

# Kiểm tra số nguyên tố

## Kiểm tra số nguyên tố

**Người viết:** Nguyễn Anh Bảo - Đại học Bách Khoa Hà Nội

**Reviewer:**

- Ngô Nhật Quang - The University of Texas at Dallas
- Phạm Hoàng Hiệp - University of Georgia

## Giới thiệu

Trong bài viết này, chúng ta sẽ cùng tìm hiểu một số thuật toán và phương pháp kiểm tra một số tự nhiên bất kì có là số nguyên tố hay không.

## Số nguyên tố

Một số tự nhiên  $n$  được gọi là số nguyên tố khi và chỉ khi  $n$  có đúng 2 ước dương là 1 và chính nó.

Ví dụ: 2, 3, 5, 101 là các số nguyên tố. 0, 1, 10, 12 không là số nguyên tố.

Trong bài viết này, chúng ta sẽ tập trung vào việc kiểm tra **một** số nguyên dương có phải số nguyên tố hay không. Để kiểm tra tính nguyên tố của nhiều số nguyên trên một đoạn  $[a, b]$ , bạn đọc có thể tham khảo bài viết [Sàng nguyên tố](#).

## 1. Thuật toán ngây thơ

### 1.1. Ngây thơ 1

Cách đơn giản nhất để kiểm tra tính nguyên tố của số tự nhiên  $n$  là trực tiếp sử dụng định nghĩa số nguyên tố:



Số tự nhiên  $n \geq 2$  là số nguyên tố khi và chỉ khi  $n$  không chia hết cho các số tự nhiên  $2, 3, \dots, n - 1$ .

Ta có thể cài đặt như sau:

```
1 | bool primeCheck(int n)
2 | {
3 |     if (n < 2)
4 |         return false;
5 |     for (int i = 2; i < n; ++i)
6 |         if (n % i == 0)
```

```

7 |         return false;
8 |     return true;
9 | }

```

Độ phức tạp thuật toán:  $\mathcal{O}(n)$ .

## 1.2. Ngây thơ 2

Để tối ưu thuật toán trên, nhận xét rằng nếu  $n$  có một ước  $d$  sao cho  $n - 1 \geq d \geq \sqrt{n}$  thì  $\frac{n}{d}$  cũng là ước của  $n$  và có  $1 < \frac{n}{d} \leq \sqrt{n}$ . Suy ra nếu  $n$  không chia hết cho các số tự nhiên lớn hơn 1 và không vượt quá  $\sqrt{n}$  thì  $n$  cũng không chia hết cho các số tự nhiên lớn hơn  $\sqrt{n}$ . Từ đó, thay vì xét tính chia hết của  $n$  cho  $i = 2, 3, \dots, n - 1$  ta chỉ cần xét  $i = 2, 3, \dots, \lfloor \sqrt{n} \rfloor$ .

Cài đặt thuật toán:

```

1 | bool primeCheck(int n)
2 | {
3 |     if (n < 2)
4 |         return false;
5 |     for (int i = 2; i * i <= n; ++i)
6 |         if (n % i == 0)
7 |             return false;
8 |     return true;
9 | }

```

Độ phức tạp thuật toán:  $\mathcal{O}(\sqrt{n})$ .

Ta có thể mở rộng thuật toán trên thành thuật toán phân tích một số nguyên dương ra thừa số nguyên tố:

```

1 | void primeFactorization(int n)
2 | {
3 |     for (int i = 2; i * i <= n; ++i)
4 |         while (n % i == 0)
5 |             {
6 |                 n /= i;
7 |                 cout << i << ' ';
8 |             }
9 |     if (n > 1)
10 |         cout << n;
11 | }

```

## 1.3. Ngây thơ 2.5

Để ý nếu số nguyên tố  $n$  lẻ thì  $n$  không chia hết cho một số chẵn bất kì. Do đó nếu  $n > 2$ , ta chỉ cần xét các số  $i$  lẻ thuộc đoạn  $[2, \lfloor \sqrt{n} \rfloor]$ . Tương tự, nếu  $n > 3$  thì ta chỉ cần xét  $i$  là các số không chia hết cho 3. Từ hai nhận xét trên, nếu  $n > 3$  thì ta chỉ cần xét các số  $i$  sao cho  $i$  chia 6 dư 1 hoặc 5.

```

1 | bool primeCheck(int n)
2 | {
3 |     if (n == 2 || n == 3)
4 |         return true;
5 |     if (n < 3 || n % 2 == 0 || n % 3 == 0)
6 |         return false;

