

Thuật toán KMP

Thuật toán KMP

Người viết: Trịnh Quang Anh - University of Melbourne

Review bởi:

- Cao Thanh Hậu - Đại học Khoa học Tự nhiên - ĐHQG-HCM
- Lê Minh Hoàng - Đại học Khoa học Tự nhiên - ĐHQG-HCM
- Hồ Ngọc Vĩnh Phát - Đại học Khoa học Tự nhiên - ĐHQG-HCM
- Hoàng Xuân Nhật - Đại học Khoa học Tự nhiên - ĐHQG-HCM

Bài viết được dịch từ [đây](#)  và đã được chỉnh sửa bổ sung thêm một số phần.

Kiến thức cần biết: Xử lý chuỗi cơ bản. Bạn đọc có thể xem lại các thuật ngữ và bài tập về chuỗi [tại đây](#) .

Giới thiệu

Một trong những bài toán kinh điển nhất về xử lý chuỗi là bài toán so khớp chuỗi: Cho chuỗi s độ dài n và chuỗi t độ dài m , đếm số lần s xuất hiện trong t . Chẳng hạn, nếu $s = \text{"ab"}$ và $t = \text{"aabcabaab"}$, thì s xuất hiện 3 lần tại các vị trí 1, 4, 7 của t . Mục tiêu của chúng ta là tìm một thuật toán tối ưu hơn để giải quyết bài toán này, thay vì duyệt từng vị trí để kiểm tra trong $O(mn)$.

Knuth-Morris-Pratt (KMP) là một thuật toán có độ phức tạp $O(n + m)$ để giải quyết bài toán so khớp chuỗi. Ý tưởng của KMP là mở rộng một hậu tố kết thúc tại i sang hậu tố kết thúc tại $i + 1$ mà vẫn đảm bảo khớp với tiền tố tương ứng, từ đó cải thiện độ phức tạp nhờ loại bỏ được thao tác so sánh chuỗi tốn kém. Các phần tiếp theo sẽ nói rõ các bước để đạt được điều này. Nhưng trước tiên, ta sẽ làm quen với khái niệm **hàm tiền tố**, một hàm giúp chúng ta phân tích kỹ hơn về các cấu trúc "tự khớp" trong chuỗi s , và là cốt lõi của thuật toán KMP.

Định nghĩa

Lưu ý: Xuyên suốt bài viết này các vị trí trong mỗi chuỗi sẽ được đếm từ 0.

Tiền tố chuẩn. Cho một chuỗi s . Một tiền tố của s được gọi là một *tiền tố chuẩn* nếu nó không là cả chuỗi s .

Hàm tiền tố. Cho một chuỗi s có độ dài n . *Hàm tiền tố* của s được định nghĩa là một mảng π có độ dài n , trong đó $\pi[i]$ là độ dài của tiền tố chuẩn dài nhất của chuỗi con $s[0 \dots i]$ mà cũng là hậu tố của chuỗi con này. Từ định nghĩa này ta có $\pi[0] = 0$.

Biểu diễn toán học của hàm tiền tố:

$$\pi[i] = \max_{k=0 \dots i} \{k : s[0 \dots k-1] = s[i-(k-1) \dots i]\}$$

Ví dụ, hàm tiền tố của xâu "abcabcd" là $[0, 0, 0, 1, 2, 3, 0]$ và hàm tiền tố của xâu "aabaaab" là $[0, 1, 0, 1, 2, 2, 3]$.

Tính nhanh hàm tiền tố là hết sức quan trọng vì nó là một bước thiết yếu trong thuật toán KMP. Ở phần sau, bài viết giới thiệu một thuật toán "ngây thơ" để tính hàm tiền tố với độ phức tạp $O(n^3)$. Từ đó, ta sẽ dùng các nhận xét để tối ưu thuật toán thành độ phức tạp $O(n)$.

Thuật toán ngây thơ


Từ biểu diễn toán học của hàm tiền tố, ta có thể xây dựng một thuật vét cạn như sau: Với mỗi i từ 0 đến $n-1$, duyệt qua mọi k để kiểm tra điều kiện $s[0 \dots k-1] = s[i-(k-1) \dots i]$, khi đó $\pi[i]$ là giá trị lớn nhất của k có điều kiện được thỏa mãn.

```

1  vector<int> prefix_function(string s) {
2      int n = (int)s.length();
3      vector<int> pi(n, 0);
4      for (int i = 0; i < n; i++)
5          for (int k = 0; k <= i; k++)
6              if (s.substr(0, k) == s.substr(i - k + 1, k))
7                  pi[i] = k;
8      return pi;
9  }
```

Do có $O(n^2)$ cặp (i, k) và so sánh hai xâu bằng `s.substr()` mất $O(n)$, độ phức tạp của thuật toán này là $O(n^3)$.

Thuật toán tối ưu để tìm hàm tiền tố

Một chút bối cảnh lịch sử: Thuật toán này được tìm ra bởi Morris, chỉ vài tuần trước khi được Knuth tìm ra một cách độc lập. Morris & Pratt đăng báo cáo vào năm 1970, rồi cả ba đồng xuất bản thuật toán này vào năm 1977 (Nguồn: [Wikipedia](#) )

Phép tối ưu đầu tiên

Nhận xét quan trọng đầu tiên là giá trị của hàm tiền tố chỉ có thể tăng thêm tối đa 1 đơn vị khi duyệt từ i lên $i+1$.

Thật vậy, giả sử $\pi[i+1] > \pi[i] + 1$, ta lấy hậu tố kết thúc ở $i+1$ có độ dài $\pi[i+1]$ và bỏ chữ cái cuối của nó đi. Xâu thu được là một hậu tố kết thúc ở i khớp với tiền tố có cùng độ dài là $\pi[i+1] - 1$, lớn hơn $\pi[i]$, mâu thuẫn với việc $\pi[i]$ là hậu tố dài nhất thỏa mãn của i .

