

Thuật toán Aho Corasick

Aho-Corasick

Người viết:

- Nguyễn Minh Nhật - HUS High School for Gifted Students

Reviewer:

- Lê Minh Hoàng - Trường Đại học Khoa học Tự nhiên - ĐHQG TP.HCM
- Nguyễn Hoàng Vũ - Trường Đại học Công nghệ - ĐHQGHN

Giới thiệu

Trước khi đọc bài viết này, bạn cần trang bị kiến thức về các chủ đề sau:

- [Trie \(Cây tiền tố\)](#) - Cấu trúc dữ liệu giúp quản lý một tập hợp các xâu.
- [Thuật toán Knuth-Morris-Pratt \(KMP\)](#) - Thuật toán giúp tìm các lần xuất hiện của một xâu trong xâu khác. Ngoài ra, dù cũng không quá cần thiết, kiến thức về KMP Automaton sẽ giúp bạn hiểu rõ bài này hơn.

Nếu như đã học qua Trie và thuật toán KMP, đã bao giờ bạn nghĩ tới việc kết hợp chúng chưa? Liệu ta có thể chạy KMP trên nhiều xâu cùng một lúc?

Aho-Corasick là một thuật toán giúp bạn quản lý một tập xâu và giải bài toán



Cho N xâu S_i và Q xâu T_j . Với mỗi xâu T_j , liệt kê tất cả các lần xuất hiện của các xâu S_i ở trong xâu T_j này.

Độ phức tạp của Aho-Corasick là:

- Xây dựng trong $O(c * \sum |S_i|)$ với c là số lượng chữ cái khác nhau trong N xâu S_i và $|S_i|$ là số chữ cái trong xâu S_i . Giả sử bài toán cho các xâu chữ cái la-tinh viết thường, độ phức tạp là $26 * \sum |S_i|$.
- Truy vấn trong $O(|T|)$, với $|T|$ là độ dài xâu truy vấn.

Ký hiệu

Trong bài viết này, ta quy ước các ký hiệu như sau:

- Nếu không có giải thích gì thêm, c là độ lớn của bảng chữ cái (thường là 26, số chữ cái từ [A-Z](#)).
- Với S là một xâu, $|S|$ là độ dài của xâu S .

- ▶ Với S là một chuỗi, S_i ($0 \leq i < |S|$) là chữ cái thứ i trong chuỗi S . Ta quy ước $S_i \in [0, c)$ (một số tài liệu khác có thể đặt giá trị này là σ)
- ▶ Với S là một chuỗi, $S_{l..r}$ ($0 \leq l \leq r < |S|$) là chuỗi con liên tiếp từ l tới r của chuỗi S .

Ngoài ra, trong bài viết này, ta tạm gọi cấu trúc dữ liệu được xây dựng bởi thuật toán Aho-Corasick là cây Aho-Corasick.

Thuật toán xây dựng

Xây dựng Trie

Cách xây dựng Trie, các bạn có thể tham khảo ở bài viết về [Trie \(Cây tiền tố\)](#). Để xây dựng cây Aho-Corasick, ta xây dựng Trie đối với tập chuỗi S . Cài đặt của bước này như sau:

```

1  struct trie{
2      struct node{
3          int cnt = 0, nxt[26];
4          node() {fill(nxt, nxt+26, -1);}
5      };
6      vector<node> g = {node()};
7      void insert_string(const string &s){
8          int p = 0;
9          for (char c: s){
10             if (g[p].nxt[c - 'a'] == -1){
11                 g[p].nxt[c - 'a'] = g.size();
12                 g.emplace_back();
13             }
14             p = g[p].nxt[c - 'a'];
15         }
16         g[p].cnt++;
17     }
18 };

```

Ở mỗi nút của Trie, ta lưu một biến **cnt** là số lượng chuỗi trong tập S kết thúc ở nút này. Tùy vào câu hỏi cụ thể của bài toán, biến này có thể lưu các giá trị khác.

Chú ý: theo tính chất của cấu trúc dữ liệu Trie, mỗi nút trong cây sẽ đại diện cho một **tiền tố của một chuỗi** S_i nào đó. Tính chất này sẽ cần sử dụng trong các phần sau.

Nhìn lại thuật toán Knuth-Morris-Pratt

Hãy cùng nhìn lại cách hoạt động của thuật toán [Knuth-Morris-Pratt](#):

```

1  vector<int> prefix_function(const string &s){
2      vector<int> pi(s.size());
3      for (int i=1, p=0; i<s.size(); i++){
4          while (p && s[p] != s[i])
5              p = pi[p-1];
6      }

```

```

7         p = pi[i] = p + (s[p] == s[i]);
8     }
9     return pi;
10 }
11
12 void print_matches(const string &s, const string &t){
13     vector<int> pi = prefix_function(s);
14     for (int i=0, p=0; i<t.size(); i++){
15         while (p && s[p] != t[i])
16             p = pi[p-1];
17         p += s[p] == t[i];
18         if (p == s.size()) cout << i << "\n";
19     }
20 }

```

Hàm `print_matches(s, t)` in ra tất cả các lần xuất hiện của xâu S trong xâu T . Hàm này thực hiện

- Xây dựng hàm tiền tố `pi` cho xâu S .
- Khởi tạo con trỏ `p` trên xâu S , ban đầu trở về vị trí 0.
- Duyệt lần lượt qua từng chữ cái của xâu T . Con trỏ `p` nhảy theo hàm `pi` cho tới khi tìm được vị trí mà $S_p = T_i$. Sau mỗi bước duyệt, con trỏ chỉ tới **tiền tố dài nhất của S mà trùng với một hậu tố của $T_{0..i}$** .