```

```

7 |     for (int i = 5; i * i <= n; i += 6)
8 |         if (n % i == 0 || n % (i + 2) == 0)
9 |             return false;
10 |     return true;
11 | }

```

**Chú ý:** Có thể sử dụng nhiều số nguyên tố đầu tiên để tối ưu thuật toán hơn. Về lý thuyết, nếu  $k$  là số số nguyên tố được dùng càng lớn thì vòng lặp chạy càng nhanh. Tuy nhiên, với  $k = 50$ , độ phức tạp vòng lặp `for` là  $\mathcal{O}\left(\frac{\sqrt{n}}{10}\right)$ . Và kể cả với  $k = 6 \cdot 10^5$  thì độ phức tạp thuật của vòng lặp `for` vẫn là  $\mathcal{O}\left(\frac{\sqrt{n}}{30}\right)$ .

Do đó thuật toán này có thể không đủ nhanh để giải quyết giới hạn  $n \leq 10^{18}$ , hoặc  $n \leq 10^9$  nhưng phải kiểm tra  $10^6$  số  $n$  trở lên. Để giải quyết các bài toán có giới hạn lớn như thế, ta phải sử dụng đến các phương pháp xác suất.

## 2. Phép thử Fermat (Định lí Fermat nhỏ)

### 2.1. Ý tưởng

Theo định lí Fermat nhỏ, nếu  $p$  là một số nguyên tố thì với mọi số nguyên  $a$  thỏa mãn  $\gcd(a, p) = 1$ , ta có:

$$a^{p-1} \equiv 1 \pmod{p}$$

Từ định lý Fermat ta có ý tưởng kiểm tra tính nguyên tố của số nguyên dương  $n$  như sau:

- ▶ Xét số nguyên  $a \in [2, n-1]$ , nếu  $a^{n-1} \not\equiv 1 \pmod{n}$  thì ta **chắc chắn**  $n$  là hợp số hoặc  $n < 2$ .
- ▶ Ngược lại, nếu  $a^{n-1} \equiv 1 \pmod{n}$  thì  $n$  **có thể** là số nguyên tố.

**Chú ý:** Phần in đậm của phép thử nghĩa là tồn tại các giá trị của  $n$  và  $a$  sao cho  $n$  là hợp số và  $a^{n-1} \equiv 1 \pmod{n}$ . Ví dụ, nếu  $n = 15$  và  $a = 4$  thì  $4^{14} \equiv 1 \pmod{15}$ . Trong trường hợp này,  $n$  được gọi là số giả nguyên tố cơ sở  $a$ , hoặc số nguyên tố xác suất cơ sở  $a$ .

Về lý thuyết, nếu ta kiểm tra đẳng thức Fermat với mọi số  $a$ , ta có thể kết luận chắc chắn tính nguyên tố của  $n$ . Tuy nhiên, việc kiểm tra đẳng thức với mọi  $a$  sẽ phức tạp hơn cả thuật toán ngây thơ. Do đó, phép thử Fermat sẽ thực hiện một số lần thử với các số  $a$  được lấy ngẫu nhiên. Trong các bài toán lập trình thi đấu, phép thử vẫn có độ chính xác đủ tốt.