Để hình dung sự mâu thuẫn này, ta xét ví dụ dưới đây. Ở vị trí i , hậu tố dài nhất mà cũng là tiền tố có độ dài 2 ($\pi[i] = 2$), và ở vị trí $i+1$ thì ta có $\pi[i+1] = 4 > 2 + 1$. Khi đó, xâu $s_0 s_1 s_2 s_3$ bằng xâu $s_{i-2} s_{i-1} s_i s_{i+1}$,

dẫn tới hai xâu $s_0 s_1 s_2$ và $s_{i-2} s_{i-1} s_i$ bằng nhau. Do đó, $\pi[i]$ phải bằng 3 chứ không phải 2.

$$\underbrace{s_0 s_1 s_2 s_3}_{\pi[i+1]=4} \dots \underbrace{s_{i-2} s_{i-1} s_i s_{i+1}}_{\pi[i+1]=4}$$

Do đó, khi duyệt sang vị trí tiếp theo, giá trị của hàm tiền tố chỉ có thể tăng thêm 1, giữ nguyên, hoặc giảm đi.

Chỉ riêng nhận xét này là đủ để chúng ta giảm độ phức tạp xuống còn $O(n^2)$: Thay vì mỗi lần duyệt i xét mọi giá trị có thể của k , thuật toán bắt đầu với $k = \pi[i - 1] + 1$ (hàm tiền tố tăng tối đa 1 nên đây là giá trị lớn nhất có thể của k) và giảm dần k đến khi so khớp thành công. Vì hàm tiền tố suốt cả quá trình duyệt i chỉ có thể tăng k tối đa $1 * n = n$ bước, dẫn tới chỉ có tối đa n bước giảm k , nên số bước thực hiện so sánh hai xâu chỉ còn là $O(n)$. Vì vậy, độ phức tạp tổng của thuật chỉ còn là $O(n^2)$ (so sánh xâu vẫn là $O(n)$).

```

1  vector<int> prefix_function(string s) {
2      int n = (int)s.length();
3      vector<int> pi(n, 0);
4      for (int i = 1; i < n; i++)
5          int k = pi[i - 1] + 1;
6          while (k && s.substr(0, k) != s.substr(i - k + 1, k))
7              k--;
8          pi[i] = k;
9      return pi;
10 }
```

Tuy nhiên, chúng ta có thể làm tốt hơn.

Phép tối ưu thứ hai

Trước tiên, ta nhận thấy nhược điểm làm cho thuật toán có độ phức tạp $O(n^2)$ là thao tác so sánh xâu mất $O(n)$.

Như đã đề cập từ đầu bài viết, ý tưởng chính để triệt tiêu thao tác này là mở rộng một hậu tố kết thúc ở i đã khớp tiền tố để có được một hậu tố kết thúc ở $i + 1$ cũng khớp với tiền tố.

Cụ thể, nếu ta có một hậu tố kết thúc tại i khớp với tiền tố cùng độ dài j , và ta có $s[j] = s[i + 1]$, thì hậu tố kết thúc tại $i + 1$ có độ dài $j + 1$ khớp với tiền tố có độ dài $j + 1$.

Vậy nếu ta nhanh chóng duyệt qua được các giá trị j là độ dài cho cặp tiền tố - hậu tố ứng với i đã khớp sẵn, theo thứ tự j giảm dần, thì giá trị j đầu tiên (và lớn nhất) cũng thỏa mãn $s[j] = s[i + 1]$ cho ta biết $\pi[i + 1] = j + 1$. Nói cách khác, nhờ có cặp xâu khớp sẵn mà ta chỉ cần so sánh một cặp ký tự (thao tác có thể thực hiện trong $O(1)$), và từ đó loại bỏ được thao tác so sánh xâu.

Ta bắt đầu duyệt bằng cách xét hậu tố kết thúc tại i có độ dài $j = \pi[i]$. Từ định nghĩa hàm tiền tố, hậu tố này khớp với tiền tố có độ dài $\pi[i]$ ($s[0 \dots \pi[i] - 1]$). Nếu $s[i + 1] = s[j] = s[\pi[i]]$, ta có thể khẳng định được rằng $\pi[i + 1] = \pi[i] + 1$ vì π chỉ tăng tối đa 1 khi từ i sang $i + 1$.

$$\underbrace{s_0 s_1 s_2 s_3}_{\pi[i+1]=\pi[i]+1} \dots \underbrace{s_{i-2} s_{i-1} s_i s_{i+1}}_{\pi[i+1]=\pi[i]+1}$$

Nếu $s[i+1] \neq s[\pi[i]]$, ta duyệt đến hậu tố kết thúc tại i có độ dài lớn nhì, và khớp với tiền tố cùng độ dài.

Nếu đặt độ dài này là k thì ta có $s[0 \dots k-1] = s[i-k+1 \dots i]$ (từ định nghĩa) và $k < j = \pi[i]$:

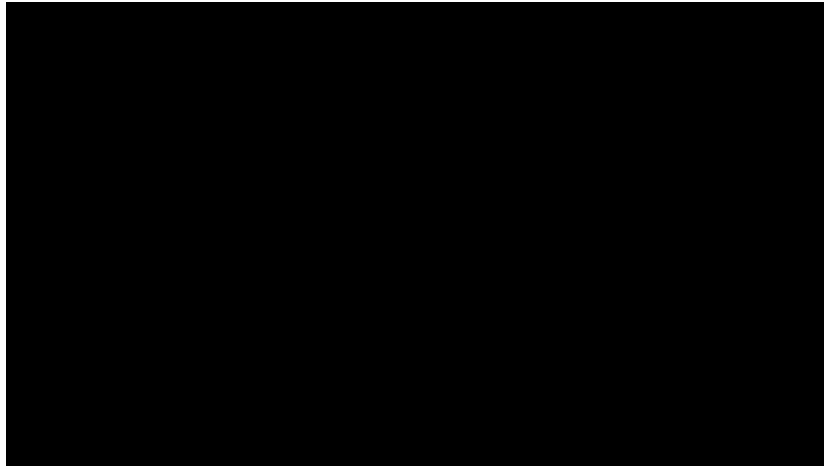
$$\underbrace{s_0 s_1 s_2 s_3 s_4}_{k} \dots \underbrace{s_{i-4} s_{i-3} s_{i-2} s_{i-1} s_i}_{k} s_{i+1}$$