Xây dựng cây Aho-Corasick

Với mỗi nút P không phải nút gốc, ta xây dựng **liên kết hậu tố (suffix link)** từ P trở tới một nút V_P có tính chất: xâu được biểu diễn bởi V_P là **hậu tố dài nhất** (khác P) của P .

Để hiểu hơn, **liên kết hậu tố** của Aho-Corasick tương đương với **hàm tiền tố** của KMP.

Giải bài toán ban đầu

Trước khi đi vào cách xây dựng liên kết hậu tố, ta giải một bài toán tương đương với bài toán ban đầu, như sau:

Cho tập N xâu S và xâu T . Liệt kê tất cả các lần xuất hiện của các xâu S_i trong xâu T .

Giả sử đã xây dựng được liên kết hậu tố cho tất cả các nút trong Trie. Ta khởi tạo một con trỏ `p` chỉ tới gốc của Trie và duyệt từng chữ cái trong xâu T . Sau bước duyệt thứ i , ta duy trì tính chất: Con trỏ `p` chỉ tới nút Trie sâu nhất (hay biểu diễn cho xâu dài nhất) mà trùng với một **hậu tố của $T_{0..i}$** .

Tính chất của con trỏ `p` tương đương với tính chất được duy trì trong hàm `print_matches`. Do việc trở tới một nút trong Trie chính là trở tới **tiền tố của một (vài) xâu**

S_i nào đó, ta có thể hình dung việc này như thực hiện Knuth-Morris-Pratt đồng thời trên tất cả các xâu trong tập S .

Để duy trì được tính chất này, con trỏ p sẽ nhảy theo liên kết hậu tố tới khi tìm được vị trí p mà **có cạnh đi tiếp với nhãn T_i** .

Phép kiểm tra này tương đương với việc kiểm tra $S_p = T_i$ của Knuth-Morris-Pratt trên tất cả các xâu cùng lúc (và dừng lại nếu có một xâu thoả mãn).

Để hiểu rõ hơn về các bước chạy của thuật toán này, ta chạy thuật toán với $T = \text{"diduduadi"}$ và $S = \{\text{"di"}, \text{"du"}, \text{"didu"}, \text{"dudua"}, \text{"duadi"}, \text{"didi"}\}$.

0:00 / 1:23

Chú ý: Để tìm được tất cả các xâu thuộc S trùng với một hậu tố của $T_{0..i}$ tương ứng với con trỏ p , ta cần phải tìm trên tất cả các nút tới được bằng cách nhảy một (vài) bước từ p theo liên kết hậu tố.

Như vậy, thuật toán giải chia thành các bước như sau:

1. Khởi tạo con trỏ p , ban đầu trỏ tới gốc của Trie
2. Duyệt T theo thứ tự từ trái sang phải. Với mỗi T_i :
 1. Kiểm tra p **có cạnh đi tiếp với nhãn T_i** hay không. Nếu không, gán $p := p.\text{suffix_link}$ và kiểm tra lại.
 2. Gán $p := p.\text{nxt}[T[i]]$.
 - Trong trường hợp bước 1 gán p là suffix link của gốc, bước 2 gán $p := \text{root}$
 3. Với mọi v tới được bằng cách nhảy theo suffix link từ p (nói cách khác, v tới được bằng cách gán $p := p.\text{suffix_link}$ một vài lần), in ra j sao cho xâu S_j được biểu diễn bởi v (nếu có).
 - Tập tất cả j được in ra ở bước thứ i chính là tất cả các xâu trong S có lần xuất hiện kết thúc tại vị trí thứ i trong T .

Cài đặt của thuật toán giải được cài đặt trong hàm `print_occurences` dưới đây:

```

1  struct aho_corasick{
2      struct node{
3          int suffix_link = -1, nxt[26];
4          vector<int> leaf;
5          node() {fill(nxt, nxt+26, -1);}
6      };
7      vector<node> g = {node()};
8      void print_sindex(int p){
9          for (int v = p; v != -1; v = g[v].suffix_link)
10             for (int j: g[v].leaf)
11                 cout << j << "\n";
12     }
13     void print_occurences(const string &t){
14         for (int i=0, p=0; i<t.size(); i++){
15             while (p != -1 && g[p].nxt[t[i] - 'a'] == -1)
16                 p = g[p].suffix_link;
17             p = p == -1 ? 0 : g[p].nxt[t[i] - 'a'];
18
19             cout << "ENDING AT POSITION " << i << ":\n";
20             print_sindex(p);
21         }
22     }
23 };

```

Xây dựng Automaton (Tối ưu thứ nhất)

Việc di chuyển p như trên là $O(1)$ amortized cho mỗi vòng lặp i . Nói cách khác, tổng số lần di chuyển trong tất cả các lần lặp là $O(|T|)$ (nhưng độ phức tạp cho mỗi lần là không xác định).

Dựa theo chứng minh độ phức tạp của Knuth-Morris-Pratt, bạn đọc có thể thử tự chứng minh độ phức tạp của thuật toán trên.

Chứng minh:

- Sau mỗi lần lặp i , độ sâu của nút được biểu diễn bởi p tăng tối đa 1. Như vậy, số lần tăng là $|T|$.
- Sau mỗi lần lặp gán $p := p.suffix_link$, độ sâu của nút được biểu diễn bởi p giảm ít nhất 1. Như vậy, số lần lặp gán nhỏ hơn hoặc bằng số lần lặp i , là $|T|$.

