From random to AC

Kiểm tra số chính phương bằng phương pháp random

Ngày 18 tháng 8 năm 2020

Lê Duy Thức - Cá Nóc Cắn Cáp

Lời tựa

Mình là Lê Duy Thức, cựu học sinh trường THPT chuyên Hùng Vương, Bình Dương khoá Toán 16-19, hiện tại là sinh viên trường Đại học Khoa học Tự nhiên. Đây là lần đầu tiên mình viết một tài liệu nên không thể tránh khỏi sai sót, rất mong nhận được ý kiến đóng góp từ các bạn. Mọi ý kiến có thể gửi về địa chỉ mail: **leduykhongngu@gmail.com**, hoặc gửi tin nhắn tới facebook cá nhân của mình: **https://www.facebook.com/leduy.khongngu**.

Bài toán kiểm tra một số có là số chính phương hay không gần như là một bài tập vỡ lòng dành cho những người vừa học lập trình thi đấu, nó là một bài toán dễ khi mà giới hạn của số không lớn (vẫn lưu trữ được dưới dạng số nguyên 64 bit), nhưng khi giới hạn lớn hơn nữa thì bài toán không còn dễ dàng gì. Trong tài liệu này mình sẽ giới thiệu một vài cách giải bài toán này, sau cùng sẽ đi tới cách giải bằng random.

Mục lục

1 Bài toán kiểm tra số chính phương

1.1 Định nghĩa

Định nghĩa 1.1

Số n được gọi là số chính phương khi và chỉ khi tồn tại số $x \in \mathbb{Z}$ sao cho $x^2 = n$.

Định nghĩa 1.2

Bài toán kiểm tra số chính phương là bài toán kiểm tra một số cho trước có phải là một số chính phương hay không. Input là một số tự nhiên n, output là 0 - tương ứng với n không là số chính phương, 1 tương ứng n là số chính phương.

Bài toán kiểm tra số chính phương có thể mô hình hoá thành việc tìm một hàm số f từ $\mathbb{N} \to \{0; 1\}$:

$$f(x) = \begin{cases} 0, & \text{nếu } x \text{ không là số chính phương} \\ 1, & \text{nếu } x \text{ là số chính phương} \end{cases}$$

1.2 Một số cách giải bài toán kiểm tra số chính phương

Mình sẽ giới thiệu một số cách kiểm tra số chính phương từ cơ bản cho tới phức tạp, sau cùng sẽ là giới thiệu cách dùng random, vì thế các bạn có thể bỏ qua phần này mà tiến hẳn phần sau.

1.2.1 Thuật toán trâu

Cách làm thuật này là duyệt qua mọi số tự nhiên i không vượt quá $\sqrt{n}$ và kiểm tra xem $i * i$ có bằng n hay không. Độ phức tạp của thuật toán này là $O(\sqrt{n})$. Dưới đây là một phần code của cách này.

```
bool F1(int64_t n)
{
   for (int i=0; 1ll * i * i <= n; ++i)
      if (1ll * i * i == n) return 1;
   return 0;
}
```

1.2.2 Thuật toán xịn hơn

Cách nếu ta đặt $x = \lfloor \sqrt{n} \rfloor$, thì chỉ cần kiểm tra xem tích $x * x$ có bằng n hay không là được. Độ phức tạp của thuật toán này là tổng độ phức tạp của việc tính căn bậc 2 của n cùng với việc tính tích 2 số.

```
bool F2(int64_t n)
{
   int64_t x = sqrt(n);
   return x * x == n;
```

```
}
```

Nếu n nhỏ (trong phạm vi số nguyên 64bit) thì thuật toán trên có thể xem là $O(1)$, nhưng khi n có giới hạn lớn hơn, vượt qua phạm vi lưu trữ của số nguyên 64bit thì việc cài đặt thuật toán trên không còn dễ dàng, và tính toán cũng tốn độ phức tạp hơn.
Khi n lớn, nếu làm việc trên C++ và không có thư viện bignum mạnh mẽ như Java hay Python thì việc tính toán căn bậc không thể chỉ gọi hàm như vậy mà phải tự tay cài hàm tính căn bậc 2.
Một cách cài khả thi đó là dùng cách chặt nhị phân giá trị căn bậc 2 của n, nhưng việc cài cực kì khó khăn vì ta phải code thư viện bignum, nhân, chia số lớn ... và độ phức tạp cũng cực kì lớn. Mình sẽ không đi sâu vào việc cài đặt của cách này khi n lớn, các bạn có thể google với keyword **find square root by binary search**.
Một cách khác để tìm căn bậc 2 của một số n có thể làm đó là giải phương trình $x^2 - n = 0$, cách giải phương trình này có thể dùng **Newton's method** để giải (nhưng vẫn phải dùng bignum để tính toán).

