Hash: A String Matching Algorithm

Author: Le Khac Minh Tue

1. Giới thiệu:

a. Hoàn cảnh:

Một lớp những bài toán rất được quan tâm trong khoa học máy tính nói chung và lập trình thi cử nói riêng, đó là xử lý xâu chuỗi. Trong lớp bài toán này, người ta thường rất hay phải đối mặt với một bài toán: tìm kiếm xâu chuỗi.

b. Phát biểu bài toán:

i. Cho một đoạn văn bản, gồm m ký tự.
ii. Cho một đoạn mẫu, gồm n ký tự.
iii. Máy tính cần trả lời câu hỏi: đoạn mẫu xuất hiện bao nhiêu lần trong đoạn văn bản và chỉ ra các vị trí xuất hiện đó.

c. Thuật toán:

Có rất nhiều thuật toán có thể giải quyết bài toán này. Người viết xin tóm tắt 2 thuật toán phổ biến được dùng trong các kì thi lập trình:

i. Brute-force approach:

Với một cách tiếp cận trực tiếp, chúng ta có thể thu được thuật toán để giải. Tuy nhiên độ phực tạp của nó là rất lớn trong trường hợp xấu nhất. Thuật toán brute-force so khớp tất cả các vị trí xuất hiện của đoạn mẫu trong đoạn văn bản. Cụ thể độ phức tạp cho thuật toán này là $O(mn)$.

ii. Knuth-Morris-Pratt algorithm

Hay còn được viết tắt là KMP, được phát minh vào năm 1974, bởi Donald Knuth, Vaughan Pratt và James H. Morris. Thuật toán này sử dụng một correction-array, là một thuật toán rất hiệu quả, có độ phức tạp là $O(m + n)$.

d. Mục đích bài viết:

Trong bài viết này, người viết chỉ tập trung vào một thuật toán. Tác giả xin gọi thuật toán này là Hash. Theo như bản thân người viết đánh giá, đây là thuật toán rất hiệu quả đặc biệt là trong thi cử. Nó hiệu quả bởi 3 yếu tố: tốc độ thực thi, linh động trong việc sử dụng (ứng dụng hiệu quả) và sự đơn giản trong cài đặt.

Đầu tiên, người viết xin được trình bày về thuật toán này. Sau đó, người viết sẽ trình bày một vài ứng dụng, cách sử dụng và phát triển thuật toán Hash trong các bài toán tin học.

2. Thuật toán Hash:

a. Ký hiệu:

i. Tập hợp các chữ cái được sử dụng: Σ.

ii. Đoạn văn bản: $T[1..m]$,
iii. Đoạn mẫu: $P[1..n]$,
iv. Đoạn con từ i đến j của một xâu s: $s[i..j]$
v. Chúng ta cần tìm ra tất cả các vị trí i $(1 \le i \le m - n + 1)$ thỏa mãn: $T[i..i + n - 1] = P$.

b. Mô tả thuật toán:

Để đơn giản, giả sử rằng $\Sigma = \{a, b, \dots, z\}$, nghĩa là Σ chỉ gồm các chữ cái Latin in thường. Để biểu diễn một xâu, thay vì dùng chữ cái, chúng ta sẽ chuyển sang biểu diễn dạng số. Ví dụ: xâu $aczd$ được viết dưới dạng số là một dãy gồm 4 số: (0,2,25,3). Như vậy, một xâu được biểu diễn dưới dạng một số ở hệ cơ số 26. Từ đây suy ra, 2 xâu bằng nhau khi và chỉ khi biểu diễn của 2 xâu ở hệ cơ số 10 giống nhau.

Đây chính là tư tưởng của thuật toán: đổi 2 xâu từ hệ cơ số 26 ra hệ cơ số 10, rồi đem so sánh ở hệ cơ số 10. Tuy nhiên, chúng ta nhận thấy rằng, khi đổi 1 xâu ra biểu diễn ở hệ cơ số 10, biểu diễn này có thể rất lớn và nằm ngoài phạm vi lưu trữ số nguyên của máy tính.

