

Số các ước và Tổng các ước

Tác giả:



- ▶ Nguyễn Đức Kiên, Trường Đại học Công nghệ, ĐHQGHN

Reviewer:

- ▶ Cao Thanh Hậu - Trường Đại học Khoa học Tự nhiên, ĐHQG-HCM
- ▶ Phạm Hoàng Hiệp - University of Georgia
- ▶ Nguyễn Minh Nhật - THPT chuyên Khoa học Tự nhiên, ĐHQGHN

Kiến thức cần biết

Để có thể hiểu được toàn bộ bài này, bạn nên nắm vững kiến thức và cài đặt được:

- ▶ [Sàng nguyên tố](#) 
- ▶ [Luỹ thừa nhanh](#) 

Một số ký hiệu được sử dụng

- ▶ $b|a$: b là ước của a . Ký hiệu này tương đương với $a:b$.
- ▶ $\log(N)$: Logarit cơ số 2 của N (quy ước chỉ trong bài viết này).

Bài toán đếm số ước

Bài toán 1: Đếm số ước của một số

Đề bài: Cho một số nguyên dương N . Đếm số ước của N .

Với nhiều coder, đây là một trong những bài toán beginner không thể không làm qua. Đây là bài toán thường được sử dụng như một ví dụ về việc phải tìm cách để tối ưu số lần lặp.

Cách 1: Duyệt để tìm ước

Giải thuật ngây thơ nhất rất đơn giản: từ nhận xét mọi ước của N đều không vượt quá N , ta duyệt mọi số nguyên dương trong đoạn $[1, N]$ và đếm số lượng ước của N trong đó:

```
1 | int divCount(int n) {  
2 |     int ret = 0;  
3 |
```

```

4 |     for (int i = 1; i <= n; i++)
5 |         ret += (n % i == 0);
6 |     return ret;
  | }

```

Cải tiến

Ta nhận thấy rằng chỉ cần duyệt các ước tới \sqrt{N} là đủ, do ứng với mọi ước d của N nhỏ hơn \sqrt{N} ta có hai ước của N là d và $\frac{N}{d}$, và khi N là số chính phương, ứng với ước \sqrt{N} ta có một ước duy nhất là chính nó.

Chứng minh

Gọi d là một ước của N , khi đó tồn tại một số nguyên g sao cho $d \times g = N$. Nếu $d = g$, khi đó ta có $d = g = \sqrt{N}$. Trường hợp $d \neq g$, không mất tính tổng quát ta giả sử $d < g$. Dễ thấy $d < \sqrt{N} < g$, vì nếu $\sqrt{N} \leq d$ thì ta có $d \times g > \sqrt{N} \times \sqrt{N} = N$ và nếu $\sqrt{N} \geq g$ thì $d \times g < \sqrt{N} \times \sqrt{N} = N$, đều mâu thuẫn với giả thiết $d \times g = N$.

```

1 | int divCount(long long n) {
2 |     int ret = 0;
3 |     for (int i = 1; 1ll * i * i <= n; i++) {
4 |         ret += 2 * (n % i == 0);
5 |         if (1ll * i * i == n) ret--;
6 |     }
7 |     return ret;
8 | }

```

Hai giải thuật trên có độ phức tạp thời gian lần lượt là $O(N)$ và $O(\sqrt{N})$, sẽ chạy tốt với $N < 10^6$ và $N < 10^{12}$. Rất khó để cải thiện độ phức tạp này để phù hợp với N lớn hơn.

Cách 2: Sử dụng sàng nguyên tố

Định lý: Nếu một số N được phân tích ra thừa số nguyên tố thành:

$$N = \prod_{i=1}^k p_i^{m_i} = p_1^{m_1} \times p_2^{m_2} \dots \times p_k^{m_k}$$

với p_i là các số nguyên tố, m_i là các số nguyên dương thì số ước của N là:

$$\tau(N) = \prod_{i=1}^k (m_i + 1)$$

(Chữ τ đọc là *tau*).