2 Thuật toán random trong bài toán kiểm tra số chính phương

2.1 Lý thuyết về thuật toán

Ký hiệu: a mod b là phần dư của phép chia a/b.

Nhận xét 2.1

Nếu n là số chính phương thì chữ số tại hàng đơn vị của n thuộc tập các số $S = \{0, 1, 4, 5, 6, 9\}$,

Nói cách khác, nếu n là số chính phương thì n mod $10 \in S$. Hay, $f(x) = 1 \Rightarrow n \bmod 10 \in S$. Suy ra n mod $10 \in S$ là điều kiện cần để n là một số chính phương.
Vậy ta có thể làm một bài test (tạm gọi là bài **test1**) đơn giản, với một số n kiểm tra xem chữ số tận cùng của n có thuộc S hay không, nếu không thì chắc chắn n không phải là một số chính phương, ngược lại thì n **có khả năng là số chính phương**. Tạm gọi đây là một bài kiểm tra số chính phương.

Để đơn giản, mình đưa ra định nghĩa một số giả chính phương sau đây:

Định nghĩa 2.2

Số giả chính phương là một số tự nhiên, tuy không phải là số chính phương, nhưng nó lại vượt qua được bài kiểm tra số chính phương của chúng ta.

Với một số tự nhiên n bất kì thì ta có 60% khả năng là n mod $10 \in S$. Có thể nói, trong một đoạn $[l; r] \subset \mathbb{N}$ đủ lớn, ngoại trừ các số chính phương thì có (xấp xỉ) 60% các số tự nhiên là số giả chính phương nếu ta chỉ dùng bài kiểm tra **test1**. Vậy nếu ta tăng số lượng bài test lên, không chỉ kiểm tra 1 chữ số tận cùng mà có thể là 2, 3, ... chữ số tận cùng, hoặc là một số cách test khác (sẽ được trình bày dưới đây) thì có giảm xác suất một số là số giả chính phương hay không? Phần sau đây mình sẽ chứng minh điều đó rằng xác suất sẽ giảm theo cơ số 2.

Mình sẽ đưa ra một số định nghĩa để cho thuận tiện trong việc giải thích và chứng minh thuật toán.

Định nghĩa 2.3

Đặt $S(p)$ là tập hợp các số dư của mọi số chính phương khi chia cho p, một cách toán học thì có thể viết dưới dạng $S(p) = \{i^2 \mod p | i \in \mathbb{N}\}$.

Nhận xét 2.4

$$|S(p)| \leq \left\lfloor \frac{p}{2} \right\rfloor + 1$$

Chứng minh **Nhận xét 2.4**:
Xét modulo p và số x bất kì, ta luôn có:

$$x^2 \equiv (p-x)^2 \pmod{p}$$

Vì:

$$(p-x)^2 \equiv (p^2 + x^2 - 2px) \equiv x^2 \pmod{p}$$

Suy ra:

$$0^2 \equiv p^2 \pmod{p}$$
$$1^2 \equiv (p-1)^2 \pmod{p}$$
$$2^2 \equiv (p-2)^2 \pmod{p}$$
$$\dots$$

Do đó, số phần tử khác nhau trong tập $S(p)$ sẽ không vượt quá $\left\lfloor \frac{p}{2} \right\rfloor + 1$, ta có điều phải chứng minh.

