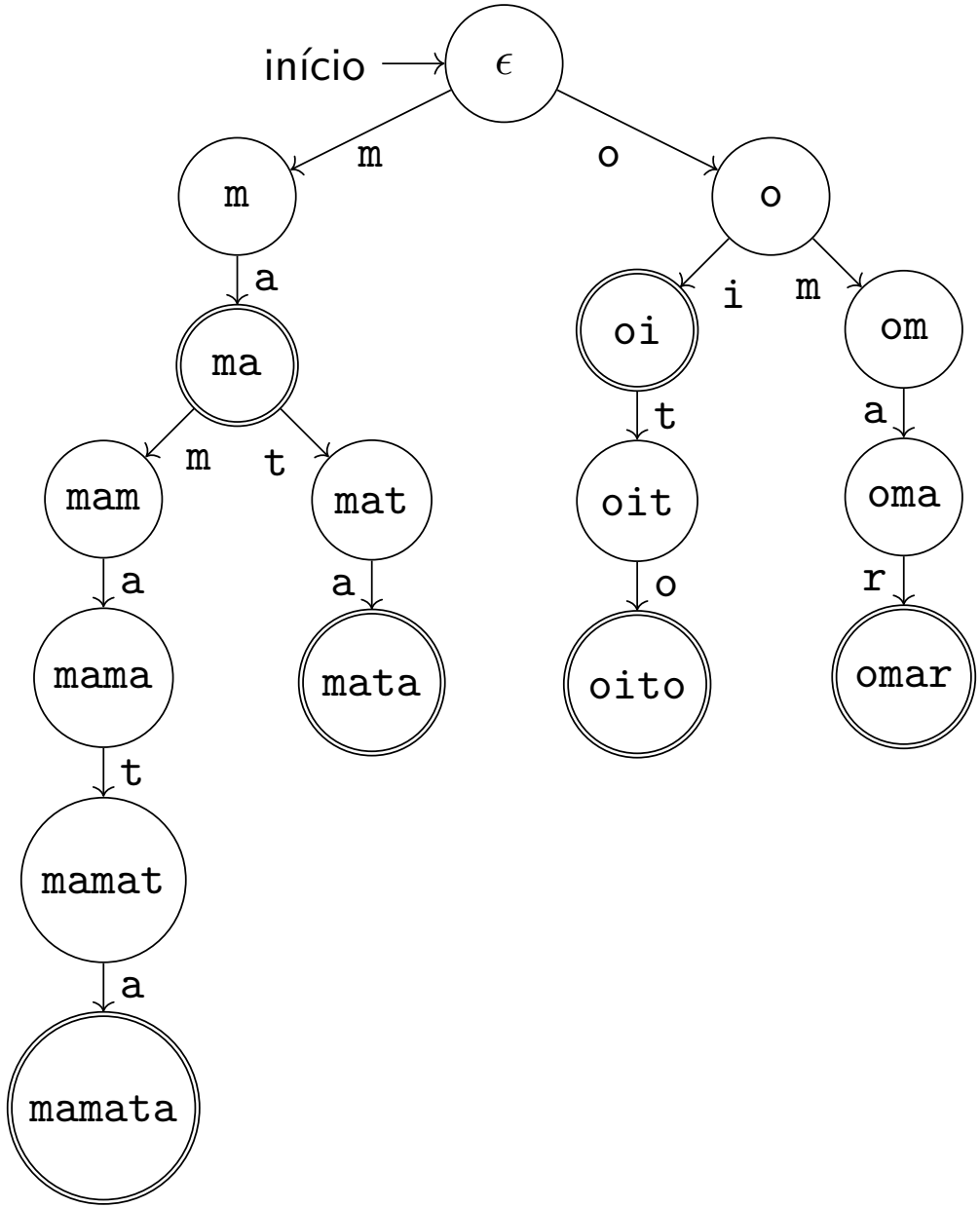


Aho-Corasick

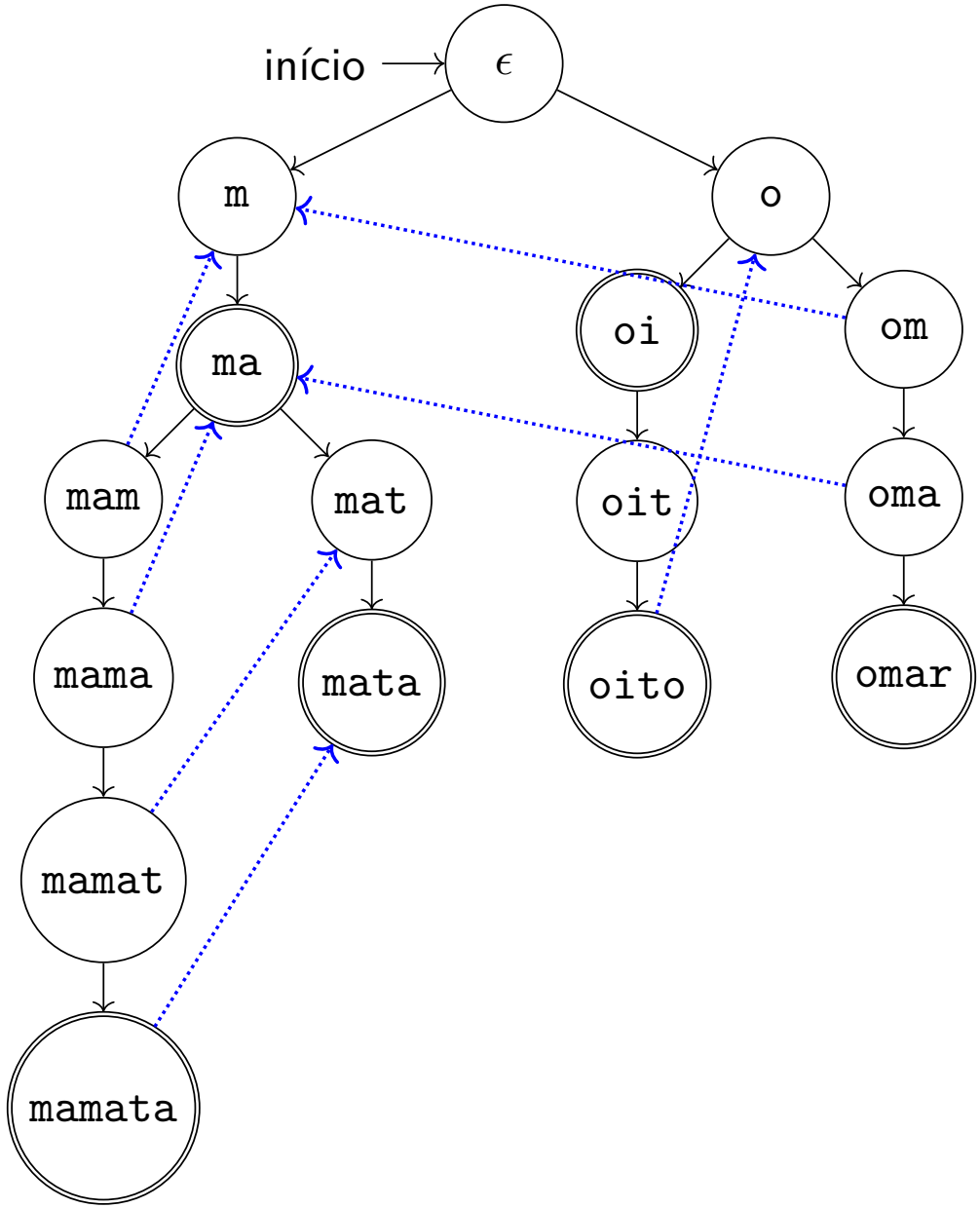
E se a gente colocasse uns ponteiros na nossa trie?

- A trie resolve muitos problemas bacanas, mas se a gente adicionasse uns ponteiros que permitissem procurar coisas nela linearmente, ela ficaria ainda mais bacana!
- Assim, nasce o algoritmo de Aho-Corasick, proposto por Alfred Aho e Margaret Corasick em 1975.
- Vamos adicionar todas as agulhas nessa estrutura de dados, e em seguida podemos se mover no autômato usando o palheiro para descobrir quais correspondências conseguimos achar de forma simultânea.
- Como no KMP, vamos precisar de ponteiros que nos ajudem no caso do casamento falhar.
- Nesse caso atalhos de sufixo: Eles apontam para o maior sufixo próprio disponível na árvore para cada nó.