Tuy nhiên, do hậu tố kết thúc ở i độ dài $\pi[i]$ cũng khớp với tiền tố độ dài $\pi[i]$ là $s[0 \dots \pi[i]-1]$, hậu tố độ dài k kết thúc ở $\pi[i]-1$ khớp với tiền tố độ dài k .

$$\underbrace{s_0 s_1 s_2 s_3 s_4}_{k} \dots \underbrace{s_{i-4} s_{i-3} s_{i-2} s_{i-1} s_i}_{k} s_{i+1}$$

Do k lớn nhất, ta phải có $k = \pi[\pi[i]-1]$. Nói cách khác, độ dài hậu tố lớn thứ nhì kết thúc ở i mà khớp với tiền tố tương ứng là $\pi[\pi[i]-1]$. So sánh $s[k]$ và $s[i+1]$, nếu chúng bằng nhau thì ta có $\pi[i+1] = k+1$.

Minh họa cho hai trường hợp trên:



Nếu $s[k] \neq s[i+1]$, lập luận tương tự, độ dài lớn thứ ba cho hậu tố kết thúc tại i khớp với tiền tố là $\pi[\pi[\pi[i]-1]-1]$, độ dài lớn thứ tư là $\pi[\pi[\pi[\pi[i]-1]-1]-1]$,...

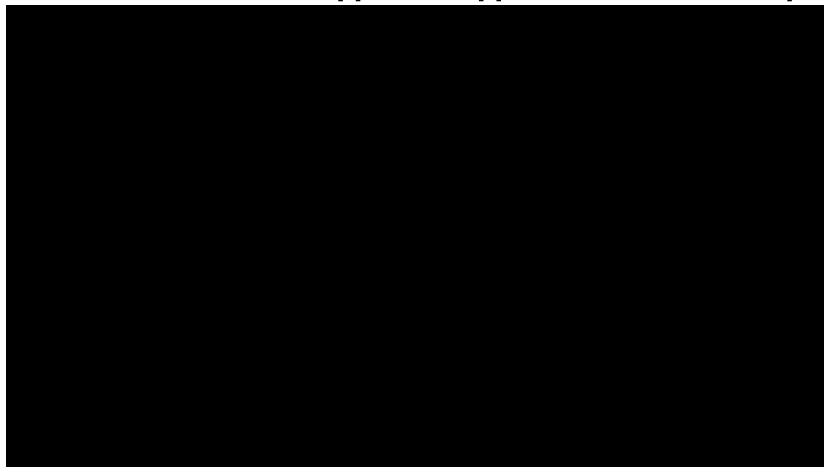
Từ đó, ta có thể duyệt qua mọi hậu tố kết thúc tại i khớp với tiền tố độ dài j như sau: Ban đầu đặt $j = \pi[i]$, để đến độ dài tiếp theo thỏa mãn, ta gán $j = \pi[j-1]$. Tương tự lập luận ở phần [phép tối ưu đầu tiên](#), khi chuyển từ i sang $i+1$ ta gán $j = \pi[i] = j+1$ nên j tăng tối đa 1, và khi cập nhật $j = \pi[j-1]$ thì j giảm ít nhất 1. Do đó, ta chỉ duyệt qua $O(n)$ giá trị của j và mỗi lần thao tác so sánh hai ký tự mất $O(1)$, dẫn đến độ phức tạp tổng là $O(n)$.

Thuật toán cuối cùng

Tổng kết lại, thuật toán của chúng ta hoạt động như sau:

1. Tính hàm tiền tố $\pi[i]$ bằng cách duyệt từ $i = 1$ đến $i = n-1$ (gán mặc định $\pi[0] = 0$).
2. Giả sử đã tính xong $\pi[0], \pi[1], \dots, \pi[i-1]$. Để tính $\pi[i]$, ta khởi tạo biến $j = \pi[i-1]$ mang ý nghĩa là độ dài của hậu tố dài nhất của $i-1$ mà chưa được kiểm tra.
3. Kiểm tra liệu hậu tố của i có độ dài $j+1$ cũng khớp với tiền tố tương ứng hay không bằng cách so sánh hai chữ cái cuối ($s[j]$ và $s[i]$).
4. Nếu chúng bằng nhau, ta tìm được $\pi[i] = j+1$. Nếu không, gán $j = \pi[j-1]$ rồi lặp lại bước 3.

5. Nếu $j = 0$ mà vẫn chưa ghép cặp được với $s[i]$, ta đặt $\pi[i] = 0$ rồi tiếp tục tính $\pi[i + 1]$.



Cài đặt

Cài đặt các bước trên rất ngắn và trực quan:

```

1  vector<int> prefix_function(string s) {
2      int n = (int)s.length();
3      vector<int> pi(n);
4      for (int i = 1; i < n; i++) {
5          int j = pi[i - 1];
6          while (j > 0 && s[i] != s[j])
7              j = pi[j - 1];
8          if (s[i] == s[j])
9              j++;
10         pi[i] = j;
11     }
12     return pi;
13 }
```

Code có độ phức tạp $O(n)$ và sử dụng $O(n)$ bộ nhớ.

Một điểm đáng lưu ý nữa là đây là một thuật toán **online**: Ta có thể đọc lần lượt từng chữ cái của xâu s , mỗi chữ cái mới vào có thể được xử lý ngay lập tức để tính được $\pi[i]$. Nói cách khác, nếu đề bài yêu cầu tính hàm tiền tố nhưng có thêm truy vấn "thêm một chữ cái vào cuối xâu" thì ta vẫn có thể làm như bình thường.