Định nghĩa 2.5

Đặt Test(n,p) là một hàm số thoả mãn:

$$\text{Text}\,(n,p) = \begin{cases} 1, & \text{nếu } n \mod p \in S(p) \\ 0, & \text{ngược lại} \end{cases}$$

Từ **nhận xét 2.4**, ta có thể thấy, khi p lớn, có khoảng 50% khả năng một số n sẽ có Test(n, p)=1. Vậy nếu chỉ dùng một hàm Test thì xác suất một số là số giả chính phương là xấp xỉ 50%, vậy sẽ thế nào nếu ta dùng hàm Test nhiều lần, mỗi lần sử dụng một modulo khác nhau? Hãy xem Pseudo code dưới đây để hiểu hơn ý mình nói:

```
Check(n): //trả về 1 nếu n là số chính phương, 0 ngược lại
    for i = 1 to 20:
        p = random();
        if Test(n,p)==0:
            return 0; //nếu n không thuộc S(p) thì chắc chắn n không là số chính
                phương
    return 1; //nếu sau 20 lần test mà n vẫn thoả, thì khả năng cao n là số chính
        phương.
```

Trong code trên, để kiểm tra n có là một số chính phương hay không, mình đã làm 20 lần test trên 20 modulo random, nếu có bất cứ một lần nào đó mà Test(n,p) ra 0 thì chắc chắn n không phải số chính phương và mình sẽ ngừng ngay việc kiểm tra. Còn nếu sau 20 lần thử mà vẫn đúng thì khả năng rất cao n chính là số chính phương. Hàm của ta chỉ sai khi số n là một số giả chính phương, vì thế sau đây mình sẽ trình bày cách tính xác suất n là một số giả chính phương.

Gọi p_i là số tự nhiên được random ra tại lần test thứ i, $P(p_i, n)$ là xác suất để một số tự nhiên n là số giả chính phương xét trên modulo p_i.
Ta có:

$$P(p_i, n) = \frac{|S(p_i)|}{p_i} \leq \frac{\left\lfloor \frac{p_i}{2} \right\rfloor + 1}{p_i} \leq \frac{1}{2} + \frac{1}{p_i}$$

Khi p_i lớn, $\frac{1}{p_i}$ sẽ nhỏ, nên có thể xem như nó bằng 0. Vậy xác suất một số tự nhiên là số giả chính phương có thể xem là xấp xỉ 50%.

Nếu ta test 1 lần, thì xác suất một số là giả chính phương là 50%, test 2 lần với 2 modulo khác nhau thì có thể xem xác suất một số là giả chính phương là 25%, ... Tuy nhiên, điều này chỉ đúng khi các bài test độc lập với nhau, nếu không, khi một số là giả chính phương đối với modulo X thì xác suất nó là giả chính phương đối với modulo Y có thể thay đổi (bạn nào đã biết xác suất có điều kiện rồi sẽ hiểu ý mình). Về mặt lý thuyết, có thể chọn các modulo p_i đôi một nguyên tố cùng nhau thì các bài kiểm tra sẽ độc lập nhau, nhưng trong khuôn khổ tài liệu này, mình tạm thời bỏ đi một ít sự chính xác để cho đơn giản hoá ý tưởng.
Tóm lại, sau mỗi lần test, xác suất một số là số giả nguyên tố sẽ giảm đi 50%, vậy sau k lần test (với bài làm của mình thì $k = 20$) thì xác suất một số là số giả chính phương là: $\frac{1}{2^k} = \frac{1}{2^{20}} \approx 9 * 10^{-7}$. Đây có thể xem như là xác suất sai của thuật toán của chúng ta, bởi vì thuật toán chúng ta chỉ sai khi n là số giả chính phương, với xác suất nhỏ như vậy, có thể đảm bảo rằng thuật toán sẽ ra kết quả đúng trên tất cả các test.