Atalhos de sufixo



Atalhos de sufixo



Implementação do Aho-Corasick (implementação do nó)

Código `acsnode.h`

```
struct node {
    int p, pc, depth, lnk, out, occ;
    bool leaf;
    vector<int> nxt, go, ix;

    node() : p (0), pc (0), depth (-1),
        lnk (-1), out (-1), occ (0),
        leaf (false), nxt (S, -1), go (S, -1) {}
};
```

Implementação do Aho-Corasick (implementação da inserção)

Código `acs.h`

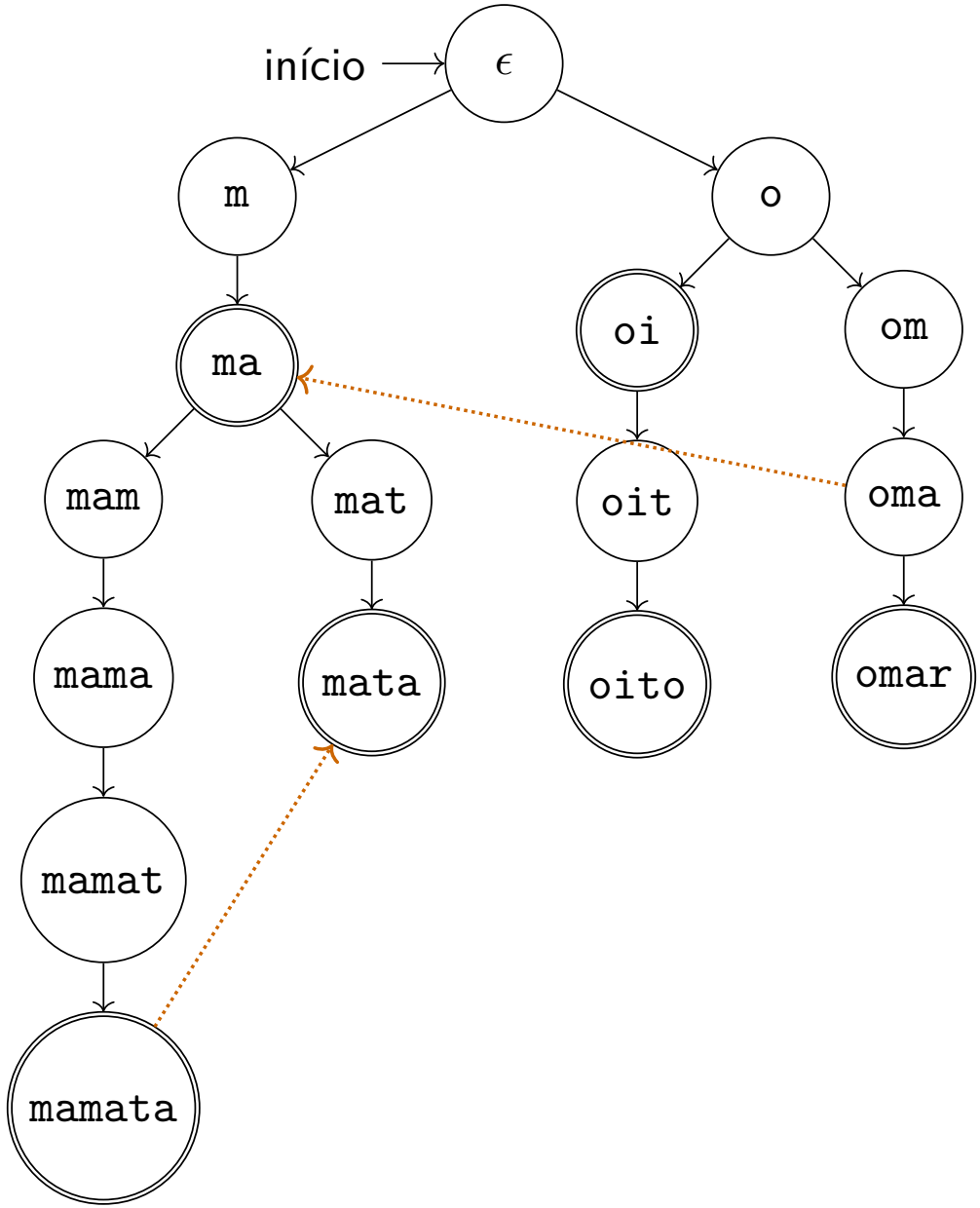
```
vector<node> aca (1);
void ins(string ne, int ix) {
    int u = 0;
    for (int i = 0; i < ne.size(); i++) {
        int ch = to_i(ne[i]);
        if (aca[u].nxt[ch] == -1) {
            aca[u].nxt[ch] = aca.size();
            node n; n.p = u; n.pc = ch; n.depth = i;
            aca.push_back(n);
        }
        u = aca[u].nxt[ch];
    }
    aca[u].leaf = true;
    aca[u].ix.push_back(ix);
}
```