Để khắc phục điều này, chúng ta chuyển sang so sánh 2 biểu diễn của 2 xâu ở hệ cơ số 10 sau khi lấy phần dư cho một số nguyên đủ lớn. Cụ thể hơn: nếu biểu diễn trong hệ thập phân của xâu a là x và biểu diễn trong hệ thập phân của xâu b là y, chúng ta sẽ coi a bằng b 'khi và chỉ khi' $x \bmod base = y \bmod base$, trong đó $base$ là một số nguyên đủ lớn.

Dễ dàng nhận thấy việc so sánh $x \bmod base$ với $y \bmod base$ rồi kết luận a có bằng với b hay không là sai. $x \bmod base = y \bmod base$ chỉ là điều kiện cần để a bằng b chứ chưa phải điều kiện đủ. Tuy nhiên, chúng ta sẽ chấp nhận lập luận sai này trong thuật toán Hash. Và coi điều kiện cần như điều kiện đủ. Trên thực tế, lập luận sai này có những lúc dẫn đến so sánh xâu không chính xác và chương trình bị chạy ra kết quả sai. Nhưng cũng thực tế cho thấy rằng, khi chọn $base$ là một số nguyên lớn, số lượng những trường hợp sai rất ít, và ta có thể coi Hash là một thuật toán chính xác.

Để đơn giản trong việc trình bày tiếp thuật toán, chúng ta sẽ gọi biểu diễn của một xâu trong hệ thập phân sau khi lấy phần dư cho $base$ là mã Hash của xâu đó. Nhắc lại, 2 xâu bằng nhau 'khi và chỉ khi' mã Hash của 2 xâu bằng nhau.

Trở lại bài toán ban đầu, chúng ta cần chỉ ra P xuất hiện ở những vị trí nào trong T. Để làm được việc này, chúng ta chỉ cần duyệt qua mọi vị trí xuất phát có thể của P trong T. Giả sử vị trí đó là i, chúng ta sẽ kiểm tra $T[i..i + n - 1]$ có bằng với P hay không. Để kiểm tra điều này, chúng ta cần tính được mã Hash của đoạn $T[i..i + n - 1]$ và mã Hash của xâu P.

Để tính mã Hash của xâu P chúng ta chỉ cần làm đơn giản như sau:

```
hashP = 0
for (i : 1 .. n)
      hashP = (hashP * 26 + P[i] - 'a') mod base
```

Phần khó hơn của thuật toán Hash là: Tính mã Hash của một đoạn con từ i đến j $T[i..j]$ của xâu T $(1 \leq i \leq j \leq m)$.

Để hình dung cho đơn giản, xét ví dụ sau: Xét xâu s và biểu diễn của nó dưới cơ số 26: (4,1,2,5,1,7,8). Chúng ta cần lấy mã Hash của đoạn con từ phần tử thứ 3 đến phần tử thứ 6, nghĩa là cần lấy mã Hash của xâu (2,5,1,7). Nhận thấy, để lấy được xâu $s[3..6]$, chỉ cần lấy số $s[1..6]$ là (4,1,2,5,1,7) trừ cho số ($s[1..2]$ nối thêm (0,0,0,0)) là (4,1,0,0,0,0) ta sẽ thu được (2,5,1,7). Tương tự, để lấy được mã Hash của xâu $s[3..6]$, chỉ cần lấy mã Hash của $s[1..6]$trừ đi (mã Hash của $s[1..2]$ nhân với 26^4).

Để cài đặt ý tưởng này, chúng ta cần khởi tạo $26^x \ mod \ base$ $(0 \leq x \leq m)$ và mã Hash của tất cả những tiền tố của s, cụ thể là mã Hash của những xâu $s[1..i]$ $(1 \leq i \leq m)$.

```
pow[0] = 1
for (i : 1 .. m)
      pow[i] = (pow[i-1] * 26) mod base

hashT[0] = 0
for (i : 1 .. m)
      hashT[i] = (hashT[i-1] * 26 + T[i] - 'a') mod base
```

Trên đoạn code trên, chúng ta thu được mảng $pow[i]$ (lưu lại $26^i \ mod \ base$) và mảng $hashT[i]$ (lưu lại mã Hash của $T[1..i]$).

