Chuyên đề A1: Số nguyên tố và ứng dụng

Ban Chuyên môn Tin học – Tổ chức The Gifted Battlefield nhiệm kỳ 2024-2025

Xuất bản vào Ngày 10 tháng 12 năm 2024

Tóm tắt nội dung

Số nguyên tố là một vẻ đẹp của toán học, mang nhiều các tính chất thú vị và hữu dụng. Trong tin học, ta thường hay dùng số nguyên tố để cải thiện thời gian chạy của các thuật toán liên quan tới tính toán các số ước. Việc học tính toán các số nguyên tố cũng là bước khởi đầu tốt để làm quen với tư duy toán học trong lập trình thi đấu.

Qua tài liệu, ta sẽ học được những thuật toán cơ bản về số nguyên tố như phân tích thừa số nguyên tố, đếm/tổng/tích ước, phi hàm Euler, Sàng nguyên tố và mở rộng, và những ước lượng liên quan tới số nguyên tố để ta tính độ phức tạp về thời gian,... Sau phần lý thuyết sẽ có một số bài tập cũng như tài liệu để bạn đọc có thể tham khảo và luyện tập thêm.

Mục lục

1 Số nguyên tố — **2**
- 1.1 Định nghĩa — 2
- 1.2 Phân tích thừa số nguyên tố — 2

2 Các hàm liên quan đến Thừa số nguyên tố — **3**
- 2.1 Đếm số lượng ước của một số — 4
- 2.2 Tổng các ước của một số — 4
- 2.3 Tích các ước của một số — 5

3 Sàng nguyên tố và Sàng nguyên tố mở rộng — **6**
- 3.1 Thuật toán Sàng nguyên tố (Sieve of Eratosthenes) — 6
- 3.2 Chứng minh độ phức tạp sàng nguyên tố — 8
- 3.3 Sàng nguyên tố trên đoạn — 9
- 3.4 Sàng nguyên tố mở rộng — 10

4 Ứng dụng — **13**
- 4.1 Tính ước chung lớn nhất — 13
- 4.2 Tính bội chung nhỏ nhất — 13

5 Bài tập ví dụ — **13**
- 5.1 Bài tập: Savrsen — 13
- 5.2 Bài tập: Chia hết — 15
- 5.3 Bài tập thêm: Số nguyên tố đối xứng — 17
- 5.4 Bài tập thêm: Khoảng cách — 17
- 5.5 Bài tập thêm: Chú gấu Tommy và các bạn — 18

6 Lời kết — **19**

1 Số nguyên tố

1.1 Định nghĩa

Định nghĩa 1.1

Số nguyên tố là số tự nhiên **lớn hơn 1** và **chỉ có đúng hai ước là 1 và chính nó**. Nói cách khác, số nguyên tố chỉ chia hết cho 1 và chính nó, không chia hết cho bất kỳ số nào khác.

Ví dụ: 2, 3, 5, 7, 11 là các số nguyên tố vì chúng chỉ chia hết cho 1 và chính chúng. Ngược lại, 4 không phải là số nguyên tố vì ngoài 1 và 4, nó còn chia hết cho 2.

Lưu ý: 0 và 1 **không** phải số nguyên tố

1.2 Phân tích thừa số nguyên tố

Định nghĩa 1.2

Phân tích thừa số nguyên tố là phân tích một số nguyên dương thành tích của các số nguyên tố sao cho khi nhân các thừa số nguyên tố này lại, ta sẽ thu được số ban đầu.

$$n = p_1^{a_1} \times p_2^{a_2} \times \cdots \times p_k^{a_k}$$

Trong đó: p_i là thừa số nguyên tố thứ i, a_i là số mũ tương ứng với p_i.

Lưu ý

Mỗi số nguyên dương có đúng 1 phân tích thừa số nguyên tố và với 2 số nguyên khác nhau, phân tích thừa số nguyên tố của chúng là khác nhau.

Thuật toán $O(\frac{\sqrt{n}}{\ln \sqrt{n}})$

Định lí 1.1

Đối với bất kỳ số nguyên n, nhiều nhất **chỉ có một thừa số nguyên tố của nó lớn hơn** $\sqrt{n}$.