► Chứng minh (nhấn để hiện)

Thuật toán

Để giải bài toán này theo cách phân tích ra thừa số nguyên tố, đầu tiên ta sử dụng sàng nguyên tố để lưu lại các số nguyên tố nhỏ hơn N . Sau đó, lần lượt chia N cho các số trên rồi thu lại số lần chia ứng với mỗi số; đó chính là số mũ của các thừa số nguyên tố tương ứng. Cuối cùng, áp dụng công thức trên để tính ra kết quả.

Cách làm này mất $O(N \log \log N)$ thời gian và không gian để chuẩn bị sàng, và $O(\pi(N)) \approx O(\frac{N}{\ln N})$ thời gian để phân tích, với $\pi(n)$ là số lượng số nguyên tố nhỏ hơn N . Tổng độ phức tạp là $O(N \log \log N)$.

Cải tiến 1

Sử dụng nhận xét sau, ta giảm được độ phức tạp xuống $O(\sqrt{N} \log \log \sqrt{N})$. Ta viết $N = X \times Y$. Trong đó, số $X = \prod_{p_i < \sqrt{N}} p_i^{m_i}$ là tích của tất cả các thừa số nguyên tố của N đã nâng lên lũy thừa có cơ số nhỏ hơn \sqrt{N} , còn $Y = \frac{N}{X}$. Xét hai trường hợp:

- ▶ $Y = 1$. Trường hợp này hoàn toàn giống với thuật toán $O(N \log \log N)$ chưa được cải tiến.
- ▶ $Y \neq 1$. Lúc này, nếu phân tích Y ra thừa số nguyên tố, ta sẽ không thu được một thừa số nào nhỏ hơn \sqrt{N} , vì các thừa số như vậy đều đã nằm trong X . Điều đó có nghĩa là Y không có ước nguyên tố nào nhỏ hơn \sqrt{Y} , do đó Y là một số nguyên tố.

Ví dụ:

- ▶ $N = 60$, với cách đặt trên ta có $X = 60$, $Y = 1$
- ▶ $N = 60\,000\,180$, với cách đặt trên ta sẽ có $X = 60$ và $Y = 1\,000\,003$.

Khi cài đặt, sau khi chia N cho toàn bộ các số nguyên tố đã thu được sau khi sàng, giá trị N còn lại chính là Y .

Một edge case nhỏ nhỏ là $N = 1$. Do 1 không thể phân tích được thành thừa số nguyên tố, ta trả về luôn 1 cho trường hợp này.

```
1  const int MAXN = 1e6;
2
3  vector<int> primes;
4  vector<bool> isPrime;
5
6  void sieve() {
7      isPrime.assign(MAXN + 1, 1);
8      isPrime[0] = isPrime[1] = 0;
9      for (int i = 2; i <= MAXN; i++) {
10         if (!isPrime[i]) continue;
11         primes.push_back(i);
12         for (int j = i + i; j <= MAXN; j += i) {
13             isPrime[j] = 0;
14         }
```

```

15     }
16 }
17 }
18
19 int countDiv(long long n) {
20     if (n == 1) return 1;
21     vector<int> powV;
22     for (auto p : primes) {
23         int cnt = 0;
24         while (n % p == 0) {
25             n /= p;
26             ++cnt;
27         }
28         if (cnt) powV.push_back(cnt);
29     }
30     if (n != 1) powV.push_back(1);
31     int ret = 1;
32     for (auto i : powV) ret *= (i + 1);
33     return ret;
34 }
35
36 void solution() {
37     sieve();
38     long long n;
39     cin >> n;
40     cout << countDiv(n);
41 }

```

Cải tiến 2


Cho tới bước này, ta có thể giải được bài toán với $N \leq 10^{12}$. Ta sẽ cải tiến để bài toán giải được với $N \leq 10^{18}$, với độ phức tạp là khoảng $O(\sqrt[3]{N})$.