Để lấy mã Hash của $T[i..j]$ ta viết hàm sau:

```
      function getHashT(i, j):
return (hashT[j] - hashT[i - 1] * pow[j - i + 1] + base * base) mod base
```

Bài toán chính đã được giải quyết, và đây là chương trình chính:

```
for (i : 1 .. m - n +1)
      if hashP = getHashT(i, i + n - 1):
            print("Match position: ", i)
```

c. Mã chương trình:

Chương trình sau, tôi viết bằng ngôn ngữ C++, là lời giải cho bài $SUBSTR$ trên hệ thống chấm bài trực tuyến VOJ.

```
#include <iostream>
#include <cstdio>
#include <cstring>
#define FOR(i,a,b) for(int i=a;i<=b;i++)
#define base 1000000003LL
#define ll long long
#define maxn 1000111
using namespace std;

ll POW[maxn],hashT[maxn];
```

```
ll getHashT(int i,int j) {
	return (hashT[j]-hashT[i-1]*POW[j-i+1]+base*base)%base;
}

int main() {
	string T,P;
	cin >> T >> P;
	int m=T.size(),n=P.size();
	T=" "+T;P=" "+P;
	POW[0]=1;
	FOR(i,1,m) POW[i]=(POW[i-1]*26) % base;
	FOR(i,1,m) hashT[i]=(hashT[i-1]*26+T[i]-'a') % base;
	ll hashP=0;
	FOR(i,1,n) hashP=(hashP*26+P[i]-'a') % base;
	FOR(i,1,m-n+1) if(hashP==getHashT(i,i+n-1)) printf("%d ",i);
}
```

d. Đánh giá:

Độ phức tạp của thuật toán là $O(m+n)$. Nhưng điều quan trọng là: chúng ta có thể kiểm tra 2 xâu có giống nhau hay không trong $O(1)$. Đây là điều tạo nên sự linh động cho thuật toán Hash. Ngoài sự linh động và tốc độ thực thi, chúng ta có thể thấy cài đặt thuật toán này thực sự rất đơn giản nếu so với các thuật toán xử lý xâu khác.

3. Ứng dụng:

Như đã đề cập ở trên, thuật toán này sẽ có trường hợp chạy sai. Tất nhiên, bên cạnh việc sử dụng Hash, còn có nhiều thuật toán xử lý xâu chuỗi khác, mang lại sự chính xác tuyệt đối. Tôi tạm gọi những thuật toán đó là 'thuật toán chuẩn'. Việc cài đặt 'thuật toán chuẩn' có thể mang lại một tốc độ chạy chương trình cao hơn, độ chính xác của chương trình lớn hơn. Tuy nhiên, người làm bài sẽ phải trả giá là sự phức tạp khi cài đặt các 'thuật toán chuẩn' đó.

Sử dụng Hash không chỉ giúp người làm bài dễ dàng cài đặt hơn mà quan trọng ở chỗ: Hash có thể làm được những việc mà 'thuật toán chuẩn' không làm được. Sau đây, tôi sẽ xét một vài ví dụ để chứng minh điều này.

a. Longest palindrome substring

Bài toán đặt ra như sau: Bạn được cho một xâu s độ dài n ($n \leq 50\,000$). Bạn cần tìm độ dài của xâu đối xứng dài nhất gồm các kí tự liên tiếp trong s. (Xâu đối xứng là xâu đọc từ 2 chiều giống nhau).

Một 'thuật toán chuẩn' không thể áp dụng vào bài toán này đó là thuật toán KMP. Ngoài KMP ra, có 2 'thuật toán chuẩn' có thể áp dụng được. Thuật toán thứ nhất đó là sử dụng thuật toán Manachar để tính bán kính đối xứng tại tất cả vị trí trong xâu. Thuật toán thứ 2 đó là sử dụng Suffix Array và LCP (Longest Common Prefix) cho xâu được nối bởi s và xâu s

viết theo thứ tự ngược lại. 2 thuật toán này đều không dễ dàng để cài đặt, và nằm ngoài phạm vi bài viết, nên tôi chỉ nêu sơ qua mà không đi vào chi tiết.