Về mặt bộ nhớ, ta vẫn cần phải lưu lại xâu và giá trị các hàm tiền tố trước đó nên vẫn cần đến $O(n)$ bộ nhớ. Trường hợp đặc biệt là khi hàm tiền tố luôn nhỏ hơn một hằng số M nào đó, khi đó ta chỉ cần lưu $M + 1$ chữ cái đầu và $M + 1$ hàm tiền tố tương ứng của xâu. Nhận xét này sẽ tỏ ra rất hữu ích ở các phần sau.

Ứng dụng

Thuật toán Knuth-Morris-Pratt (KMP)

Quay trở lại với bài toán ban đầu: Đếm số lần xâu s độ dài n xuất hiện trong xâu t độ dài m . Lời giải cho bài toán này - thuật toán KMP - là một áp dụng kinh điển của hàm tiền tố. Vậy làm thế nào để dùng hàm tiền tố khi

có hai xâu cần khớp chứ không phải trong một xâu? Bằng cách gộp chúng vào nhau.

Nhận xét rằng nếu ta nối xâu t vào sau xâu s và hai xâu được ngăn cách bởi một ký tự $\#$ không nằm trong cả 2 xâu (ví dụ nếu s, t gồm toàn chữ cái thì có thể lấy $\#$ là 0, xâu mới là $s + 0 + t$), thì mỗi một lần s xuất hiện trong t tương đương với một vị trí i ở xâu mới có hàm tiền tố $= \pi[i] = n$.

Thuật toán KMP đảm bảo tính đúng đắn, vì nếu i thỏa mãn $\pi[i] = n$ thì phải có $i > n$ (nghĩa là các giá trị i thỏa mãn phải nằm ở nửa sau tạo bởi t) và hậu tố của i có độ dài $\pi[i]$ không thể "tràn" sang phần của s .

Ta có thể cài đặt KMP bằng cách sử dụng code trên: `prefix_function(s + # + t)` trả về hàm tiền tố cho $s + \# + t$ và ta chỉ cần đếm xem có bao nhiêu phần tử trong đó có giá trị là n . Cách cài đặt này tốn $O(n + m)$ bộ nhớ.

Tuy nhiên, việc chọn ký tự ngăn cách là một chữ cái không nằm trong cả hai xâu còn dẫn đến $\pi[i] \leq n$. Như đã nói ở trên, chặn trên này cho phép ta chỉ lưu xâu $s + \#$ và hàm tiền tố của xâu này. Với các vị trí thuộc xâu t thì ta có thể tính π lần lượt (bằng cách lưu một biến j chứa giá trị π ở vị trí hiện tại) và thêm 1 vào đáp án nếu π tại vị trí đang xét là n :

```

1  vector<int> pi = prefix_function(s);
2  int ans = 0; // Số lần s xuất hiện trong t
3  int j = 0; // Hàm tiền tố ở vị trí đang xét của xâu t
4
5  for (int i = 0; i < m; i++) {
6      while (j > 0 && t[i] != s[j])
7          j = pi[j - 1];
8      if (t[i] == s[j])
9          j++;
10     if (j == n)
11         ans++;
12 }
```

Code trên lưu cả hai xâu s, t nên vẫn dùng $O(n + m)$ bộ nhớ, nhưng có thể giảm xuống còn $O(n)$ bộ nhớ nếu ta đọc lần lượt từng ký tự của t rồi tính luôn π thay vì lưu cả xâu rồi mới tính.

Tổng kết lại, thuật toán KMP giải quyết được bài toán so khớp chuỗi trong thời gian $O(n + m)$ và sử dụng $O(n)$ bộ nhớ.

Đếm số lần xuất hiện của từng tiền tố

Bài toán

Ở đây ta xét hai bài toán tương đối giống nhau:

Cho một xâu s độ dài n .

- Với mỗi $0 \leq i < n$, đếm số lần tiền tố $s[0 \dots i]$ xuất hiện trong xâu s .
- Thay vì đếm trong xâu s , đếm số lần $s[0 \dots i]$ xuất hiện trong một xâu t khác có độ dài m .

Lời giải

Trước tiên, ta giải phiên bản 1.

Xét giá trị của $\pi[i]$ tại vị trí i . Dựa vào định nghĩa của π , một trong các hậu tố của $s[0 \dots i]$ là tiền tố

$s[0 \dots \pi[i] - 1]$. $\pi[i]$ lớn nhất nên không thể có tiền tố nào dài hơn mà cũng khớp với một hậu tố của $s[0 \dots i]$, **nhưng có thể có một số tiền tố ngắn hơn thỏa mãn.**

Ý tưởng là duyệt qua mỗi i và kiểm soát xem số lần các tiền tố $s[0 \dots 0]$, $s[0 \dots 1]$, \dots , $s[0 \dots i]$ xuất hiện thay đổi như thế nào. Xét các hậu tố kết thúc ở i mà khớp với tiền tố có cùng độ dài, khi đó ta đơn giản cộng một vào số lần xuất hiện của các tiền tố này.