Vẫn viết N dưới dạng $N = X \times Y$ như trên, X được định nghĩa tương tự như trên, chỉ khác giới hạn các thừa số nguyên tố lúc này là $\sqrt[3]{N}$: $X = \prod_{p_i < \sqrt[3]{N}} p_i^{m_i}$. Lúc này ta không thể chắc chắn Y là số nguyên tố nữa. Tuy nhiên, Y lại chỉ có thể thuộc vào một trong bốn trường hợp:

- $Y = 1$. Hiển nhiên $\tau(N) = \tau(X)$
- Y là số nguyên tố. Khi đó nó chỉ có 2 ước là 1 và Y , và khi đó $\tau(N) = \tau(X) \times 2$.
- Y là bình phương của một số nguyên tố. Khi đó nó có 3 ước là 1, \sqrt{Y} và Y , và $\tau(N) = \tau(X) \times 3$.
- Y là tích của hai số nguyên tố. Khi đó nó có 4 ước và $\tau(N) = \tau(X) \times 4$.

Các trường hợp khác:

- ▶ Y không là tích của nhiều hơn hai số nguyên tố. Tất cả các số nguyên tố nhỏ hơn $\sqrt[3]{N}$ đã bị sàng nguyên tố loại bỏ và trở thành ước của X , vì vậy tích ba số nguyên tố bất kỳ còn lại sẽ lớn hơn N .
- ▶ Y không là tích của một hợp số với một số nguyên tố hoặc tích của hai hợp số. Ta biết (các) ước hợp số này không thể chứa các ước nguyên tố nhỏ hơn $\sqrt[3]{N}$ do chúng đã bị loại khỏi sàng, nghĩa là nếu trường hợp này xảy ra, Y sẽ là tích của ít nhất 3 số nguyên tố.

Để kiểm tra trường hợp thứ nhất, ta có thể dùng [thuật toán Rabin-Miller](#) . Để kiểm tra trường hợp thứ hai, ta kiểm tra xem Y có phải số chính phương không. Các số còn lại sẽ rơi vào trường hợp thứ ba.

```

1  const int MAXN = 1e6;
2
3  vector<int> primes;
4  vector<bool> isPrime;
5
6  void sieve() {
7      isPrime.assign(MAXN + 1, 1);
8      isPrime[0] = isPrime[1] = 0;
9      for (int i = 2; i <= MAXN; i++) {
10         if (!isPrime[i]) continue;
11         primes.push_back(i);
12         for (int j = i + i; j <= MAXN; j += i) {
13             isPrime[j] = 0;
14         }
15     }
16 }
17
18 long long binaryPower(long long a, long long k, long long n) {
19     a = a % n;
20     long long res = 1;
21     while (k) {
22         if (k & 1)
23             res = (res * a) % n;
24         a = (a * a) % n;
25         k /= 2;
26     }
27     return res;
28 }
29
30 bool test(long long a, long long n, long long k, long long m) {
31     long long mod = binaryPower(a, m, n);
32     if (mod == 1 || mod == n - 1)
33         return 1;
34     for (int l = 1; l < k; ++l) {
35         mod = (mod * mod) % n;

```

```

36         if (mod == n - 1)
37             return 1;
38     }
39     return 0;
40 }
41
42 bool isPrimeRabinMiller(long long n) {
43     vector<int> checkSet = {2, 3, 5, 7,
44                           11, 13, 17, 19,
45                           23, 29, 31, 37};
46     if (n <= 37) {
47         for (int i : checkSet) {
48             if (i == n) return 1;
49         }
50         return 0;
51     }
52
53     long long k = 0, m = n - 1;
54     while (m % 2 == 0) {
55         m /= 2;
56         k++;
57     }
58
59     for (auto a : checkSet)
60         if (!test(a, n, k, m))
61             return 0;
62     return 1;
63 }
64
65 bool isSquare(long long n){
66     long long c = sqrt(n + 4);
67     return c * c == n || (c - 1) * (c - 1) == n;
68 }
69
70 int countDiv(long long n) {
71     if (n == 1) return 1;
72     vector<int> powV;
73     for (auto p : primes) {
74         int cnt = 0;
75         while (n % p == 0) {
76             n /= p;
77             ++cnt;
78         }
79         if (cnt) powV.push_back(cnt);
80     }
81     if (n != 1) {
82         if (isPrimeRabinMiller(n)) powV.push_back(1);
83         else if (isSquare(n)) powV.push_back(2);