Như vậy, số lần lặp để thay đổi p (hay p) là $O(T)$.

Ta sẽ tối ưu thao tác di chuyển này thành $O(1)$ cho mỗi bước.

Ở mỗi bước lặp trong hàm `print_occurences`, ta chỉ quan tâm tới hai giá trị p (con trỏ p) và $t[i]$. Như vậy, ta có thể xây dựng hàm `go(p, c)` trả về giá trị mới của con trỏ nếu con trỏ hiện tại đang ở p và chữ cái tiếp theo là c .

Bạn có thể để ý: Aho-Corasick là một Finite Deterministic Automaton, trong đó nút của Trie là trạng thái (state), và `go` là hàm chuyển (transition function).

Tối ưu này tương đương với việc xây dựng KMP Automaton cho một chuỗi.

Xây dựng liên kết hậu tố

Hàm `go` trong tối ưu thứ nhất được xây dựng dựa trên liên kết hậu tố. Thử vị thay, việc xây dựng liên kết hậu tố cũng sử dụng hàm `go`. Sau đây, ta sẽ xây dựng cả hai cấu trúc này cùng lúc.

Nhắc lại, liên kết hậu tố của một nút P sẽ biểu diễn cho **hậu tố dài nhất** (khác P) của P . Rõ ràng, liên kết hậu tố của một nút sẽ là một nút có độ sâu nhỏ hơn (hay độ dài của chuỗi được biểu diễn ngắn hơn). Như vậy, ta có thể sử dụng thuật toán duyệt theo chiều sâu BFS.

Sau đây là cài đặt Aho-Corasick của người viết, xây dựng liên kết hậu tố và mảng `go` từ Trie đã xây dựng từ bước trên.

```

1  struct aho_corasick{
2      struct node{
3          int suffix_link = -1, cnt = 0, nxt[26], go[26];
4          node() {fill(nxt, nxt+26, -1);}
5      };
6      vector<node> g = {node()};
7      void build_automaton(){
8          for (deque<int> q = {0}; q.size(); q.pop_front()){
9              int v = q.front(), suffix_link = g[v].suffix_link;
10             for (int i=0; i<26; i++){
11                 int &nxt = g[v].nxt[i], nxt_sf = v ? g[suffix_link].nxt[i] : 0;
12                 if (nxt == -1) nxt = nxt_sf;
13                 else{
14                     g[nxt].suffix_link = nxt_sf;
15                     q.push_back(nxt);
16                 }
17             }
18         }
19     }
20 };

```

Đối với nút gốc, ta đặt `suffix_link := -1` do liên kết hậu tố của nút gốc không được định nghĩa. Đối với nút P là con trực tiếp của nút gốc trên Trie, rõ ràng chuỗi được biểu diễn bởi nút P chỉ có 1 chữ cái và `suffix_link` của nút phải trở về gốc.

Quan sát ví dụ, bạn đọc hãy thử tự chứng minh công thức `g[nxt].suffix_link = g[g[v].suffix_link].go[i]`

Gợi ý: `g[v].suffix_link` là hậu tố dài nhất của `v`, và ta đang thêm chữ cái `i` vào cả hai chuỗi.

Chứng minh:

- Trường hợp `g[v].nxt[i] != -1`: Rõ ràng `g[v].go[i] = g[v].nxt[i]` theo định nghĩa.
- Trường hợp `g[v].nxt[i] == -1`: Việc thực hiện `g[g[v].suffix_link].go[i]` tương đương với việc ta thực hiện một lần nhảy `p` theo liên kết hậu tố, sau đó cố gắng đi theo cạnh `i`. Trong trường hợp ở nút liên kết hậu tố `g[v].suffix_link` cũng không đi được theo `i`, `g[g[v].suffix_link].go[i]` sẽ dẫn tới các nút cha cho tới khi nào ta có thể đi được theo cạnh `i` (hoặc không thể đi được và phải đi về nút gốc).

Như vậy, ta đã xây dựng được cây Aho-Corasick.

Xây dựng liên kết thoát (tối ưu thứ hai)

Sau bước lập thứ i (đã duyệt qua $T_{0..i}$), ta nhận được con trỏ `p`. Nếu nhảy theo liên kết hậu tố của `p` cho tới khi về tới gốc (như hàm `print_sindex`), ta có thể tìm thấy được tất cả chuỗi $Y \in S$ trùng với hậu tố của chuỗi $T_{0..i}$. Nói cách khác, với mỗi $0 \leq i < |T|$, ta tìm được các j sao cho $T_{j..i}$ thuộc S . Tuy nhiên, với mỗi i , việc tìm kiếm này là $O(n)$ do số lượng lần nhảy theo liên kết hậu tố từ một nút P bất kỳ là $O(|P|) = O(n)$ (với $|P|$ là độ dài chuỗi biểu diễn bởi nút P).

Với giới hạn đầu vào 10^5 , ta có thể tìm được tất cả các j hay không? Quan trọng hơn, số lượng j thoả mãn điều kiện này là bao nhiêu?

Sau đây, ta sẽ chứng minh số lượng vị trí j thoả mãn với mỗi vị trí i là $O(\sqrt{\sum |S_k|})$ (với $\sum |S_k|$ là tổng số chữ cái trong các chuỗi thuộc S).

Gọi Y là tập các chuỗi thuộc S mà tồn tại j sao cho $Y = T_{j..i}$. Do các vị trí j khác nhau, độ dài các chuỗi thuộc Y khác nhau. Rõ ràng, $\sum |Y_i| \geq \frac{|Y| * |Y+1|}{2}$ (chuỗi ngắn nhất trong Y có độ dài ≥ 1 , chuỗi ngắn thứ hai có độ dài ≥ 2 , ...). Như vậy, $|Y| = O(\sqrt{\sum |S_i|})$. Chứng minh hoàn tất.