Implementação do Aho-Corasick (implementação dos atalhos)

Código `acsgo.h`

```
int go(int u, int c);
int link(int u) {
    if (aca[u].lnk != -1) { return aca[u].lnk; }
    if (u == 0 || aca[u].p == 0) { return aca[u].lnk = 0; }
    return aca[u].lnk = go(link(aca[u].p), aca[u].pc);
}
int go(int u, int c) {
    if (aca[u].go[c] != -1) { return aca[u].go[c]; }
    if (aca[u].nxt[c] != -1)
        return aca[u].go[c] = aca[u].nxt[c];
    if (u == 0) { return aca[u].go[c] = 0; }
    return aca[u].go[c] = go(link(u), c);
}
int out(int u) {
    if (aca[u].out != -1) { return aca[u].out; }
    int v = link(u);
    if (v == 0 || aca[v].leaf) { return aca[u].out = v; }
    return aca[u].out = out(v);
}
```

Atalhos de saída



Implementação do Aho-Corasick (processamento do palheiro)

Código acsprocess.h

```
vector<int> occ (N); vector<vector<int>> occix (N);
void process(string hay) {
    int u = 0;
    for (int i = 0; i < hay.size(); i++) {
        int ch = to_i(hay[i]);
        u = go(u, ch);
        for (int v = u; v != 0; v = out(v)) {
            for (auto ix : aca[v].ix)
                // match at (i - aca[v].depth), O(len + ans)
                occix[ix].push_back(i - aca[v].depth);
            aca[v].occ++;
        }
    }
    for (int u = 0; u < aca.size(); u++)
        for (auto ix : aca[u].ix)
            // number of matches, O(len)
            occ[ix] += aca[u].occ;
}
```

Implementação do Aho-Corasick

Código acs.cpp

```
#include <bits/stdc++.h>
using namespace std;
const int N = 5e5+15, S = 26;
int to_i(char c) { return c - 'a'; }
#include "acsnode.h"
#include "acs.h"
#include "acsgo.h"
#include "acsprocess.h"
int main() {
    string s; cin >> s;
    int k; cin >> k;
    for (int i = 0; i < k; i++) {
        string t; cin >> t; ins(t, i); }
    process(s);
    for (int i = 0; i < k; i++)
        cout << occ[i] << "\n";
}
```

Implementação do Aho-Corasick (continuado)