```

```

84         else {
85             powV.push_back(1);
86             powV.push_back(1);
87         }
88     }
89     int ret = 1;
90     for (auto i : powV) ret *= (i + 1);
91     return ret;
92 }
93
94 void solution() {
95     sieve();
96     long long n;
97     cin >> n;
98     cout << countDiv(n);
99 }

```

Phần Rabin-Miller có độ phức tạp là $O(12 \log N)$. Phần kiểm tra số chính phương có độ phức tạp là $O(1)$, với chú ý rằng cần phải xét một vài số lân cận $\sqrt[4]{N}$ để đưa ra kết quả chính xác. Tổng hợp lại, độ phức tạp của giải thuật trên là $O(\sqrt[3]{N} \log \log \sqrt[3]{N} + \log N)$.

Giải thuật này thậm chí **còn có thể tối ưu** tới khoảng $O(\sqrt[4]{N})$. Tuy vậy số trường hợp được đặt ra là tương đối lớn và kiểm tra chúng cũng không hề dễ dàng.

Ngoài ra, nếu có thể phân tích N ra thừa số nguyên tố bằng **thuật toán rho của Pollard**, lời giải lúc này cũng có độ phức tạp là khoảng $O(\sqrt[4]{N})$.

Bài toán 2: Đếm số ước của nhiều số

Đề bài: Cho Q truy vấn. Ở truy vấn thứ i , cần tìm số các ước của số A_i với $A_i < N$.

Bằng cách duyệt qua mọi ước của từng số một, độ phức tạp thời gian sẽ là $O(Q\sqrt{N})$. Còn với cách thứ hai giống như ở trên, độ phức tạp tốt nhất là $O(Q\sqrt[3]{N} \log \log \sqrt[3]{N} + Q \log N)$. Với Q nhỏ, ta hoàn toàn có thể làm tương tự như trên.

Khi Q đủ lớn ($Q \leq 10^6$) và N không quá lớn ($N \leq 10^6$), các hướng làm trên tỏ ra khá tồi.

Với cách thứ hai, ta có thể thay đổi một chút để có thể đưa ra kết quả của truy vấn nhanh hơn. Khi chuẩn bị, với mỗi số x , thay vì chỉ lưu `isPrime[x]`, ta lưu lại ước nguyên tố nhỏ nhất của x , là `minPDiv[x]`. Khi thực hiện truy vấn, thay vì chia N cho toàn bộ các số nguyên tố, ta chỉ cần chia liên tục N cho ước nguyên tố nhỏ nhất của nó tại thời điểm đó, ta sẽ dần thu được kết quả.

```

1  const int MAXN = 1e6;
2
3  vector<int> minPDiv;
4
5

```

```

5  void sieve() {
6      minPDiv.resize(MAXN + 1);
7      for (int i = 2; i <= MAXN; i++) {
8          if (minPDiv[i]) continue;
9          minPDiv[i] = i;
10         for (int j = i + i; j <= MAXN; j += i) {
11             minPDiv[j] = i;
12         }
13     }
14 }
15
16 int countDiv(int n) {
17     if (n == 1) return 1;
18     vector<int> powV;
19     int lastDiv = 0;
20     int cnt = 0;
21     while (n != 1) {
22         if (minPDiv[n] != lastDiv) {
23             if (cnt) powV.push_back(cnt);
24             cnt = 0;
25         }
26         ++cnt;
27         lastDiv = minPDiv[n];
28         n /= minPDiv[n];
29     }
30     if (cnt) powV.push_back(cnt);
31     int ret = 1;
32     for (auto i : powV) ret *= (i + 1);
33     return ret;
34 }
35
36 void solution() {
37     sieve();
38     int q;
39     cin >> q;
40     while (q--) {
41         int n;
42         cin >> n;
43         cout << countDiv(n) << "\n";
44     }
45 }

```

Phần chuẩn bị sàng nguyên tố mất $O(N)$ không gian và $O(N \log \log N)$ thời gian. Mỗi truy vấn, số phép tính được thực hiện bằng đúng tổng số mũ của các thừa số nguyên tố trong phân tích của N . Dễ thấy giá trị này không vượt quá $\log N$, và do đó độ phức tạp thời gian của phần này là $O(Q \log N)$. Độ phức tạp thời gian nói chung của cả giải thuật là $O(N \log \log N + Q \log N)$.