Bây giờ, chúng ta sẽ xét thuật toán 'không chuẩn' là thuật toán Hash. Để đơn giản, chúng ta xét trường hợp độ dài của xâu đối xứng là lẻ (trường hợp chẵn xử lý hoàn toàn tương tự). Giả sử xâu đối xứng độ dài lẻ dài nhất có độ dài là l. Dễ thấy, trong xâu s tồn tại xâu đối xứng độ dài $l-2, l-4, \ldots$ Tuy nhiên, xâu s không tồn tài xâu đối xứng độ dài $l+2, l+4, \ldots$ Như vậy, l thỏa mãn tính chất chia nhị phân. Chúng ta sẽ chia nhị phân để tìm độ dài lớn nhất có thể. Với mỗi độ dài l, chúng ta cần kiểm tra xem trong xâu có tồn tại một xâu con là xâu đối xứng độ dài l hay không. Để làm việc này, ta duyệt qua tất cả tất cả các xâu con độ dài l trong s.

Bài toán còn lại là: kiểm tra xem $s[i..j](1 \le i \le j \le n; (j-i+1)\ mod\ 2 = 1)$ có phải là xâu đối xứng hay không.

Cách làm như sau. Gọi t là xâu s viết theo thứ tự ngược lại. Bằng thuật toán Hash, chúng ta có thể kiểm tra được một xâu con nào đó của t có bằng một xâu con nào đó của s hay không. Như vậy, chúng ta cần kiểm tra $s[i..k]$ có bằng $t[n-j+1..n-k+1]$ hay không với k là tâm đối xứng, nói cách khác $k = \frac{i+j}{2}$. Như vậy bài toán đã được giải. Độ phức tạp cho cách làm này là $O(n\log(n))$.

b. k-th alphabetical cyclic

Bài toán đặt ra như sau: Bạn được cho một dãy $a_1, a_2, \ldots, a_n$ ($n \le 50\ 000$). Sắp xếp n hoán vị vòng quanh của dãy này theo thứ tự từ điển. Cụ thể, các hoán vị vòng quanh của dãy này là $(a_1, a_2, \ldots, a_n)$, $(a_2, a_3, \ldots, a_n, a_1)$, $(a_3, a_4, \ldots, a_n, a_1, a_2)$,... Dãy này có thứ tự từ điển nhỏ hơn dãy kia nếu số đầu tiên khác nhau của dãy này nhỏ hơn dãy kia. Yêu cầu bài toán là: In ra dãy có thứ tự từ điển lớn thứ k.

Nếu tiếp cận một cách trực tiếp, chúng ta sẽ sinh ra tất cả các dãy hoán vị vòng quanh, rồi sau đó dùng một thuật toán sắp xếp để sắp xếp lại chúng theo thứ tự từ điển, cuối cùng chỉ việc in ra dãy thứ k sau khi sắp xếp. Tuy nhiên độ phức tạp của thuật toán này là rất lớn và không thể đáp ứng được yêu cầu về thời gian. Cụ thể, cách này có độ phức tạp là $O(n^2 * \log(n))$, đây là tích của độ phức tạp của sắp xếp và độ phức tạp của mỗi phép so sánh dãy.

Vẫn giữ tư tưởng là sắp xếp lại tất cả các dãy hoán vị vòng quanh rồi in ra dãy đứng ở vị trí thứ k, chúng ta cố gắng cải tiến độ phức tạp của việc so sánh thứ tự từ điển của 2 dãy.

Nhắc lại định nghĩa về thứ tự từ điển của 2 dãy: Xét 2 dãy a và b có cùng số phần tử. Gọi ví trí thứ i là vị trí đầu tiên từ trái sang mà $a_i \ne b_i$. $a < b \leftrightarrow a_i < b_i$. Như vậy, ta phải tìm đoạn tiền tố giống nhau dài nhất của a và b, rồi so sánh kí tự tiếp theo. Để tìm được đoạn tiền tố giống nhau dài nhất, ta có thể sử dụng Hash kết hợp với chia nhị phân.