Như đã đề cập ở phần **thuật toán cuối cùng**, độ dài các tiền tố mà cũng là hậu tố kết thúc ở i từ lớn đến bé là $\pi[i]$, $\pi[\pi[i] - 1]$, $\pi[\pi[\pi[i] - 1] - 1]$, \dots . Ta duyệt qua lần lượt từng tiền tố này rồi cộng 1 vào đáp án cho tiền tố đó (đáp án cho tiền tố độ dài i là $ans[i - 1]$), thì độ phức tạp sẽ là $O(n^2)$:

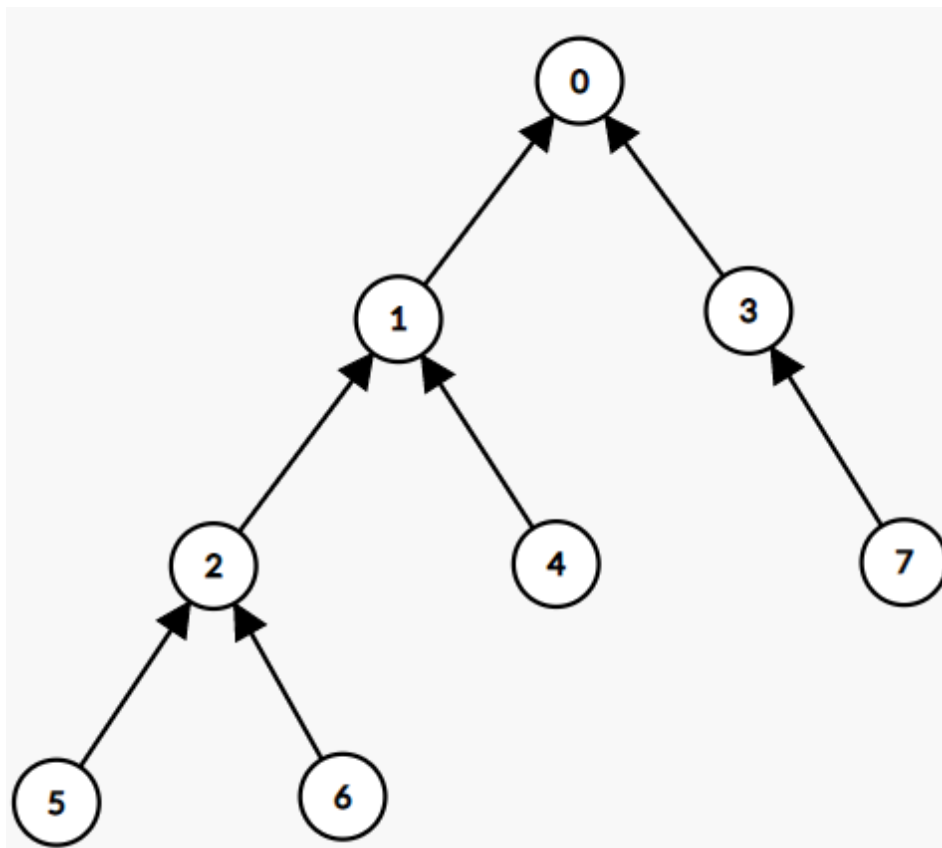
```

1  vector<int> ans(n, 0);
2  for (int i = 0; i < n; i++) {
3      for (int j = i; pi[j]; j = pi[j] - 1) {
4          ans[j]++;
5          if (!pi[j]) {
6              break;
7          }
8      }
9  }
```

Thuật toán trên có độ phức tạp lớn do mỗi tiền tố ta phải duyệt qua nhiều lần. Để khắc phục điều này, ta sẽ dùng ý tưởng tương tự như mảng cộng dồn.

Để ý rằng nếu ta xét đồ thị cho $n + 1$ đỉnh $0, 1, 2, \dots, n$, trong đó đỉnh i ($i \leq n$) ứng với tiền tố thứ i , và có các cạnh nối từ $i + 1$ đến $\pi[i]$ với mọi i từ 0 đến $n - 1$, thì đồ thị này là một cây có hướng.

Ví dụ: với xâu "abcabcd", ta có hàm tiền tố $[0, 1, 0, 1, 2, 2, 3]$, tương ứng với các cạnh $(1, 0)$, $(2, 1)$, $(3, 0)$, $(4, 1)$, $(5, 2)$, $(6, 2)$, $(7, 3)$ của cây:



Ý tưởng này phần nào giúp chúng ta dễ hình dung hơn cách giải: việc tính ans đưa về bài toán cập nhật truy vấn cộng 1 vào các nút trên đường đi từ một nút đến gốc và tìm giá trị tại tất cả các nút sau mọi truy vấn - một bài toán có cách giải dùng mảng cộng dồn.

Giả sử cần cộng 1 vào các giá trị $ans[i], ans[\pi[i] - 1], ans[\pi[\pi[i] - 1] - 1], \dots$. Thay vì duyệt qua từng giá trị một ta chỉ gán $ans[i] = 1$. Sau đó, duyệt giảm dần từ $n - 1$ về 0, khi duyệt đến i thì cộng $ans[i]$ vào $ans[\pi[i] - 1]$ khi duyệt đến $\pi[i] - 1$ thì cộng $ans[\pi[i] - 1]$ vào $ans[\pi[\pi[i] - 1]]$, khi duyệt đến $\pi[\pi[i] - 1]$ thì cộng $ans[\pi[\pi[i] - 1]]$ vào $ans[\pi[\pi[\pi[i] - 1] - 1]]$,... Đến khi duyệt xong thì giá trị của $ans[\pi[i]], ans[\pi[\pi[i] - 1]], \dots$ đều là đã được cộng thêm 1.

Do ta duyệt qua mỗi phần tử đúng một lần và độ phức tạp tính hàm tiền tố là $O(n)$, tổng độ phức tạp thuật chỉ còn là $O(n)$.

```

1 | vector<int> ans(n, 1);
2 | for (int i = n - 1; i > 0; i--)
3 |     if (pi[i])
4 |         ans[pi[i] - 1] += ans[i];

```

Để giải quyết phiên bản thứ 2, ta chỉ cần áp dụng kỹ thuật được sử dụng ở thuật toán KMP: **nối hai xâu để tạo xâu mới $s + \# + t$ và xây dựng hàm tiền tố cho xâu này**. Sau đó, tìm ans tương tự như phiên bản 1, đáp án là ans của các vị trí i thuộc về xâu t ($i \geq n + 1$).

Đếm số xâu con phân biệt trong một xâu

Bài toán

Cho một xâu s có độ dài n . Đếm số xâu con phân biệt của s .