Nói một cách nôm na, với mỗi tiền tố X của T , có $O(\sqrt{\sum |S_i|})$ hậu tố của X trùng với một chuỗi thuộc S . Do tất cả các hậu tố của các tiền tố chính là tất cả các chuỗi con liên tiếp và có đúng $|T|$ chuỗi X , có $O(|T| * \sqrt{\sum |S_i|})$ lần xuất hiện của các chuỗi thuộc S trong chuỗi T .

Để có thể tìm kiếm nhanh chóng tất cả các hậu tố của X mà thuộc S , ta lưu thêm "liên kết thoát" (exit link) trên mỗi nút. Liên kết thoát của P sẽ trỏ tới nút E_P sao cho khi nhảy theo liên kết hậu tố từ P dương lên, E_P là nút đầu tiên có $cnt \neq 0$ (đồng nghĩa với việc chuỗi được biểu diễn bởi nút q thuộc S).

Cài đặt hoàn chỉnh

Sau đây là cài đặt hoàn chỉnh của người viết. Tùy vào mục đích sử dụng, bạn có thể cần thêm các biến trên mỗi node hoặc chỉnh sửa số lượng chữ cái. Trong cài đặt này, người viết sử dụng chung mảng `nxt` và `go` để

giúp tiết kiệm bộ nhớ và đơn giản hoá cài đặt.

```

1  struct aho_corasick{
2      struct node{
3          int suffix_link = -1, exit_link = -1, cnt = 0, nxt[26];
4          node() {fill(nxt, nxt+26, -1);}
5      };
6      vector<node> g = {node()};
7      void insert_string(const string &s){
8          int p = 0;
9          for (char c: s){
10             if (g[p].nxt[c - 'a'] == -1){
11                 g[p].nxt[c - 'a'] = g.size();
12                 g.emplace_back();
13             }
14             p = g[p].nxt[c - 'a'];
15         }
16         g[p].cnt++;
17     }
18     void build_automaton(){
19         for (deque<int> q = {0}; q.size(); q.pop_front()){
20             int v = q.front(), suffix_link = g[v].suffix_link;
21             if (v) g[v].exit_link = g[suffix_link].cnt ? suffix_link : g[suffi:
22             for (int i=0; i<26; i++){
23                 int &nxt = g[v].nxt[i], nxt_sf = v ? g[suffix_link].nxt[i] : 0
24                 if (nxt == -1) nxt = nxt_sf;
25                 else{
26                     g[nxt].suffix_link = nxt_sf;
27                     q.push_back(nxt);
28                 }
29             }
30         }
31     }
32 };

```

Giải bài toán ví dụ

Bài toán được trình bày trong phần [Giới thiệu](#) chính là bài [Kattis stringmultimatching: String Multimatching](#) [↗](#). Do bài này sử dụng bảng chữ cái bao gồm tất cả các chữ cái in được (trừ dấu xuống dòng), ta sử dụng bảng chữ cái có độ lớn 128 để dễ dàng cài đặt.

```

#include <bits/stdc++.h>
using namespace std;

```

```

struct aho_corasick{
    struct node{
        int suffix_link = -1, exit_link = -1, nxt[128];
    };

```



```

        vector<int> leaf;
        node() {fill(nxt, nxt+128, -1);}
    };
    vector<node> g = {node()};
    void insert_string(const string &s, int idx){
        int p = 0;
        for (char c: s){
            if (g[p].nxt[c] == -1){
                g[p].nxt[c] = g.size();
                g.emplace_back();
            }
            p = g[p].nxt[c];
        }
        g[p].leaf.push_back(idx);
    }
    void build_automaton(){
        for (deque<int> q = {0}; q.size(); q.pop_front()){
            int v = q.front(), suffix_link = g[v].suffix_link;
            if (v) g[v].exit_link = g[suffix_link].leaf.size() ? suffix_link : 0;
            for (int i=0; i<128; i++){
                int &nxt = g[v].nxt[i], nxt_sf = v ? g[suffix_link].nxt[i] : 0;
                if (nxt == -1) nxt = nxt_sf;
                else{
                    g[nxt].suffix_link = nxt_sf;
                    q.push_back(nxt);
                }
            }
        }
    }
    vector<int> get_sindex(int p){
        vector<int> a;
        for (int v = g[p].leaf.size() ? p : g[p].exit_link; v != -1; v = g[v].exit_link){
            for (int j: g[v].leaf)
                a.push_back(j);
        }
        return a;
    }
};

signed main(){
    cin.tie(0)->sync_with_stdio(0);
    string n_line;
    while (getline(cin, n_line)){
        int n = stoi(n_line);

        vector<int> s_size(n);
        aho_corasick ac;
        for (int i=0; i<n; i++){
            string s; getline(cin, s);
            ac.insert_string(s, i);
            s_size[i] = s.size();
        }
    }
}

```

```

59         ac.build_automaton();
60
61         vector<vector<int>> result(n);
62         string t; getline(cin, t);
63         for (int i=0, p=0; i<t.size(); i++){
64             p = ac.g[p].nxt[t[i]];
65             for (int j: ac.get_sindex(p))
66                 result[j].push_back(i - s_size[j] + 1);
67         }
68
69         for (const vector<int> &v: result){
70             if (v.size() == 0) cout << "\n";
71             else for (int i=0; i<v.size(); i++)
72                 cout << v[i] << " \n"[i == v.size()-1];
73         }
74     }
}

```

Do Aho-Corasick cho ta biết vị trí kết thúc trên T của các lần S_i xuất hiện, ta cần trừ đi $|S_i|$ để tìm được vị trí bắt đầu. Ngoài ra, ta cũng có thể đảo ngược tất cả các xâu đề bài cho và lấy $|T| - 1$ trừ đi kết quả trả về của thuật toán để được vị trí bắt đầu trên T của các lần xuất hiện.