### 2.2. Cài đặt

Ta có thể cài đặt kết quả của phép tính  $a^{n-1} \pmod{n}$  bằng lũy thừa nhị phân.

```

1 | // Tính a^k (mod n)
2 | int binaryPower(long long a, int k, int n)
3 | {
4 |     a = a % n;
5 |     long long res = 1;
6 |     while (k)
7 |     {
8 |         if (k & 1)
9 |             res = (res * a) % n;
10 |        a = (a * a) % n;
11 |        k /= 2;
12 |    }
13 | }

```

```

13 |     return res;
14 | }

```

Cài đặt phép thử Fermat:

```

1 | bool isProbablyPrime(int n)
2 | {
3 |     if (n < 7)
4 |         return n == 2 || n == 3 || n == 5;
5 |
6 |     static const int repeatNum = 5;
7 |     for (int i = 0; i < repeatNum; ++i)
8 |     {
9 |         int a = rand() % (n - 3) + 2;
10 |         if (binaryPower(a, n - 1, n) != 1)
11 |             return false;
12 |     }
13 |     return true;
14 | }

```

Độ phức tạp phép thử là  $\mathcal{O}(c \log n)$  với  $c$  là số cơ số  $a$  được thử.

Do thuật toán `binaryPower` có sử dụng phép tính  $(a * a) \% n$  nên nếu  $n \geq 2^{32}$  sẽ có thể bị tràn số. Phiên bản được trình bày ở trên xử lý  $n < 2^{32}$ . Để áp dụng thuật toán lũy thừa nhị phân với  $10^{18} \geq n$ , bạn phải dùng kiểu số nguyên 128 — *bit* nếu ngôn ngữ lập trình cho phép. Nếu ngôn ngữ lập trình chỉ có loại số nguyên 64 — *bit*, bạn có thể tham khảo thuật toán được chỉnh sửa như sau:

```

1 | // Tính a * b mod n
2 | long long binaryMul(long long a, long long b, long long n)
3 | {
4 |     a = a % n;
5 |     long long res = 0;
6 |     while (b)
7 |     {
8 |         if (b & 1)
9 |             res = (res + a) % n;
10 |         a = (2 * a) % n;
11 |         b /= 2;
12 |     }
13 |     return res;
14 | }
15 |
16 | // Tính a^b mod n
17 | long long binaryPow(long long a, long long k, long long n)
18 | {
19 |     a = a % n;
20 |     long long res = 1;
21 |     while (k)
22 |     {
23 |         if (k & 1)
24 |             res = binaryMul(res, a, n);
25 |         a = binaryMul(a, a, n) % n;
26 |         k /= 2;
27 |     }
28 | }

```

```

28 |
29 | } return res;

```

Khi đó độ phức tạp thuật toán là  $\mathcal{O}(c \log^2 n)$ .

**Chú ý:** Do không có tính chính xác tuyệt đối nên phép thử Fermat không phải là một thuật toán.

Tuy tốc độ cao và dễ cài đặt, vẫn có những trường hợp xác suất phép thử Fermat thất bại là rất cao. Ví dụ xét số  $n = 561 = 3 \times 11 \times 17$ . Số này có tính chất với mọi số nguyên  $a$  mà  $\gcd(a, n) = 1$  thì  $a^{n-1} \equiv 1 \pmod n$ . Do đó, trừ khi trong các lần thử ngẫu nhiên ta chọn được  $a$  chia hết cho 3, 11 hoặc 17 thì phép thử sẽ cho kết quả sai.

Các số có tính chất trên được gọi là số *Carmichael*.

## 2.3. Vấn đề với số Carmichael

Xét  $n$  là một số Carmichael có  $k$  ước nguyên tố,  $c$  là số lần chọn cơ số  $a$  (`repeatNum`). Nếu  $k = 3$  thì số  $n$  nhỏ nhất là 561. Xác suất phép thử kết luận đúng với  $k = 3$  do đó bị chặn bởi  $T(c)$ .  $T(c)$  cho bởi bảng sau:


$c$	1	2	3	4	5	...
$T(c)$	49%	74%	86%	93%	97%	...

Từ bảng trên có thể thấy để có độ chính xác  $\geq 90\%$  thì ta phải thử ít nhất 4 lần, và có thể nhiều hơn. Trong khi nếu ta chỉ thử 1 lần duy nhất thì xác suất phép thử kết luận sai là  $> \frac{1}{2}$ .

Với  $k = 4$ , số Carmichael nhỏ nhất có 4 ước nguyên tố là  $41041 = 7 \times 11 \times 13 \times 41$ . Tương tự, xét xác suất phép thử cho kết quả đúng với  $n$  là số Carmichael dạng này bị chặn bởi  $T(c)$ .  $T(c)$  được cho bởi bảng sau:

$c$	1	2	3	4	5	...
$T(c)$	34%	56%	71%	81%	88%	...