Lời giải

Hướng làm của chúng ta là bắt đầu với xâu rỗng, rồi mở rộng xâu bằng cách thêm lần lượt $s[0], s[1], \dots, s[n-1]$ vào cuối xâu, đồng thời cập nhật số xâu con phân biệt hiện tại. Sau khi đã thêm đủ n ký tự $s[i]$, ta sẽ có được số xâu con phân biệt của xâu ban đầu.

Mỗi lần thêm ký tự $s[i]$ ta sẽ có thêm i xâu mới là $s[0 \dots i], s[1 \dots i], \dots, s[i \dots i]$, nhưng ta chỉ được thêm vào đáp án các xâu không nằm trong xâu trước khi thêm $s[i]$ (hay nói cách khác là không nằm trong $s[0 \dots i-1]$). Làm thế nào để đếm nhanh số xâu không lặp này? Một lần nữa ta sử dụng hàm tiền tố.

Gọi xâu có được sau khi thêm $s[i]$ vào cuối là t . Nhận xét là nếu ta đảo ngược xâu t thì số xâu bị lặp chính là số tiền tố của t khớp với một xâu con nào đó của t . Nếu ta tính hàm tiền tố π trên xâu t bị đảo ngược, thì số xâu bị lặp này chính là giá trị lớn nhất π_{max} của π vì:

- ▶ $s[i - \pi_{max} \dots i]$ khớp với một xâu trong t không phải chính nó (theo định nghĩa hàm tiền tố) nên nó bị lặp, dẫn các xâu $s[i - \pi_{max} + 1], \dots, s[i - 1 \dots i], s[i \dots i]$ cũng bị lặp.
- ▶ Các xâu $s[j \dots i]$ ($j \leq i$) có độ dài lớn hơn π_{max} chắc chắn không lặp vì nếu có thì π_{max} sẽ không phải giá trị lớn nhất của π .

Vậy số xâu mới bị lặp với một xâu con có trước là π_{max} , dẫn đến số xâu cần thêm vào đáp án là $i - \pi_{max}$ (Do có i xâu mới). Tính π_{max} với mỗi i hết $O(i)$ nên tổng độ phức tạp thuật toán là $O(n^2)$.

Ngoài ra, thay vì mỗi bước cập nhật kết quả sau khi thêm một chữ cái vào cuối xâu hiện tại, ta cũng có thể thêm một chữ vào đầu xâu hiện tại, hoặc bắt đầu với xâu hoàn chỉnh và mỗi bước bỏ đi chữ cái đầu (hoặc cuối). Độ phức tạp $O(n^2)$ cũng chưa phải là độ phức tạp tốt nhất - ta có thể đếm số xâu con của một xâu trong $O(n)$ hoặc $O(n \log n)$ nhờ mảng hậu tố (Suffix Array).

Nén xâu

Bài toán

Cho một xâu s độ dài n . Tìm xâu t có độ dài nhỏ nhất sao cho có thể tạo được xâu s bằng cách lặp lại xâu t hữu hạn lần.

Lời giải

Trước tiên, nhận xét là chúng ta chỉ cần tìm độ dài của t , vì khi đó đáp án của bài toán sẽ là tiền tố của s có độ dài này.

Giả sử ta cố định độ dài của t là l (chỉ xét l là ước của n) thì khi đó bài toán đưa về kiểm tra $s[0 \dots l-1] = s[l \dots 2l-1] = \dots = s[n-l, n-1]$ (*) có được thỏa mãn hay không.

Nếu ta dùng thẳng điều kiện (*), với mỗi giá trị của l kiểm tra sẽ cần duyệt qua $O(\frac{n}{l})$ cặp xâu liền kề, mỗi cặp kiểm tra xem chúng bằng nhau không mất độ phức tạp $O(l)$, nên độ phức tạp để kiểm tra khi cố định l là $O(\frac{n}{l}) * O(l) = O(n)$. l chỉ có thể nhận $O(n^{\frac{1}{3}})$ giá trị khác nhau do là ước của n , dẫn tới tổng độ phức tạp là $O(n^{\frac{4}{3}})$. Độ phức tạp này là đủ tốt để AC với $n \leq 10^5$, nhưng ta có thể dùng hàm tiền tố để làm tốt hơn.

Chìa khóa ở đây là thay đổi cách viết (*) thành:

$$s[0 \dots l-1] = s[l \dots 2l-1]s[l \dots 2l-1] = s[2l \dots 3l-1] \dots s[n-2l \dots n-l-1] = s[n-l, n-1]$$

Rồi gộp các xâu ở các vế trái thành một, gộp các xâu ở các vế phải thành một, ta thu được:

$$s[0 \dots n-l-1] = s[l \dots n-1]$$

Hay nói cách khác, l phải thỏa mãn tiên tố độ dài $n - l$ khớp với hậu tố có cùng độ dài của xâu. Kiểm tra điều kiện rút gọn này là đủ vì từ nó ta có thể suy ngược lại ra điều kiện (*). Áp dụng hàm tiền tố: Các tiên tố mà cũng là hậu tố kết thúc tại $n - 1$ có độ dài $\pi[n - 1], \pi[\pi[n - 1] - 1], \pi[\pi[\pi[n - 1] - 1] - 1], \dots$, duyệt qua từng độ dài này và dừng lại khi có một độ dài l thỏa mãn n chia hết cho $n - l$, khi đó đáp án sẽ là $n - l$. Nếu không có độ dài nào thỏa mãn, đáp án khi đó là n . Độ phức tạp của thuật chỉ là $O(n)$.