Các kỹ thuật nâng cao với Aho-Corasick

Xây dựng cây liên kết hậu tố

Dễ dàng nhận thấy: các cạnh tạo bởi các liên kết hậu tố tạo thành một cây. Với nhiều bài toán liên quan tới Aho-Corasick, ta cần xây dựng cây này và áp dụng các kỹ thuật như [Đường đi Euler trên cây](#).

Để hiểu rõ hơn về kỹ thuật này, hãy làm thử bài [Codeforces 547E: Mike and Friends](#) [☑](#).

Tóm tắt đề bài

Cho một tập xâu S có n phần tử ($1 \leq n, \sum |S_i| \leq 2 * 10^5$) và q ($1 \leq q \leq 5 * 10^5$) truy vấn có dạng (l, r, k) . Với mỗi truy vấn, trả về tổng số lần xuất hiện của s_k trong các xâu s_i ($l \leq i \leq r$).

Ý tưởng liên kết thoát

Truy vấn của đề bài có thể được đơn giản hóa như sau: Với mỗi truy vấn (l, r, k) , ta giải truy vấn $(1, r, k)$ và $(1, l - 1, k)$, sau đó lấy hiệu hai truy vấn này. Sau đây, ta gọi truy vấn dạng $(1, r, k)$ là (r, k) .

Ta xây dựng cây Aho-Corasick của tập xâu S , sau đó duyệt các truy vấn (r, k) theo thứ tự r tăng dần. Khi duyệt tới $r = r_0$ nào đó, ta sẽ "bật" các xâu thuộc $S[1..r_0]$.

Cụ thể, với mỗi nút V , ta duy trì biến **apr** là số lần xuất hiện của xâu được biểu diễn bởi V trong tập xâu đã "bật". Kết quả của truy vấn chính là giá trị của **apr** tại nút biểu diễn cho xâu S_k .

Ý tưởng tự nhiên cho lời giải là sử dụng liên kết thoát. Để "bật" xâu T , do tập tất cả các hậu tố của các tiền tố chính là tập mọi xâu con, ta duyệt qua tất cả các nút P là tiền tố của xâu T và tăng **apr** cho mọi nút tổ tiên của P trên cây liên kết hậu tố (do tổ tiên của một nút trên cây liên kết hậu tố là tất cả các nút V thoả mãn V là

hậu tố của P). Các truy vấn chỉ quan tâm tới các nút lá (nút biểu diễn hoàn chỉnh một xâu S_i nào đó), ta chỉ cần cập nhật các nút tới được theo liên kết thoát.

Cài đặt liên kết thoát

```
#include <bits/stdc++.h>
using namespace std;

struct query{
    int r, k, idx, coef;
};

struct aho_corasick{
    struct node{
        int suffix_link = -1, exit_link = -1, cnt = 0, apr = 0, nxt[26];
        node() {fill(nxt, nxt+26, -1);}
    };
    vector<node> g = {node()};
    vector<int> insert_string(const string &s){
        vector<int> ptr = {0};
        for (char c: s){
            if (g[ptr.back()].nxt[c - 'a'] == -1){
                g[ptr.back()].nxt[c - 'a'] = g.size();
                g.emplace_back();
            }
            ptr.push_back(g[ptr.back()].nxt[c - 'a']);
        }
        g[ptr.back()].cnt++;
        return ptr;
    }
    void build_automaton(){
        for (deque<int> q = {0}; q.size(); q.pop_front()){
            int v = q.front(), suffix_link = g[v].suffix_link;
            if (v) g[v].exit_link = g[suffix_link].cnt ? suffix_link : g[suffi
            for (int i=0; i<26; i++){
                int &nxt = g[v].nxt[i], nxt_sf = v ? g[suffix_link].nxt[i] : 0
                if (nxt == -1) nxt = nxt_sf;
                else{
                    g[nxt].suffix_link = nxt_sf;
                    q.push_back(nxt);
                }
            }
        }
    }
    void update(int ptr){
        for (ptr = g[ptr].cnt ? ptr : g[ptr].exit_link; ptr != -1; ptr = g[ptr
        g[ptr].apr++;
    }
};

signed main(){
```

```

47     cin.tie(0)->sync_with_stdio(0);
48     int n, q; cin >> n >> q;
49
50     vector<string> s(n);
51     for (int i=0; i<n; i++) cin >> s[i];
52
53     aho_corasick ac;
54     vector<vector<int>> ptrs(n);
55     for (int i=0; i<n; i++) ptrs[i] = ac.insert_string(s[i]);
56     ac.build_automaton();
57
58     vector<query> a;
59     for (int i=0; i<q; i++){
60         int l, r, k; cin >> l >> r >> k;
61         a.push_back({r-1, k-1, i, 1});
62         if (l != 1) a.push_back({l-2, k-1, i, -1});
63     }
64     sort(a.begin(), a.end(), [](const query &a, const query &b){return a.r < b.r});
65
66     vector<int> result(q);
67
68     int ptr_s = 0;
69     for (const query &qr: a){
70         for (; ptr_s <= qr.r; ptr_s++){
71             for (int pos: ptrs[ptr_s])
72                 ac.update(pos);
73             int v = ptrs[qr.k].back();
74             result[qr.idx] += qr.coef * ac.g[v].apr;
75         }
76     }
77     for (int v: result) cout << v << "\n";
78 }

```

Độ phức tạp thời gian của cài đặt trên là $O(Q + \sum S_i \sqrt{\sum S_i})$ do mỗi xâu chỉ được "bật" tối đa một lần.

