

Hash: A String Matching Algorithm

Hash: A String Matching Algorithm

Tác giả: Lê Khắc Minh Tuệ

Chỉnh sửa: Nguyễn RR Thành Trung, Phạm Văn Hạnh

Giới thiệu

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 chuỗi](#). Trong lớp bài toán này, người ta thường rất hay phải đối mặt với một bài toán: tìm kiếm chuỗi.

Phát biểu bài toán

- ▶ Cho một đoạn văn bản, gồm m ký tự.
- ▶ Cho một đoạn mẫu, gồm n ký tự.
- ▶ 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.

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:

- ▶ **Brute-force:** Với một cách tiếp cận trực tiếp, chúng ta có thể thu được thuật toán để giải. Tuy nhiên độ phức 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 $O(mn)$.
- ▶ **Knuth-Morris-Pratt:** 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 $O(m + n)$.

Mục đích bài viết

Trong bài viết này, người viết chỉ tập trung vào thuật toán Hash (Tên chuẩn của thuật toán này là [Rabin-Karp](#) ☑ hoặc [Rolling Hash](#) ☑, tuy nhiên ở Việt Nam thường được gọi 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.

Thuật toán Hash

Ký hiệu

- Tập hợp các chữ cái được sử dụng: Σ
- Đoạn con từ i đến j của một chuỗi s : $s[i..j]$
- Đoạn văn bản: $T[1..m]$
- Đoạn mẫu: $P[1..n]$

Chúng ta cần tìm ra tất cả các vị trí i ($1 \leq i \leq m - n + 1$) thỏa mãn: $T[i..i + n - 1] = P$.

Mô tả thuật toán

Để đơn giản, giả sử rằng $\Sigma = a, b, \dots, z$ (nói cách khác, Σ chỉ gồm các chữ cái in thường). Để biểu diễn một chuỗi, thay vì dùng chữ cái, chúng ta sẽ chuyển sang biểu diễn dạng số. Ví dụ: chuỗi **aczd** được viết dưới dạng số là một dãy gồm 4 số: **(1, 3, 26, 4)**. Như vậy, một chuỗi được biểu diễn dưới dạng một số ở hệ cơ số $base$ với $base > 26$. Từ đây suy ra, 2 chuỗi bằng nhau khi và chỉ khi biểu diễn của 2 chuỗi ở hệ cơ số 10 giống nhau.

Lưu ý:

- Ở đây mình đổi chữ **a** thành số 1 chứ không phải số 0. Đây là chi tiết vô cùng quan trọng, để tránh 2 chuỗi: **abc** và **bc** bằng nhau khi đổi ra số. Bạn có thể đọc thêm chi tiết ở phần [Chi tiết cài đặt](#).
- Thông thường ta chọn $base$ là một số nguyên tố. Mình sẽ giải thích thêm trong phần [Chi tiết cài đặt](#).

Đây chính là tư tưởng của thuật toán: đổi 2 chuỗi từ hệ cơ số $base$ ra hệ cơ số 10, rồi đem so sánh ở hệ cơ số 10. Tuy nhiên, chúng ta nhận thấy rằng, khi đổi 1 chuỗi 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 chuỗi 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 chuỗi a là x và biểu diễn trong hệ thập phân của chuỗi b là y , chúng ta sẽ coi a bằng b 'khi và chỉ khi' $x \bmod MOD = y \bmod MOD$ trong đó MOD là một số nguyên đủ lớn.

Lưu ý: Lý do chọn MOD là số nguyên tố được giải thích thêm trong phần [Chi tiết cài đặt](#).

Dễ dàng nhận thấy việc so sánh $x \bmod MOD$ với $y \bmod MOD$ rồi kết luận a có bằng với b hay không là sai. $x \bmod MOD = y \bmod MOD$ 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 thể dẫn đến kết quả sai nếu bạn không hiểu rõ mình đang làm gì. Để hiểu rõ về tỉ lệ sai của thuật toán Hash, các bạn đọc thêm phần [Đánh giá độ chính xác](#). Phần [Chi tiết cài đặt](#) cũng nói thêm về cách tránh bị sai số khi cài đặt Hash.