Để giải được bài này, cần sử dụng thêm một kỹ thuật nhỏ nữa: Thay vì sinh ra tất cả các hoán vị vòng quanh, chúng ta chỉ cần nhân đôi dãy a lên, dãy mới sẽ có $2 * n$ phần tử $(a_1, a_2, \dots, a_n, a_1, a_2, \dots, a_n)$. Một hoán vị vòng quanh sẽ là một dãy con liên tiếp độ dài n của dãy nhân đôi này.

c. Longest substring and appear at least k times

Bài toán đặt ra như sau: Bạn được cho xâu s độ dài n ($n \leq 50000$), bạn cần tìm ra xâu con của s có độ dài lớn nhất, và xâu con này xuất hiện ít nhất k lần.

Bài toán này có thể được giải bằng Suffix Array, tuy nhiên cách cài đặt phức tạp và không phải trọng tâm của bài viết nên tôi sẽ không nêu ra ở đây.

Tiếp tục bàn đến thuật toán Hash để thay thế thuật toán chuẩn. Nhận xét rằng, giả sử độ dài lớn nhất tìm được là l, thì với mọi $l' \leq l$, luôn tồn tại xâu có độ dài l' xuất hiện ít nhất k lần. Tuy nhiên, với mọi $l' > l$, không tồn tại xâu có độ dài l' xuất hiện ít nhất k lần (do l đã là lớn nhất). Như vậy, l thỏa mãn tính chất chia nhị phân. Chúng ta có thể áp dụng thuật toán tìm kiếm nhị phân để tìm ra l lớn nhất.

Bây giờ, với mỗi l khi đang chia nhị phân, chúng ta sẽ phải kiểm tra liệu có tồn tại xâu con nào xuất hiện ít nhất k lần hay không. Điều này được làm rất đơn giản, bằng cách sinh mọi mã Hash của các xâu con độ dài l trong s. Sau đó sắp xếp lại các mã Hash này theo chiều tăng dần, rồi kiểm tra xem có một đoạn liên tiếp các mã Hash nào giống nhau độ dài k hay không.

Như vậy, độ phức tạp để chia nhị phân là $O(\log(n))$, độ phức tạp của sắp xếp là $O(n \log(n))$, vậy độ phức tạp của cả bài toán là $O(n \log(n)^2)$.

4. Tổng kết:

a. Thuật toán:

Ý tưởng thuật toán Hash dựa trên việc đổi từ hệ cơ số lớn sang hệ thập phân, so sánh hai số thập phân lớn bằng cách so sánh phần dư của chúng với một số đủ lớn.

b. Ưu điểm:

Ưu điểm của thuật toán Hash là cài đặt rất dễ dàng. Linh động trong ứng dụng và có thể thay thế các thuật toán chuẩn 'hầm hố' khác.

c. Nhược điểm:

Nhược điểm của thuật toán Hash là tính chính xác. Mặc dù rất khó sinh test để có thể làm cho thuật toán chạy sai, nhưng không phải là không thể. Vì vậy, để nâng cao tính chính xác của thuật toán, người ta thường dùng nhiều modulo khác nhau để so sánh mã Hash (ví dụ như dùng 3 modulo một lúc).

5. **Các nguồn tham khảo:**

   - http://en.wikipedia.org/wiki/String_searching_algorithm
   - http://en.wikipedia.org/wiki/Knuth%E2%80%93Morris%E2%80%93Pratt_algorithm
   - http://en.wikipedia.org/wiki/Rabin-Karp_string_search_algorithm
   - http://vn.spoj.com/problems/SUBSTR/
   - http://vn.spoj.com/problems/PALINY/
   - http://acm.sgu.ru/problem.php?contest=0&problem=426
   - http://vn.spoj.com/problems/DTKSUB/
   - http://en.wikipedia.org/wiki/Alphabetical_order