Ý tưởng Euler tour trên cây

Trong các bài toán với Aho-Corasick, đây sẽ là ý tưởng thường được sử dụng hơn.

Thay vì chỉ cập nhật với các nút tới được liên kết thoát, ta vẫn có thể cập nhật tất cả các tổ tiên của một nút. Bài toán đặt ra là: Cho một cây. Xử lý các truy vấn thuộc một trong hai dạng:

1. Với mỗi nút trên đường đi từ nút tới gốc, thêm 1 vào **apr** của nút này.
2. Truy vấn **apr** của một nút nào đó.

Bài toán con này có thể dễ dàng giải được với **Cập nhật đường đi, truy vấn đỉnh**

Cài đặt Euler tour trên cây

```

#include <bits/stdc++.h>
using namespace std;

struct query{
    int r, k, idx, coef;
};

struct aho_corasick{
    struct node{
        int suffix_link = -1, exit_link = -1, cnt = 0, nxt[26];
        node() {fill(nxt, nxt+26, -1);}
    };
    vector<node> g = {node()};
    vector<int> insert_string(const string &s){
        vector<int> ptr = {0};
        for (char c: s){
            if (g[ptr.back()].nxt[c - 'a'] == -1){
                g[ptr.back()].nxt[c - 'a'] = g.size();
                g.emplace_back();
            }
            ptr.push_back(g[ptr.back()].nxt[c - 'a']);
        }
        g[ptr.back()].cnt++;
        return ptr;
    }
    void build_automaton(){
        for (deque<int> q = {0}; q.size(); q.pop_front()){
            int v = q.front(), suffix_link = g[v].suffix_link;
            if (v) g[v].exit_link = g[suffix_link].cnt ? suffix_link : g[suffi:
            for (int i=0; i<26; i++){
                int &nxt = g[v].nxt[i], nxt_sf = v ? g[suffix_link].nxt[i] : 0
                if (nxt == -1) nxt = nxt_sf;
                else{
                    g[nxt].suffix_link = nxt_sf;
                    q.push_back(nxt);
                }
            }
        }
    }
    vector<vector<int>> to_tree(){
        vector<vector<int>> tree(g.size());
        for (int i=1; i<g.size(); i++)
            tree[g[i].suffix_link].push_back(i);
        return tree;
    }
};

struct BIT{
    static const int OFFSET = 2;
    vector<int> g;
};

```

```

BIT(int n): g(n + OFFSET, 0) {}
void update(int p, int v){
    for (p += OFFSET; p < g.size(); p += p & (-p))
        g[p] += v;
}
int query(int p){
    int v = 0;
    for (p += OFFSET; p; p -= p & (-p))
        v += g[p];
    return v;
}

};

void dfs(const vector<vector<int>> &g, int &tdfs, vector<int> &tin, vector<int> &tout){
    tin[v] = tdfs++;
    for (int u: g[v])
        dfs(g, tdfs, tin, tout, u);
    tout[v] = tdfs-1;
}

signed main(){
    cin.tie(0)->sync_with_stdio(0);
    int n, q; cin >> n >> q;

    vector<string> s(n);
    for (int i=0; i<n; i++) cin >> s[i];

    aho_corasick ac;
    vector<vector<int>> ptrs(n);
    for (int i=0; i<n; i++) ptrs[i] = ac.insert_string(s[i]);
    ac.build_automaton();

    vector<vector<int>> g = ac.to_tree();
    vector<int> tin(g.size()), tout(g.size());
    int tdfs = 0;
    dfs(g, tdfs, tin, tout, 0);

    vector<query> a;
    for (int i=0; i<q; i++){
        int l, r, k; cin >> l >> r >> k;
        a.push_back({r-1, k-1, i, 1});
        if (l != 1) a.push_back({l-2, k-1, i, -1});
    }
    sort(a.begin(), a.end(), [](const query &a, const query &b){return a.r < b.r});

    vector<int> result(q);
    BIT bit(g.size());

    int ptr_s = 0;
    for (const query &qr: a){
        for (; ptr_s <= qr.r; ptr_s++)
            for (int pos: ptrs[ptr_s])

```

```

103         bit.update(tin[pos], 1);
104         int v = ptrs[qr.k].back();
105         result[qr.idx] += qr.coef * (bit.query(tout[v]) - bit.query(tin[v] - 1
106     }
107
108     for (int v: result) cout << v << "\n";
}

```

Độ phức tạp thời gian của cài đặt này là $O((\sum |S_i| + Q) \lg \sum |S_i|)$

Bonus: Ý tưởng chia căn

Trong trường hợp không nghĩ tới việc tách query ra thành hai phần, ta có thể sử dụng [MO's algorithm](#) để đảo lại thứ tự duyệt của các truy vấn cho tối ưu.

Chú ý: Do số lượng thao tác cần dùng để thêm mỗi xâu là khác nhau, ta cần xét tới độ dài các xâu khi sử dụng MO's algorithm.

Như vậy, ta sẽ thực hiện $O(\sum S_i \sqrt{\sum S_i})$ thao tác cập nhật một đường đi trên cây Aho-Corasick và $O(Q)$ thao tác truy vấn.