2.2 Cài đặt thuật toán

Dưới đây là cách cài đặt thuật toán trên bằng C++, độ phức tạp của cách cài đặt này là $O(k * len(n) + Sum(p))$, với k là số lần test, $len(n)$ là độ dài của số n, $Sum(p)$ là tổng độ lớn của tất cả modulo trong các lần test.

```
#include <bits/stdc++.h>
using namespace std;
// rd dùng để random, thay cho hàm rand() của C++.
// Có rất nhiều lý do để không dùng hàm rand() mà mình không tiện nói ở đây.
mt19937 rd(chrono::steady_clock::now().time_since_epoch().count());
int getRemainder(const string &n, int p) //trả về số dư của phép chia n/p
{
    int ans = 0;
    for (char digit : n)
        ans = (ans * 10 + digit - '0') % p;
    return ans;
}
bool insideSetS(int remainder, int p) //trả về 1 nếu remainder thuộc tập hợp S(p)
{
    for (int i = 0; i <= p / 2 + 1; ++i)
        if (i * i % p == remainder)
```

```
            return 1;
    return 0;
}
bool Test(const string &n, int p) {
    int remainder = getRemainder(n, p);
    return insideSetS(remainder, p);
}
string F(const string &n) //trả về n là số chính phương hay không, do n lớn nên lưu
    dưới dạng xâu.
{
    //kiểm tra 20 lần
    for (int i = 1; i <= 20; ++i) {
        int p = uniform_int_distribution<int> (50000, 100000)(rd); //random p là một
            số nằm trong đoạn [50000; 100000]
        if (Test(n, p) == false)
            return n + " khong phai la so chinh phuong";
    }
    return n + " la so chinh phuong";
}
int main() {
    cout << F("4") << endl;
    cout << F("10") << endl;
    cout << F("15241578780673678515622620750190521") << endl;
        //123456789123456789*123456789123456789
    cout << F("15241578780673678515622620750190522") << endl;
        //123456789123456789*123456789123456789+1
    return 0;
}
```

3 Một số nhận xét và tối ưu thuật toán

Lưu ý: Phần này được thêm vào sau khi mình đã đăng bản gốc lên mà nhận được một số góp ý, cám ơn bạn Trịnh Hữu Gia Phúc, Đặng Đoàn Đức Trung đã có một vài nhận xét để giúp mình tối ưu thuật toán.

3.1 Tối ưu cho việc kiểm tra nhiều số

Ở cách cài đặt trên, mỗi khi mình kiểm tra một số n mới thì mình cần random ra nhiều số p_i, rồi lại cần sinh ra toàn bộ tập hợp $S(p_i)$, điều này rất phí. Tưởng tượng mình cần kiểm tra một lúc khoảng 10^4 số xem nó có là chính phương hay không, thì mình lại dùng 10^4 bộ các số p_i khác nhau, điều này là không cần thiết.

Một cách tối ưu đơn giản là mình có thể dùng một bộ các số p_i duy nhất để kiểm tra cho toàn bộ các số. Bằng cách này, mình có thể sinh ra tập hợp $S(p_i)$ trước, sau đó hàm có thể được thực hiện trong độ phức tạp $O(logp_i)$ với cấu trúc dữ liệu set hoặc $O(1)$ với mảng đánh dấu (chỉ dùng được mảng đánh dấu nếu p_i nhỏ).

Độ phức tạp của cách làm này là $O(Sum(p))$ cho việc khởi tạo, và $O(k * len(n) * X)$ cho mỗi lần kiểm tra, với X là độ phức tạp của hàm **insideSetS**.

3.2 Tối ưu việc chọn modulo

Từ định lý Thặng dư Trung Hoa, ta có thể suy ra trực tiếp nhận xét sau:
Nhận xét 3.1. *Nếu cách cài đặt ở* **phần 2.2** *random ra các modulo* $p_1, p_2, p_3, ..., p_k$ *dùng để kiểm tra, thì độ chính xác tương đương với việc kiểm tra bằng modulo* q *với* $q = LCM_{i=1}^{k} p_i$*, nói cách khác,* q *là bội chung nhỏ nhất của tất cả các modulo* p_i*.*

Do đó, ở đây ta có thể nghĩ tới một cách để tăng độ chính xác đó là chọn các p_i đôi một nguyên tố cùng nhau, hoặc đơn giản hơn là chọn luôn các p_i là số nguyên tố.

3.3 Tối ưu hàm insideSetS(r, p)

Hàm này thực chất là kiểm tra xem phương trình: $x^2 = r \pmod p$ có nghiệm hay không. Việc này dẫn tới việc kiểm tra xem có tồn tại một căn bậc 2 của r trong modulo p hay không. Đây là một bài toán rất nổi tiếng **Quadratic residue**.

Dựa vào **Euler's criterion**, với p là một số nguyên tố lẻ, thì phương trình trên có nghiệm khi và chỉ khi $r^{\frac{p-1}{2}} = 1 \pmod p$. Việc kiểm tra điều kiện trên có thể thực hiện trong $O(logp)$ bằng phép lũy thừa nhanh và không tốn chi phí khởi tạo.