Để đơn giản trong việc trình bày tiếp thuật toán, chúng ta sẽ gọi biểu diễn của một chuỗi trong hệ thập phân sau khi lấy phần dư cho MOD là mã Hash của chuỗi đó. Nhắc lại, 2 chuỗi 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:

```

1 | const base = 31;
2 | hashP = 0
3 | for (i : 1 .. n)
4 |     hashP = (hashP * base + P[i] - 'a' + 1) mod MOD

```

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

- Để hình dung cho đơn giản, xét ví dụ sau: Xét xâu s và biểu diễn của nó dưới cơ số $base$: (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 $(s[1..2]$ nhân với $base^4$) là (4, 1, 0, 0, 0, 0) ta sẽ thu được (2, 5, 1, 7).
- Để cài đặt ý tưởng này, chúng ta cần khởi tạo $base^x \bmod MOD$ với ($0 \leq x \leq m$) và mã Hash của tất cả những tiền tố của s , cụ thể là mã Hash của những xâu $s[1..i]$ với ($1 \leq i \leq m$).

```

1 | pow[0] = 1
2 | for (i : 1 .. m)
3 |     pow[i] = (pow[i-1] * base) mod MOD
4 |
5 |
6 | hashT[0] = 0
7 | for (i : 1 .. m)
8 |     hashT[i] = (hashT[i-1] * base + T[i] - 'a') mod MOD

```

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

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

```

1 | function getHashT(i, j):
2 |     // Chú ý rằng `hashT[i-1] * pow[j-i+1]` có thể âm.
3 |     // Với 1 số ngôn ngữ như C++, toán tử mod sẽ trả kết quả sai với số âm.
4 |     // Do đó ta cần thêm "+ MOD * MOD" để đảm bảo kết quả luôn chính xác.
5 |     return (hashT[j] - hashT[i-1] * pow[j-i+1] + MOD * MOD) mod MOD

```


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

```

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

```

Mã chương trình

Chương trình sau, tôi viết bằng ngôn ngữ C++, là lời giải cho bài [SUBSTR](#)  :

```

1 | typedef long long ll;
2 |
3 | const int base = 31;
4 | const ll MOD = 1000000003;
5 | const ll maxn = 1000111;
6 |
7 | using namespace std;
8 |
9 | ll POW[maxn], hashT[maxn];
10 |
11 |
12 | ll getHashT(int i, int j) {
13 |     return (hashT[j] - hashT[i - 1] * POW[j - i + 1] + MOD * MOD) % MOD;
14 | }
15 |
16 |
17 | int main() {
18 |     // Input
19 |     string T, P;
20 |     cin >> T >> P;
21 |
22 |     // Initialize
23 |     int lenT = T.size(), lenP = P.size();
24 |     T = " " + T;
25 |     P = " " + P;
26 |     POW[0] = 1;
27 |
28 |     // Precalculate base^i
29 |     for (int i = 1; i <= lenT; i++)
30 |         POW[i] = (POW[i - 1] * base) % MOD;
31 |
32 |     // Calculate hash value of T[1..i]
33 |     for (int i = 1; i <= lenT; i++)
34 |         hashT[i] = (hashT[i - 1] * base + T[i] - 'a' + 1) % MOD;
35 |
36 |     // Calculate hash value of P
37 |     ll hashP=0;
38 |     for (int i = 1; i <= lenP; i++)
39 |         hashP = (hashP * base + P[i] - 'a' + 1) % MOD;
40 |
41 |     // Finding substrings of T equal to string P

```

```

42 |         for (int i = 1; i <= lenT - lenP + 1; i++)
43 |             if (hashP == getHashT(i, i + lenP - 1))
44 |                 printf("%d ", i);
45 |     }

```

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.

Chi tiết cài đặt

Trong thuật toán hash, có hai yếu tố cần quan tâm là hệ cơ số (base) và modulo (mod).

1. Chọn số nguyên tố cho hệ cơ số và modulo

Ý tưởng của thuật toán Hash là dựa trên một ngộ nhận sai lầm nhưng xảy ra sai sót với xác suất vô cùng nhỏ: $a \bmod M = b \bmod M \Leftrightarrow a = b$. Để xác suất xảy ra sai là $1/M$ cho một truy vấn, các bạn cần chọn hệ cơ số và modulo thỏa mãn đồng thời:

- $base$ là số nguyên tố lớn hơn các chữ cái của xâu S .
- MOD là số nguyên tố.

Phần chứng minh sai số bạn có thể đọc thêm trong [blog rng_58](#) , tuy nhiên phần chứng minh rất phức tạp nên mình sẽ không trình bày ở đây.