Ý tưởng tự nhiên ở bước này là sử dụng một cấu trúc dữ liệu có khả năng thực hiện cập nhật trong $O(1)$ và truy vấn tổng một đoạn trong $O(\sqrt{N})$. Bài toán con này có thể xử lý được bằng phương pháp [Chia căn đoạn](#).

Cài đặt ý tưởng chia căn

```

#include <bits/stdc++.h>
using namespace std;
const int SQRT = 450;

struct query{
    int l, r, k, idx;
};

struct aho_corasick{
    struct node{
        int suffix_link = -1, exit_link = -1, cnt = 0, nxt[26];
        node() {fill(nxt, nxt+26, -1);}
    };
    vector<node> g = {node()};
    vector<int> insert_string(const string &s){
        vector<int> ptr = {0};
        for (char c: s){
            if (g[ptr.back()].nxt[c - 'a'] == -1){
                g[ptr.back()].nxt[c - 'a'] = g.size();
                g.emplace_back();
            }
            ptr.push_back(g[ptr.back()].nxt[c - 'a']);
        }
    }
}

```

```

        g[ptr.back()].cnt++;
        return ptr;
    }
    void build_automaton(){
        for (deque<int> q = {0}; q.size(); q.pop_front()){
            int v = q.front(), suffix_link = g[v].suffix_link;
            if (v) g[v].exit_link = g[suffix_link].cnt ? suffix_link : g[suffi:
            for (int i=0; i<26; i++){
                int &nxt = g[v].nxt[i], nxt_sf = v ? g[suffix_link].nxt[i] : 0
                if (nxt == -1) nxt = nxt_sf;
                else{
                    g[nxt].suffix_link = nxt_sf;
                    q.push_back(nxt);
                }
            }
        }
    }
    vector<vector<int>> to_tree(){
        vector<vector<int>> tree(g.size());
        for (int i=1; i<g.size(); i++)
            tree[g[i].suffix_link].push_back(i);
        return tree;
    }
};

void dfs(const vector<vector<int>> &g, int &tdfs, vector<int> &tin, vector<int:
    tin[v] = tdfs++;
    for (int u: g[v])
        dfs(g, tdfs, tin, tout, u);
    tout[v] = tdfs-1;
}

signed main(){
    cin.tie(0)->sync_with_stdio(0);
    int n, q; cin >> n >> q;

    vector<string> s(n);
    for (int i=0; i<n; i++) cin >> s[i];

    vector<int> pfs(n);
    for (int i=0; i<n; i++) pfs[i] = s[i].size();
    partial_sum(pfs.begin(), pfs.end(), pfs.begin());

    aho_corasick ac;
    vector<vector<int>> ptrs(n);
    for (int i=0; i<n; i++) ptrs[i] = ac.insert_string(s[i]);
    ac.build_automaton();

    vector<vector<int>> g = ac.to_tree();
    vector<int> tin(g.size()), tout(g.size());
    int tdfs = 0;
    dfs(g, tdfs, tin, tout, 0);

```



```

76
77     vector<query> a(q);
78     for (int i=0; i<q; i++){
79         int l, r, k; cin >> l >> r >> k;
80         a[i] = {l-1, r-1, k-1, i};
81     }
82     sort(a.begin(), a.end(), [&pfs](const query &a, const query &b) -> bool{
83         int la = a.l == 0 ? 0 : pfs[a.l-1], lb = b.l == 0 ? 0 : pfs[b.l-1];
84         int ra = pfs[a.r], rb = pfs[b.r];
85
86         int block_a = la / SQRT, block_b = lb / SQRT;
87         if (block_a != block_b) return block_a < block_b;
88         return ra == rb ? 0 : (ra < rb) ^ (block_a % 2);
89     });
90
91     vector<int> apr(g.size()), bapr((g.size()-1) / SQRT + 1);
92     auto update_string = [&ptrs, &apr, &bapr, &tin](int ptr, int coef){
93         for (int pos: ptrs[ptr]){
94             pos = tin[pos];
95             apr[pos] += coef;
96             bapr[pos/SQRT] += coef;
97         }
98     };
99
100     int l = 0, r = -1;
101     vector<int> result(q);
102     for (const query &q: a){
103         for (; r+1 <= q.r; r++) update_string(r+1, 1);
104         for (; l-1 >= q.l; l--) update_string(l-1, 1);
105
106         for (; r > q.r; r--) update_string(r, -1);
107         for (; l < q.l; l++) update_string(l, -1);
108
109         int v = ptrs[q.k].back();
110         int &rs = result[q.idx];
111         for (int i=tin[v]; i<=tout[v];){
112             if (i + SQRT > tout[v]+1 || i % SQRT) rs += apr[i], i++;
113             else rs += bapr[i / SQRT], i += SQRT;
114         }
115     }
116
117     for (int v: result) cout << v << "\n";
    }

```

Độ phức tạp thời gian của cài đặt này là $O(Q\sqrt{\sum |S_i|})$

Xử lý query thay đổi tập xâu

Dễ thấy rằng thuật toán xây dựng của chúng ta là một thuật toán offline (tập xâu S không được thay đổi). Trên thực tế, có một số bài toán đòi hỏi thêm bớt các xâu qua từng truy vấn. Đối với những bài toán này, ta có các cách giải quyết như sau:

Xử lý Offline

Nếu bài toán cho các truy vấn từ đầu, ta có thể nhập tất cả các truy vấn, sau đó xây dựng cây Aho-Corasick trên các truy vấn này. Ta xây dựng cây thứ hai với các cạnh là các liên kết hậu tố, sau đó quản lý các nút qua từng truy vấn bằng cách sử dụng Cây phân đoạn và [Đường đi Euler trên cây](#).