Ta cũng có thể lưu lại thêm cả số mũ của ước nguyên tố nhỏ nhất cùng với ước đó của từng số. Cách làm này sẽ có độ phức tạp là $O(N \log N + Q \log \log N)$ thời gian, và tốn thêm $O(N)$ không gian so với cách đã trình bày ở trên.

Bài toán 3: Đếm tổng số lượng ước trên một khoảng

Đề bài: Cho một số nguyên dương N . Tính: $\sum_{i=1}^N \tau(i)$ với $\tau(x)$ là số ước của x .


Thử áp dụng hai hướng giải quyết ở trên, ta thu được độ phức tạp lần lượt là $O(N\sqrt[3]{N})$ và $O(N \log N)$. Tuy vậy, bài toán này còn có một lời giải tốt hơn, có thời gian chạy dưới mức tuyến tính.

Dễ thấy với mỗi số d , sẽ tồn tại $\lfloor \frac{N}{d} \rfloor$ bội của d trong đoạn $[1, N]$. Vì vậy, nếu liệt kê tất cả các ước của các số từ 1 tới N , số d sẽ xuất hiện $\lfloor \frac{N}{d} \rfloor$ lần.

Do vậy, kết quả của bài toán sẽ trở thành:

$$\sum_{i=1}^N \tau(i) = \sum_{d=1}^N \left\lfloor \frac{N}{d} \right\rfloor$$

Tổng trên có thể được tính một cách dễ dàng trong $O(n)$. Tuy nhiên, dựa vào nhận xét sau, ta có thể tối ưu tính toán xuống $O(\sqrt{N})$:

Nhận xét: Với $d \leq \lfloor \sqrt{N} \rfloor$, phép tính $\lfloor \frac{N}{d} \rfloor$ nhận giá trị khác nhau với mỗi giá trị của d . Với $d > \lfloor \sqrt{N} \rfloor$, phép tính này nhận tổng cộng $\lfloor \sqrt{N} \rfloor$ hoặc $\lfloor \sqrt{N} \rfloor - 1$ giá trị khác nhau, các giá trị này không vượt quá $\lfloor \sqrt{N} \rfloor$ (Xem chứng minh nhận xét này tại [đây](#) ).

Dựa vào nhận xét trên, ta chỉ cần tính tổng $\lfloor \frac{N}{d} \rfloor$ tới $d = \lfloor \sqrt{N} \rfloor$, còn lại với mỗi $\lfloor \frac{N}{d} \rfloor$ ta đếm xem có bao nhiêu lần giá trị này xuất hiện rồi cộng vào tổng tích lũy.

```
1 | long long accCountDiv(long long n) {
2 |     long long ret = 0;
3 |     for (int i = 1; 1ll * i * i <= n; i++)
4 |         ret += n / i;
5 |     for (int i = 1; i < n / (int)sqrt(n); i++)
6 |         ret += 1ll * (n / i - n / (i + 1)) * i;
7 |     return ret;
8 | }
9 |
10 | void solution() {
11 |     long long n;
12 |     cin >> n;
13 |     cout << accCountDiv(n);
14 | }
```

Bài toán tính tổng các ước

Đề bài 1: Cho một số nguyên dương N . Tính tổng các ước của N .

Đề bài 2: Cho một số nguyên dương N . Cho Q truy vấn. Ở truy vấn thứ i , cần tìm tổng các ước của số A_i với $A_i < N$.