Vậy nếu ta có một cách random ra các modulo p_i là số nguyên tố lẻ, thì ta có thể thực hiện hàm Test(n, p) trong độ phức tạp $O(len(n)+log(p))$ mà không tốn chi phí khởi tạo như thuật toán đã nêu ở **phần 2.2** hay **phần 3.1**. Điều này giúp ta có thể chọn các số p_i lớn hơn nhiều nếu như ta có danh sách các số nguyên tố lớn. Chẳng hạn có thể chọn số $10^9 + 7$ thì vẫn có thể kiểm tra được.

Dưới đây là cách cài đặt thuật toán trên bằng C++, độ phức tạp của cách cài đặt này là $O(k * len(n) + Sum(log(p_i))$, với k là số lần test, $len(n)$ là độ dài của số n, $Sum(log(p_i))$ là tổng $log(p_i)$. Để đơn giản thì mình chọn 20 số nguyên tố cố định, nhưng khi làm bài thì bạn nên chọn random các số nguyên tố từ một danh sách nào đó cho trước, để tránh bị counter.

```
#include <bits/stdc++.h>
using namespace std;
// Danh sách 20 số nguyên tố >= 1e9
const int nPrime = 20;
int primeList[] = {1000000007, 1000000009, 1000000021, 1000000033, 1000000087,
    1000000093, 1000000097, 1000000103, 1000000123, 1000000181, 1000000207,
    1000000223, 1000000241, 1000000271, 1000000289, 1000000297, 1000000321,
    1000000349, 1000000363, 1000000403};
int getRemainder(const string &n, int p) //trả về số dư của phép chia n/p
{
    int ans = 0;
    for (char digit : n)
        ans = (10ll * ans + digit - '0') % p;
    return ans;
}
// Tính a^b trong modulo MOD bằng thuật chia nhị phân
int fastPow(int a, int b, int MOD) {
    int res = 1;
    for (; b; b >>= 1, a = 1ll * a * a % MOD)
        if (b & 1)
            res = 1ll * res * a % MOD;
    assert(res == 1 || res == MOD - 1);
    return res;
}
```

```
bool insideSetS(int remainder, int p) //trả về 1 nếu remainder thuộc tập hợp S(p)
{
    return fastPow(remainder, (p - 1) / 2, p) == 1;
}
bool Test(const string &n, int p) {
    int remainder = getRemainder(n, p);
    return insideSetS(remainder, p);
}
string F(const string &n) //trả về n là số chính phương hay không, do n lớn nên lưu
    dưới dạng xâu.
{
    //kiểm tra 20 lần
    for (int i = 0; i < nPrime; ++i) {
        if (Test(n, primeList[i]) == false)
            return n + " khong phai la so chinh phuong";
    }
    return n + " la so chinh phuong";
}
int main() {
    cout << F("4") << endl;
    cout << F("9") << endl;
    cout << F("15241578780673678515622620750190521") << endl;
        //123456789123456789*123456789123456789
    cout << F("15241578780673678515622620750190522") << endl;
        //123456789123456789*123456789123456789+1
    return 0;
}
```

4 Kết luận đánh giá

Có thể thấy, việc kiểm tra số chính phương khi giới hạn nhỏ là một công việc rất chi là dễ dàng, nhưng khi tăng giới hạn lên thì nó lại trờ nên cực kì khó khăn nếu như không có công cụ mạnh mẽ (ở đây là bignum).

Với cách dùng random này, chúng ta đánh đổi đi tính chính xác (không còn là 100% chính xác như cài bignum) để giảm độ phức tạp thuật toán cũng như sự phức tạp trong cài đặt. Dù là thế, độ chính xác cũng đã tiệm cận 100% nên không có vấn đề gì lắm. Với các contest dạng OI thì không phải lo, còn nếu dạng ACM thì sẽ có một chút lo ngại nếu như phải dùng hàm này nhiều lần, có thể rơi vào 1 test sai và dẫn tới sai toàn bộ bài (nhưng trường hợp này rất hiếm, mình chưa gặp bao giờ).