Xử lý Online

Trong một số trường hợp, bài toán yêu cầu tạo ra xâu sử dụng kết quả từ truy vấn trước (ta không thể nhập tất cả các truy vấn sau trước khi đưa ra kết quả cho truy vấn trước). Trong trường hợp này, người viết biết tới hai cách xử lý.

Tăng độ phức tạp thêm $O(\lg Q)$

Người viết xin phép chỉ trình bày cách xử lý truy vấn thêm xâu; truy vấn xoá xâu là bài tập dành cho bạn đọc.


Ta lưu $\lg Q$ cây Aho-Corasick khác nhau; cây thứ i có độ lớn là 2^i . Với truy vấn 1, ta thêm vào cây thứ 0 rồi xây dựng luôn. Với truy vấn 2, ta lấy xâu ở truy vấn 1 và truy vấn 2, thêm vào cây thứ 1 rồi xây dựng, và xoá cây thứ 0. Với truy vấn 3, ta lại thêm vào cây thứ 0 rồi xây dựng luôn.

Cứ như vậy, với truy vấn thứ i , ta cố gắng thêm vào cây 1. Nếu cây 1 đã đầy, ta lấy thêm hết xâu ở cây 1 rồi thêm vào cây 2. Nếu cây 2 đã đầy, ta lấy thêm hết và thêm vào cây 3, cứ như vậy. Nếu nháp các truy vấn này, bạn đọc có thể dễ dàng chứng minh được ở cây i sẽ có hoặc 0 hoặc 2^i xâu.

Độ phức tạp được cộng thêm là $O(\lg Q)$, do sau mỗi thao tác, một xâu đang ở tập i chỉ có thể được đưa về các tập sau tập i . Ta chỉ có $\lg Q$ tập như vậy vì tập $\lg Q$ sẽ lưu được $\geq Q$ xâu. Như vậy, mỗi xâu chỉ có thể bị xây dựng lại tối đa $\lg Q$ lần.

Khi có truy vấn hỏi, ta thực hiện các truy vấn này với từng cây Aho-Corasick. Như vậy, độ phức tạp của tất cả các thao tác bị nhân thêm $O(\lg Q)$.

Giữ nguyên độ phức tạp

Cách xây dựng giữ nguyên độ phức tạp được mô tả trong paper [Incremental string matching](#) . Tuy nhiên, trên thực tế, các bài toán thuộc dạng này là rất hiếm. Trong trường hợp gặp, thay vì sử dụng kỹ thuật trên, bạn đọc có thể sử dụng các kỹ thuật mạnh hơn như Suffix Automaton.

Bài tập và gợi ý

1. [Codeforces 1202E: You Are Given Some Strings...](#)

- Gợi ý 1: Với mỗi vị trí x trong T , đếm số i, j sao cho x là vị trí cuối cùng của S_i và $x + 1$ là vị trí đầu tiên của S_j .
- Gợi ý 2: Tính mảng A với A_i là số lượng lần xuất hiện **bắt đầu** tại i , B với B_i là số lượng lần xuất hiện **kết thúc** tại i .

2. [HackerRank two-two: Two Two](#)

- Gợi ý 1: Sinh ra tất cả các xâu là biểu diễn thập phân của 2^x với x nào đó và có chiều dài $\leq |A|$.

3. [Codeforces 696D: Legen...](#)

- Gợi ý 1: Biến đổi bài toán về tìm đường đi qua đúng l cạnh có tổng trọng số lớn nhất.
- Gợi ý 2: Giải bằng nhân ma trận.

4. [Codeforces 163E: e-Government](#)

- Gợi ý 1: Sử dụng Euler tour trên cây.
- Gợi ý 2: Với mỗi tên được bật, cộng 1 vào biến đếm của nút tương ứng trên Aho-Corasick.
- Gợi ý 3: Với mỗi tiền tố của xâu được hỏi, cộng vào đáp án một lượng bằng tổng biến đếm trên đường đi từ con trở tới gốc.

5. [Codeforces 1207G: Indie Album](#)

- Gợi ý 1: Xây dựng Aho-Corasick với các xâu t và Trie với các xâu S .
- Gợi ý 2: Tìm kiếm theo chiều sâu trên Trie, đồng thời di chuyển con trở trên Aho-Corasick.
- Gợi ý 3: Sử dụng Euler tour trên cây.
- Gợi ý 4: Khi di chuyển con trở trên Aho-Corasick, cộng 1 vào các nút tổ tiên trên cây liên kết hậu tố của con trở mới. Khi tìm kiếm xong một nút trên Trie, đảo ngược lại thay đổi con trở và trừ 1 và các nút đã cộng 1.

6. [SPOJ MORSE: Decoding Morse Sequences](#)

- Gợi ý 1: Biến đổi các xâu từ điển được cho thành dãy Morse và sử dụng quy hoạch động
- Gợi ý 2: $dp(i)$ là số lượng cách diễn giải tiền tố độ dài i của mã được cho.
- Gợi ý 3: Xây Aho-Corasick với mã Morse của các xâu từ điển.
- Gợi ý 3: Chỉ có căn giá trị $dp(j)$ đóng góp vào giá trị $dp(i)$.
- Gợi ý 4: Sử dụng liên kết thoát để tìm tất cả các cách diễn giải hậu tố của tiền tố độ dài i của mã được cho.

Được cung cấp bởi [Wiki.js](#)