Tức là kể cả ta có thử đến 5 lần thì xác suất phép thử kết luận đúng vẫn không thể quá 90%.

Một điều khá thú vị là các số Carmichael phân bố rất ít trong tập các số tự nhiên. Theo [OEIS-A055553](https://oeis.org/A055553) 

- Số các số Carmichael nhỏ hơn  $10^6$  là 43
- Số các số Carmichael nhỏ hơn  $10^9$  là 646
- Số các số Carmichael nhỏ hơn  $10^{18}$  là 1401644

Do đó, các bạn có thể yên tâm khi sử dụng phép thử Fermat nếu test được sinh ngẫu nhiên, vì xác suất gặp số Carmichael rất thấp. Nếu test cố tình chọn số Carmichael thì phép thử không còn đáng tin cậy. Rất may là có những phép thử hiệu quả và chính xác hơn phép thử Fermat. Trong phần tiếp theo chúng ta sẽ cùng tìm hiểu thuật toán Rabin-Miller.

## 3. Thuật toán Rabin-Miller

### 3.1. Ý tưởng

Thuật toán Rabin-Miller là phiên bản mở rộng và mạnh hơn của phép thử Fermat. Thuật toán dựa vào nhận xét sau:



Với mọi số nguyên dương  $x$ , ta tìm được duy nhất hai số tự nhiên  $k, m$  sao cho  $x = 2^k \times m$  và  $m$  lẻ.

Ví dụ:  $6 = 2^1 \times 3, 100 = 2^2 \times 25, 9 = 2^0 \times 9, \dots$

Do đó, xét số  $n$ , ta có thể phân tích  $n - 1$  thành  $2^k \times m$ , với  $m$  là số lẻ.

Theo định lý nhỏ Fermat, nếu  $n$  là số nguyên tố thì với mọi  $a$  sao cho  $\gcd(a, n) = 1$  ta có:

$$a^{n-1} \equiv 1 \pmod n \Leftrightarrow a^{2^k \cdot m} - 1 \equiv 0 \pmod n \Leftrightarrow (a^{2^{k-1}m} + 1)(a^{2^{k-2}m} + 1) \dots (a^m + 1)(a^m - 1) \equiv 0 \pmod n$$

Vì  $n$  là số nguyên tố nên tồn tại ít nhất một trong các nhân tử của vế trái chia hết cho  $n$ . Do đó, thay vì kiểm tra kết luận của định lý Fermat nhỏ, ta sẽ kiểm tra điều kiện sau:

- $a^m \equiv 1 \pmod n$  hoặc
- Tồn tại  $0 \leq l \leq k - 1$  sao cho  $a^{2^l m} \equiv -1 \pmod n$

Nếu cả hai điều kiện không được thỏa mãn thì chắc chắn  $n$  là hợp số.

Nhưng nếu cả hai điều kiện được thỏa mãn thì  $n$  có phải số nguyên tố không?

Câu trả lời là **không**. Ví dụ: với  $n = 28, a = 19$  thì  $n - 1 = 2^0 \times 27$  và  $19^{27} \equiv -1 \pmod{28}$ .

Do đó, để áp dụng ý tưởng trên, ta có thể triển khai theo hai cách sau:

### 3.2. Phép thử xác suất (Probabilistic)

Để tăng tính chính xác của thuật toán ta có thể lặp lại bước kiểm tra với nhiều cơ số  $a$ , giống như phép thử Fermat. Hơn thế nữa, chứng minh được nếu  $n$  là hợp số, chỉ có  $\approx 25\%$  số cơ số  $a$  trong đoạn  $[2, n - 1]$  thỏa mãn một trong hai điều kiện.

Nghĩa là với hợp số  $n$  bất kì, xác suất để thuật toán chứng minh được  $n$  là hợp số sau lần kiểm tra đầu tiên là  $\geq 75\%$ , lần thứ hai là  $\geq 93.75\%$ , lần thứ ba là  $\geq 98.43\%$ , lần thứ  $x$  là  $(1 - \frac{1}{4^x}) \times 100\%$ . Có thể thấy độ chính xác của thuật toán Rabin-Miller cao hơn nhiều so với phép thử Fermat, và tất nhiên là đủ tốt cho các bài toán lập trình thi đấu.