Một cách rất tự nhiên, ta có một lời giải đơn giản cho bài toán này giống như khi tìm số ước:

```
1 | int divSum(int n) {  
2 |     int ret = 0;  
3 |     for (int i = 1; i * i <= n; i++) {  
4 |         ret += (i + n / i) * (n % i == 0);  
5 |         if (i * i == n) ret -= i;  
6 |     }  
7 |     return ret;  
8 | }
```

Độ phức tạp thời gian của giải thuật trên là $O(\sqrt{N})$.

Tương ứng với cách thứ hai của bài toán trên, hàm tổng các ước cũng có một tính chất đặc biệt:

Định lý: Nếu một số N được phân tích ra thừa số nguyên tố thành: $N = \prod_{i=1}^k p_i^{m_i} = p_1^{m_1} \times p_2^{m_2} \dots \times p_k^{m_k}$ với p_i là các số nguyên tố, m_i là các số nguyên dương thì tổng các ước của N là:

$$\sigma(N) = \prod_{i=1}^k \frac{p_i^{m_i+1} - 1}{p_i - 1}$$

► Chứng minh (nhấn để hiện):

Với tính chất này, ta dễ dàng sử dụng sàng nguyên tố để tính tích trên với từng thừa số nguyên tố. Tùy vào bài toán mà độ phức tạp phù hợp sẽ là $O(\sqrt{N} \log \log \sqrt{N})$ hoặc $O(N \log \log N + Q \log N)$.

Cải tiến $O(\sqrt[3]{N})$ không thể áp dụng để tính tổng do nó không cho ta biết cụ thể các ước. Tuy nhiên, bài toán tính tổng các ước của một số duy nhất cho trước vẫn có thể giải trong $O(\sqrt[4]{N})$ bằng [thuật toán rho của Pollard](#) \square .

Chú ý thêm

- Độ phức tạp thời gian của các giải thuật cho cả hai bài toán cũng phụ thuộc vào các sàng nguyên tố được sử dụng. Một số cải tiến của sàng nguyên tố cũng làm tăng thêm hiệu quả

của các thuật trên.

- Hàm $\sigma(x)$ (tổng các ước) là một [hàm nhân tính](#) \square . Cụ thể, với hai số nguyên dương x và y nguyên tố cùng nhau, ta có $\sigma(x)\sigma(y) = \sigma(xy)$. Dựa vào tính chất này, nếu gặp khó khăn trong việc nhớ công thức tính $\sigma(N)$ dựa vào phân tích thừa số nguyên tố của N , ta có thể thử suy nghĩ lại với hướng trong bài viết đã liên kết.
- Hàm $\sigma_k(N) = \sum_{d|N} d^k$ cũng là một hàm nhân tính và có thể tính được bằng cách phân tích ra thừa số nguyên tố. Họ hàm này gọi là [hàm ước số](#) \square và cũng có một số tính chất đặc biệt. Với bài viết này, ta có $k = 0$ (hàm đếm số ước) và $k = 1$ (hàm tổng các ước).
- Các số thoả mãn $N = 2\sigma(N)$ được gọi là [số hoàn thiện \(perfect number\)](#) \square (hoặc hoàn chỉnh, hoàn hảo, ... tùy cách dịch). Một tính chất thú vị của loại số này là: Các số hoàn thiện chẵn có dạng $2^{p-1}(2^p - 1)$, với điều kiện p và $2^p - 1$ đều là số nguyên tố.

Bài tập áp dụng

- [Codeforces - Divisions](#) \square (Bài F)
- [CSES - Counting Divisors](#) \square
- [CSES - Sum of Divisors](#) \square
- [CSES - Divisor Analysis](#) \square
- [VNOJ - Educational Backtracking: Số ước số](#) \square

Tài liệu tham khảo

- [himanshuaju \(Codeforces\) - Counting Divisors of a Number in \[tutorial\]](#) \square .
- [Toán học Việt Nam - Chứng minh định lý về hàm tổng các ước số nguyên dương](#) \square
- [USACO Guide - Solution - Sum of Divisors \(CSES\)](#) \square