2. Chọn hệ cơ số

Mình khuyến khích các bạn chọn hệ cơ số > 256 (Mình thường chọn là số nguyên tố 311). Nếu bạn chọn hệ cơ số là 31, bạn chỉ làm việc với xâu gồm toàn các ký tự in thường, và phải "mã hóa" các ký tự từ **a** đến **z** thành các số từ 1 đến 26. Điều này khiến code của bạn bị dài. Nếu bài toán cho xâu có các ký tự 'A'...'Z', 'a'...'z' và '0'...'9', việc bạn mã hóa chúng thành các số từ 1 đến 64 là phức tạp và không cần thiết.

Chưa kể, nếu bạn quên mất không **+1** và mã hoá **a** thành **0** là hành động tự treo cổ vì rất dễ bị hack.


Nếu bạn chọn hệ cơ số > 256 , bạn chỉ cần dùng mã ASCII của các ký tự là xong, và lại tránh bị hack.

3. Chọn modulo

Nếu bạn không hiểu rõ về cách đánh giá độ chính xác của thuật Hash (trình bày ở mục [Đánh giá độ chính xác](#)), bạn chỉ cần chọn 3-4 số nguyên tố khác nhau làm MOD . Bạn có thể tham khảo [code của Phạm Văn Hạnh](#) . Tuy nhiên các bạn cũng nên lưu ý là dùng nhiều MOD quá cũng làm chương trình chạy chậm đi.

4. Hash tràn số và Hash có MOD

Trên thực tế, khi cài đặt Hash sử dụng nhiều phép `mod` sẽ làm chương trình chạy chậm. Vì vậy, để tăng tốc độ, người ta thường cài đặt với $MOD = 2^{64}$. Do đó, nếu sử dụng kiểu dữ liệu số 64-bit, ta không cần dùng phép `mod` mà cứ để các phép tính tràn số. Kỹ thuật này được gọi là Hash tràn số. Tuy nhiên khi cài đặt như vậy có một vài chú ý:

- Việc sử dụng MOD không phải là số nguyên tố (và hơn nữa lại là 1 số cố định) khiến cho hàm Hash không đủ tốt. Nếu test được sinh ngẫu nhiên, thì nó không có vấn đề gì cả. Nhưng ở trên Codeforces, vì những người thi cũng có thể "hack" code của bạn bằng test tự sinh, nên bạn hầu như không thể AC các bài Hash với Hash tràn số. Bạn có thể đọc thêm về cách sinh test giết Hash tràn số [ở đây](#) . Cách giải quyết là dùng hash tràn số kết hợp với một MOD khác.
- Nếu dùng Pascal, cần tắt báo tràn số (`$Q-`), nếu không chương trình sẽ chạy bị lỗi.

5. Một số lời khuyên nho nhỏ

Chỉ so sánh mã hash của hai xâu có cùng độ dài. Hiển nhiên, hai xâu ký tự không cùng độ dài thì không bằng nhau. Điều này có thể giảm xác suất rủi ro khi hash một modulo đáng kể.

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.

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 ký 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 Manacher 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, 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 $l - 2, l - 4, \dots$. Tuy nhiên, xâu s không tồn tại xâu đối xứng độ dài $l + 2, l + 4, \dots$. Như vậy, s 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 $s[i..j]$ với $(1 \leq i \leq j \leq m; (j - i + 1) \bmod 2 = 1)$ 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 $k = (i + j)/2$. Như vậy bài toán đã được giải. Độ phức tạp cho cách làm này là $O(n \log(n))$.

k-th alphabetical cyclic

Bài toán đặt ra như sau: Bạn được cho một dãy $a_1, a_2, \dots, a_n (n \leq 50,000)$. Sắp xếp n hoán vị vòng tròn của dãy này theo thứ tự từ điển. Cụ thể, các hoán vị vòng quanh của dãy này là (a_1, a_2, \dots, a_n) , $(a_2, a_3, \dots, a_n, a_1)$, $(a_3, a_4, \dots, a_n, a_1, a_2)$,... 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 .

- ▶ 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.
- ▶ 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 vị trí thứ i là vị trí đầu tiên từ trái sang mà $a_i \neq b_i$. $a < b \Leftrightarrow a_i < b_i$. 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 $2n$ 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.