Thậm chí ta chỉ cần kiểm tra cho tiên tố độ dài $\pi[n - 1]$ là đủ: nếu $k = n - \pi[n - 1]$ là ước của n thì đó chính là đáp án, nếu không thì đáp án là n . Chứng minh:

- Nếu n chia hết cho k , do $\pi[n - 1]$ là tiên tố dài nhất cũng là hậu tố của xâu, k là độ dài ngắn nhất thỏa mãn.
- Nếu n không chia hết cho k , giả sử phản chứng tồn tại $p < n$ sao cho nén được xâu s thành tiên tố độ dài p của xâu. Khi đó ta có $\pi[n - 1] > n - p$, dẫn đến hậu tố độ dài $\pi[n - 1]$ của xâu sẽ chứa một phần của $s[0 \dots p - 1]$.

Xét xâu $s[k \dots (k + p - 1)]$. Do tiên tố và hậu độ dài $\pi[n - 1]$ bằng nhau, $s[k \dots (k + p - 1)] = s[0 \dots p - 1]$, so sánh từng cặp chữ cái tương ứng của hai xâu này, ta rút ra $s_i = s_{(i+k) \bmod p} \forall 0 \leq i < p$.

Đặt $d = \gcd(k, p)$, điều kiện trên khi đó tương đương với $s_i = s_{(i+d) \bmod p} \forall 0 \leq i < p$, nghĩa là $s[0 \dots d - 1]$ là một xâu nén của $s[0 \dots p - 1]$. Do $d \leq k < p$, $s[0 \dots d - 1]$ là một xâu nén của s có độ dài nhỏ hơn p , mâu thuẫn với cách chọn p là xâu nén nhỏ nhất. Ta có điều phải chứng minh.

Ví dụ minh họa:

$$\begin{array}{c} \overbrace{s_0 s_1 s_2 s_3 s_4}^p \quad \overbrace{s_0 s_1 s_2 s_3 s_4}^p \quad \overbrace{s_0 s_1 s_2 s_3 s_4}^p \\ \\ \underbrace{s_0 s_1}_k \quad \overbrace{s_2 s_3 s_4 s_0 s_1 s_2 s_3 s_4 s_0 s_1 s_2 s_3 s_4}_{\pi[15]=13} \end{array}$$

$$s_0 = s_2, s_1 = s_3, s_2 = s_4, s_3 = s_0, s_4 = s_1 \Rightarrow s_0 = s_1 = s_2 = s_3 = s_4 \Rightarrow s_0 \text{ nén được } s$$

Tạo automaton từ hàm tiền tố

Một kỹ thuật đặc trưng đã được sử dụng ở các bài nêu trên là: ghép hai xâu s, t vào nhau bằng một ký tự ngăn cách $\#$ không nằm trong s hoặc t , rồi tính hàm tiền tố cho xâu $s + \# + t$ này.

Việc $\#$ là ký tự không nằm trong cả hai xâu dẫn tới $\pi[i]$ không thể vượt quá $|s|$ với mọi i , qua đó cho phép chúng ta chỉ cần lưu xâu $s + \#$ và các giá trị π tương ứng với xâu này.

$$\underbrace{s_0 s_1 \dots s_{n-1} \#}_{\text{cần lưu}} \quad \underbrace{t_0 t_1 \dots t_{m-1}}_{\text{không cần lưu}}$$

Để tính hàm tiền tố cho các chữ cái sau đó, ta chỉ cần biết chữ cái tiếp theo c trong xâu t và giá trị của hàm tiền tố ở vị trí trước mà không cần lưu lại cả xâu t .

Nói cách khác, ta có thể dựng một **automaton** (máy hữu hạn trạng thái), trong đó các trạng thái là các giá trị có thể của hàm tiền tố hiện tại (từ 0 đến $|s|$), và các bước chuyển trạng thái là các giá trị có thể của chữ cái tiếp theo.

Do đó, kể cả khi không có xâu t thì ta vẫn có thể tính ma trận chuyển trạng thái $(old_\pi, c) \rightarrow new_\pi$ tương tự như khi ta tìm $\pi[i]$ hiện tại, chỉ là ở vị trí i có thể là bất kỳ ký tự nào trong bảng chữ cái:

```

1  void compute_automaton(string s, vector<vector<int>> & aut) {
2      s += '#';
3      int n = s.size();
4      vector<int> pi = prefix_function(s);
5      aut.assign(n, vector<int>(26));
6      for (int i = 0; i < n; i++) {
7          for (int c = 0; c < 26; c++) {
8              int j = i;
9              while (j > 0 && 'a' + c != s[j])
10                 j = pi[j - 1];
11                 if ('a' + c == s[j])
12                     j++;
13                 aut[i][c] = j;
14             }
15         }
16     }

```

Đối với bảng chữ cái tiếng Anh không hoa (26 chữ cái), độ phức tạp của thuật toán trên là $O(n^2 * 26)$.

Một lần nữa, lý do độ phức tạp của thuật toán lớn là vì chưa tận dụng các thông tin đã được tính trước đó. Do đó ta tối ưu bằng quy hoạch động: Khi chúng ta gán j bằng $\pi[j - 1]$, bước chuyển (j, c) dẫn đến cũng trạng thái với bước chuyển $(\pi[j - 1], c)$, và trạng thái ứng với $(\pi[j - 1], c)$ thì đã được tính trước đó, nên với mỗi cặp (j, c) ta tính được trong $O(1)$ và độ phức tạp tổng chỉ là $O(n * 26)$.

```

1  void compute_automaton(string s, vector<vector<int>> & aut) {
2      s += '#';
3      int n = s.size();
4      vector<int> pi = prefix_function(s);
5      aut.assign(n, vector<int>(26));
6      for (int i = 0; i < n; i++) {
7          for (int c = 0; c < 26; c++) {
8              if (i > 0 && 'a' + c != s[i])
9                  aut[i][c] = aut[pi[i - 1]][c];
10             else
11                 aut[i][c] = i + ('a' + c == s[i]);
12         }
13     }
14 }

```

Vậy thì tại sao xây một automaton như vậy lại hữu ích?

Trước tiên, ta cần hiểu mục đích chính của việc tạo xâu $s + \# + t$ là để đếm số lần xâu s xuất hiện trong xâu t .