Entrada	Saída
aybabtuabaayb	1
	0
8	5
bab	1
abc	4
a	2
bt	1
b	2
yb	
abtu	
ab	

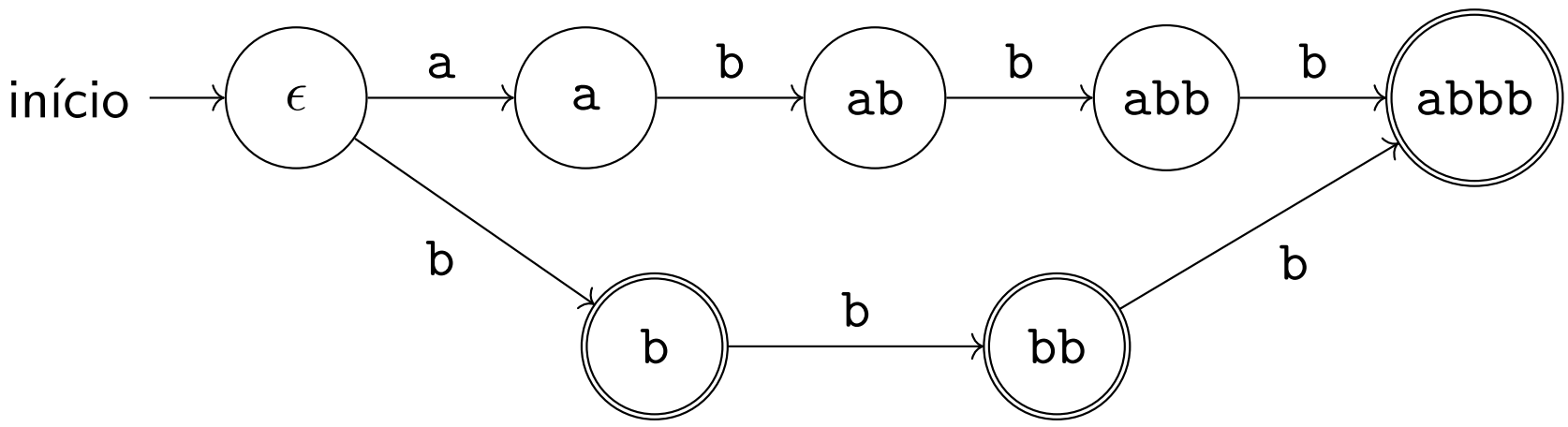
Autômato de Sufixos

Todas as substrings, é?

- É possível fazer uma árvore com todos os sufixos de uma string. Essa árvore é chamada de Árvores de Sufixos.
- O código da mesma é bem grande, então não vamos se aprofundar nela, deixamos esse trabalho pros vídeos do MaratonUSP do Yan Couto.
- Porém, vamos falar de uma estrutura que é tão poderosa quanto: O autômato de sufixos.
- Por ser composto de todos os sufixos, uma propriedade excelente é que todas substrings são um caminho da raiz até algum vértice do autômato.
- O mais fantástico é que precisam de apenas $\mathcal{O}(NS)$ de memória, pois cabem todos os sufixos em um espaço $2N$.

O autômato de sufixos

Para uma string abbb:



Implementação do Autômato de Sufixos (inserção)

Código `saut.h`

```
int last = 0, sz = 1;
vector<int> len (2*N), lnk (2*N), acc (2*N);
vector<vector<int>> nxt (2*N, vector<int>(S));
void add(int c) {
    int cur = sz++; len[cur] = len[last]+1;
    int p = last; last = cur;
    for (; ~p && !nxt[p][c]; p = lnk[p])
        nxt[p][c] = cur;
    if (!~p) { lnk[cur] = 0; return; }
    int q = nxt[p][c];
    if (len[q] == len[p]+1) { lnk[cur] = q; return; }
    int qq = sz++; len[qq] = len[p]+1; lnk[qq] = lnk[q];
    copy(nxt[q].begin(), nxt[q].end(), nxt[qq].begin());
    for (; ~p && nxt[p][c] == q; p = lnk[p])
        nxt[p][c] = qq;
    lnk[cur] = lnk[q] = qq;
}
```

Implementação do Autômato de Sufixos (construção)

Código `sautbuild.h`

```
void build(string& s) {
    len[0] = 0, lnk[0] = -1;
    for (char ch : s) { add(to_i(ch)); }
    for (int u = last; u; u = lnk[u]) { acc[u] = 1; }
}
```

Implementação do Autômato de Sufixos (contagem)

Código `sautcnt.h`

```
vector<int> cnt (2*N);
vector<int> ord;
void reverse_topo(int u) {
    cnt[u] = 1;
    for (int c = 0; c < S; c++) if (nxt[u][c])
        if (!cnt[nxt[u][c]])
            reverse_topo(nxt[u][c]);
    ord.push_back(u);
}
void process_count() {
    for (int u : ord) {
        cnt[u] = 0;
        for (int c = 0; c < S; c++) if (nxt[u][c])
            cnt[u] += cnt[nxt[u][c]];
        if (acc[u]) { cnt[u]++; }
    }
}
```

Implementação do Autômato de Sufixos

Código saut.cpp

```
#include <bits/stdc++.h>
using namespace std;
const int N = 1e5+15, S = 26;
int to_i(char c) { return c - 'a'; }
#include "saut.h"
#include "sautbuild.h"
#include "sautcnt.h"
int search(string& t) {
    int u = 0; for (auto c : t) {
        u = nxt[u][to_i(c)];
        if (!u) { return 0; }}
    return cnt[u];
}
int main() {
    string s; cin >> s; build(s);
    reverse_topo(0); process_count();
    int k; cin >> k;
    while (k--) { string t; cin >> t; cout << search(t) << "\n"; }
}
```

Implementação do Autômato de Sufixos

Entrada	Saída
aybabtuabaayb	1
	0
8	5
bab	1
abc	4
a	2
bt	1
b	2
yb	
abtu	
ab	

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$

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!