**Cài đặt thuật toán:**

```

1 // Tính a^k mod n
2 long long binaryPower(long long a, long long k, long long n)
3 {
4     a = a % n;
5     long long res = 1;
6     while (k)
7     {
8         if (k & 1)
9             res = (res * a) % n;
10        a = (a * a) % n;
11        k /= 2;
12    }
13    return res;
14 }
```

```

15
16 // Kiểm tra điều kiện thuật toán với a cố định
17 bool test(long long a, long long n, long long k, long long m)
18 {
19     long long mod = binaryPower(a, m, n);
20     if (mod == 1 || mod == n - 1)
21         return true;
22     for (int l = 1; l < k; ++l)
23     {
24         mod = (mod * mod) % n;
25         if (mod == n - 1)
26             return true;
27     }
28     return false;
29 }
30
31 bool RabinMiller(long long n)
32 {
33     // Kiểm tra với các giá trị nhỏ
34     if (n == 2 || n == 3 || n == 5 || n == 7)
35         return true;
36     if (n < 11)
37         return false;
38
39     // Tính m và k
40     long long k = 0, m = n - 1;
41     while (m % 2 == 0)
42     {
43         m /= 2;
44         k++;
45     }
46
47     // Lặp lại bước kiểm tra với a ngẫu nhiên
48     const static int repeatTime = 3;
49     for (int i = 0; i < repeatTime; ++i)
50     {
51         long long a = rand() % (n - 3) + 2;
52         if (!test(a, n, k, m))
53             return false;
54     }
55     return true;
56 }

```

Độ phức tạp là  $\mathcal{O}(c \log n)$ .

### 3.3. Thuật toán đơn định (Deterministic)

Phép thử xác suất có thể trở thành thuật toán bằng cách thay vì xét  $a$  ngẫu nhiên, ta sẽ xét tất cả  $a$  bị chặn bởi một hàm theo  $n$ . Miller chứng minh được nếu [Định đề Riemann tổng quát \(GRH\)](#)  $\square$  là đúng thì ta chỉ cần kiểm tra  $a \in [2, \mathcal{O}(\ln^2 n)]$ . Sau đó, Bach chứng minh được chỉ cần xét  $a \in [2, 2 \ln^2 n]$ .

Với  $n$  đủ lớn thì vẫn có rất nhiều giá trị cần kiểm tra. Nhưng người ta cũng chứng minh được rằng:

- ▶ Nếu  $n \leq 3 \cdot 10^9$ , chỉ cần xét  $a \in \{2; 3; 5; 7\}$
- ▶ Nếu  $n \leq 2^{64}$ , chỉ cần xét  $a \in \{2; 3; 5; 7; 11; 13; 17; 19; 23; 29; 31; 37\}$





Do đó, ta có phiên bản thuật toán (độ chính xác 100%) của phép thử như sau:

```
1  bool MillerRabin(long long n)
2  {
3      if (n == 2 || n == 3 || n == 5 || n == 7)
4          return true;
5      if (n < 11)
6          return false;
7
8      long long k = 0, m = n - 1;
9      while (m % 2 == 0)
10     {
11         m /= 2;
12         k++;
13     }
14
15     static vector<int> checkSet = {2, 3, 5, 7, 11, 13, 17, 19, 23, 29, 31, 37};
16     for (auto a : checkSet)
17         if (!test(a, n, k, m))
18             return false;
19     return true;
20 }
```

## 4. Bài tập luyện tập:

- ▶ [SPOJ - Prime Or Not](#) 

## 5. Tham khảo:

- ▶ [CP Algorithm - Primality Test](#) 
- ▶ [Wikipedia - Primality Miller-Rabin test](#) 
- ▶ [Wikipedia - Primality test](#) 
- ▶ [Wikipedia - Carmichael numbers](#) 

Được cung cấp bởi [Wiki.js](#)