Do đó, lợi ích gần nhất của automaton này là **tăng tốc độ tính hàm tiền tố** cho xâu $s + \# + t$. Nhờ có automaton mà ta không cần lưu lại xâu s và hàm tiền tố π , vì mọi bước chuyển trạng thái có thể đều đã được tính trong ma trận.

Nhưng còn có một ứng dụng khác, ít hiển nhiên hơn của automaton: giải bài toán so khớp chuỗi khi t là một xâu cực đại được xây theo một quy tắc nào đó. Ví dụ, xâu t có thể là các xâu Gray (định nghĩa ở dưới), hoặc một xâu được định nghĩa theo một công thức đệ quy dựa vào các xâu đầu vào. Để minh họa cụ thể hơn, ta xét bài tập sau:

Bài toán

Cho một số nguyên dương $k \leq 10^5$ và một xâu s có độ dài $\leq 10^5$. Tính số lần s xuất hiện trong xâu Gray thứ k , trong đó các xâu Gray g_i được định nghĩa như sau:

$g_1 = "a"$

$g_2 = "aba"$

$g_3 = "abacaba"$

$g_4 = "abacabadabacaba"$

Lưu ý: chữ cái ở chính giữa g_i không nhất thiết phải là chữ cái thứ i trong bảng chữ cái mà có thể là bất kỳ chữ nào, và chữ cái ở chính giữa của g_i, g_j ($i \neq j$) được phép giống nhau.

Lời giải

Hướng làm của chúng ta vẫn không đổi: tính hàm tiền tố cho $t = g_i$. Viết theo "ngôn ngữ" automaton, ta cần duy trì trạng thái hiện tại khi thêm dần các chữ cái của t từ trái sang phải.

Dựng hẳn xâu t là không thể trong trường hợp này, vì xâu Gray thứ k có độ dài $2^k - 1$, quá lớn so với giới hạn bộ nhớ cho phép.

Tuy vậy, bản chất xâu g là lặp lại chính nó, nên cách trạng thái hiện tại thay đổi trong automaton cũng sẽ có thể tính được dựa vào các bước trước.

Cụ thể hơn, trạng thái của automaton khi duyệt tới cuối g_i có thể được tính nếu chúng ta biết trạng thái của automaton ở đầu g_i .

- Đặt $G[i][j]$ là trạng thái của automaton sau khi duyệt qua tất cả các chữ cái của g_i từ trái sang phải, biết rằng trạng thái lúc ở đầu g_i là j .
- Do cấu trúc $g_i = g_{i-1} + \text{char}(i) + g_{i-1}$, khi duyệt đến hết nửa đầu của g_i (duyet qua các chữ cái của g_{i-1}) thì trạng thái đã chuyển từ j sang $G[i-1][j]$.
- Khi đọc đến $\text{char}(i)$, trạng thái mới dễ dàng tính được bằng ma trận chuyển trạng thái: $G[i-1][j] \rightarrow \text{mid}$, trong đó $\text{mid} = \text{aut}(G[i-1][j], \text{char}(i))$.
- Cuối cùng, ta duyệt qua nửa bên phải của g_i , tương đương với duyệt qua g_{i-1} một lần nữa, nên trạng thái cuối cùng là $G[i-1][\text{mid}]$.

Ta có thể tóm gọn cách trạng thái thay đổi bằng công thức:

$$\text{mid} = \text{aut}[G[i-1][j]][i]$$

$$G[i][j] = G[i-1][\text{mid}]$$

Biết được trạng thái thay đổi như thế nào, ta có thể dùng nó để tính lại đáp án khi đi từ g_{i-1} đến g_i . Để thống nhất với cách ta tính G , đặt $K[i][j]$ là số lần xâu s xuất hiện trong g_i , biết trạng thái ở đầu xâu là j . Khi duyệt

đến giữa xâu, ta đang có $K[i][j] = K[i-1][j]$. Khi trạng thái chuyển thành mid , ta cộng 1 vào $K[i][j]$ nếu $mid = |s|$. Phần còn lại chỉ là $K[i-1][mid]$. Công thức:

$$K[i][j] = K[i-1][j] + (mid == |s|) + K[i-1][mid]$$

Chốt lại, nhờ việc sử dụng automaton mà ta giải được bài toán so khớp trên xâu Gray, và ta có thể dùng lại ý tưởng này cho nhiều xâu khác được tạo bởi các quy tắc phức tạp hơn.

Cụ thể là bài toán: Tìm số lần s xuất hiện trong t_i , biết t_i được tạo bằng cách nối các xâu t_k^{cnt} (lặp lại t_k cnt lần, $k < n$) vào nhau.

Ví dụ một quy tắc như thế:

$$t_1 = \text{"abdeca"}$$

$$t_2 = \text{"abc"} + t_1^{30} + \text{"abd"}$$










$$t_3 = t_2^{50} + t_1^{100}$$

$$t_4 = t_2^{10} + t_3^{100}$$

Ta không thể dựng cả xâu vì độ lớn bùng nổ lên tới 100^{100} chữ cái (!). Tuy nhiên, cách giải cho bài này không khác gì bài trước, đó là xây dựng automaton và tìm quy hoạch động để miêu tả cách trạng thái thay đổi.

Bài tập

Test code KMP: [VNOJ - SUBSTR](#) 

- [Codeforces - Anthem of Berland](#) 
- [Codeforces - MUH and Cube Walls](#) 
- [Codeforces - Prefixes and Suffixes](#) 
- [Codeforces - Password](#) 
- [Codeforces - Lucky Common Subsequence](#) 
- [SPOJ - PSTRING](#) 
- [AtCoder Beginner Contest 257 - Prefix Concatenation](#) 
- [NOI 2014 Problem 4 - Zoo](#) 
- [LightOJ - Unlucky Strings](#) 

Được cung cấp bởi [Wiki.js](#)