Từ đó ta thu được thuật toán với độ phức tạp $O(n * \log^2(n))$

Longest substring and appear at least k times

Bài toán đặt ra như sau: Bạn được cho xâu s độ dài $n (n \leq 50,000)$, 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

k 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 l 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 $O(n\log^2(n))$.

Đánh giá độ chính xác

Thông thường, khi sử dụng Hash, ta thường gặp phải 2 trường hợp như sau:

1. Trả lời Q truy vấn, mỗi truy vấn có dạng kiểm tra 2 đoạn con của 2 xâu có bằng nhau hay không
2. Cho N xâu, kiểm tra tất cả các xâu có phân biệt hay không.

Giả sử ta chọn MOD là một số nguyên tố khoảng 10^9 , và giả sử dữ liệu được sinh ngẫu nhiên và hàm hash của chúng ta đủ tốt để Hash của các xâu được phân bố đều và ngẫu nhiên.


Trường hợp 1

Giả sử Q khoảng 10^5 , và bộ test có $T = 100$ test.

- Với 2 xâu khác nhau, xác suất để nó có cùng Hash là xấp xỉ $1/10^9$. Như vậy, xác suất để trả lời đúng 1 truy vấn là: $1 - 1/10^9$.
- Ở trường hợp xấu nhất, ta có Q truy vấn mà mỗi truy vấn là một cặp xâu khác nhau. Xác suất để ta trả lời đúng tất cả các truy vấn là: $(1 - 1/10^9)^Q$.
- Xác suất để ta trả lời đúng tất cả các truy vấn của tất cả các test là: $(1 - 1/10^9)^{Q \cdot T}$.

Thay số vào, xác suất để trả lời đúng tất cả các truy vấn là 0.9900, đủ lớn để ta yên tâm qua tất cả các test, với điều kiện test không được sinh dựa trên P . (Chú ý nếu bạn đang thi những contest như Topcoder/Codeforces, người khác có thể đọc được P của bạn và sinh test để challenge code của bạn).

Trường hợp 2

Theo [Birthday Paradox](#) , ta dễ dàng thấy rằng, nếu có $\sqrt{P} = 3 \cdot 10^4$ xâu, xác suất để 2 xâu bằng nhau là rất lớn. Thật vậy, xác suất để tất cả các xâu khác nhau là:

$$(1 - 1/10^9) \cdot (1 - 2/10^9) \cdot (1 - 3/10^9) \cdot \dots \cdot (1 - N/10^9).$$

Với $N = 30,000$, tích trên là 0.6376, nghĩa là bạn có gần 0.40 xác suất trả lời sai. Do vậy, bạn bắt buộc phải dùng nhiều MOD khác nhau.

Tổng kết

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.

Cài đặt

Khi cài đặt Hash, ta cần chọn:

- Hệ cơ số
- MOD

Với những trang web mà người khác có thể đọc code bạn rồi tìm test sai (như Codeforces, Topcoder), nếu Hash tràn số hoặc MOD là 1 số nguyên biết trước, có thể sinh test để làm code bạn sai. Với những kỳ thi như HSG QG, IOI, ACM, và các Online Judge, thông thường sẽ không có những test như vậy. Tóm lại, khi chọn *MOD* và hệ cơ số ta làm như sau:

- Chọn hệ cơ số là số nguyên tố lớn hơn số lượng ký tự (ví dụ 31, 71).
- Chọn *MOD* đủ lớn theo nguyên tắc đã phân tích ở phần [Đánh giá độ chính xác](#).
- Nếu là Topcoder hoặc Codeforces, dùng kết hợp Hash tràn số và 1 *MOD* đủ lớn.
- Nếu không phải Topcoder hay Codeforces, có thể dùng Hash tràn số. Nếu $MOD = 2^{64}$ chưa đủ lớn thì dùng thêm 1 *MOD* nữa.

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 'hàm hồ' khác.



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).

Bài tập áp dụng

- [VNOJ - SUBSTR](#) 
- [VNOJ - PALINY](#) 
- [VNOJ - DTKSUB](#) 
- [VNOJ - DTCSTR](#) 
- [VNOJ - TWOOPERS](#) 
- [VNOJ - VOSTR](#) 
- [SGU 426](#) 

Các nguồn tham khảo

- [Wikipedia - String Searching Algorithm](#) 
- [Wikipedia - KMP](#) 

- [Wikipedia - Rabin Karp](#) 
- [Wikipedia - Alphabetical Order](#) 

Được cung cấp bởi [Wiki.js](#)