Chứng minh. Nếu n có nhiều hơn 1 thừa số nguyên tố lớn hơn $\sqrt{n}$ thì tích của các thừa số này sẽ lớn hơn n ($\sqrt{n} \times \sqrt{n} = n$. Vì vậy, nếu $p_1 > \sqrt{n}$ và $p_2 > \sqrt{n}$ thì $p_1 \times p_2 > n$). Điều này là không thể, bởi vì tích của các thừa số nguyên tố của n phải đúng bằng n. ■

Vì vậy, để phân tích thừa số nguyên tố của một số, **chỉ cần kiểm tra các số nhỏ hơn hoặc bằng** $\sqrt{n}$. Sau đó, nếu còn lại một thừa số nguyên tố nữa, nó sẽ là thừa số duy nhất lớn hơn $\sqrt{n}$.

Chúng ta có thể tối ưu hóa thuật toán hơn nữa bằng cách chuẩn bị trước một danh sách các số nguyên tố không vượt quá N (trong đó N là giới hạn trên cho n). Với sự tối ưu hóa này, độ phức tạp thời gian của thuật toán giảm xuống còn $O(\frac{\sqrt{n}}{\ln \sqrt{n}})$.

```
void factorize(long long n) {
    // Duyệt qua toàn bộ số nguyên tố trong prList cho đến sqrt(n)
    for (int i = 0; i < prList.size() && 1LL * prList[i] * prList[i] <= n; i++) {
        // Nếu n không chia hết cho prList[i], bỏ qua nó
        if (n % prList[i])
            continue;

        // Nếu chia hết thì đếm số mũ của thừa số nguyên tố
        int pw = 0;
```

```
10          while (n % prList[i] == 0) {
11              pw++;
12              n /= prList[i];   // Chia n cho prList[i] cho đến không còn chia hết nữa
13          }
14
15          // Xử lý thừa số prList[i] với số mũ pw
16          // Ví dụ: in ra hoặc lưu (prList[i], pw)
17          // ...
18      }
19
20      // Sau khi lặp, nếu n không phải là 1, thì n chính là một thừa số nguyên tố > sqrt(n ban đầu)
21      if (n != 1) {
22          // Xử lý n như là một thừa số nguyên tố với số mũ 1
23          // ...
24      }
25  }
```

Tính chất kiểm tra ước số

Một tính chất quan trọng của phân tích thừa số nguyên tố là chúng ta có thể kiểm tra xem một số nguyên x có phải là ước của một số nguyên y hay không, bằng cách so sánh các số mũ trong phân tích thừa số nguyên tố của chúng.

Cụ thể, x là ước của y **khi và chỉ khi** mọi số mũ trong phân tích thừa số nguyên tố của x đều bé hơn hoặc bằng mọi số mũ tương ứng trong phân tích thừa số nguyên tố của y.

Ví dụ 1.1

Với $x = 18 = 2^1 \times 3^2$ và $y = 540 = 2^2 \times 3^3 \times 5^1$.

Để xác định xem x có phải là ước của y hay không, ta so sánh các số mũ của từng thừa số nguyên tố trong phân tích thừa số nguyên tố của x so với y:

- Với thừa số 2: số mũ của 2 trong x là 1, trong khi trong y là 2. Điều kiện $1 \leq 2$ thỏa mãn.
- Với thừa số 3: số mũ của 3 trong x là 2, trong khi trong y là 3. Điều kiện $2 \leq 3$ cũng thỏa mãn.

Vì mọi số mũ của các thừa số nguyên tố trong x đều nhỏ hơn hoặc bằng số mũ tương ứng trong y, x là ước của y.

2 Các hàm liên quan đến Thừa số nguyên tố

Trong phần này, ta quy ước một số n được biểu diễn dưới dạng thừa số nguyên tố như sau:

$$n = p_1^{a_1} \times p_2^{a_2} \times \cdots \times p_k^{a_k}$$

với:

- $p_1, p_2, ..., p_n$: lần lượt là các số nguyên tố.
- $a_1, a_2, ..., a_n$: lần lượt là luỹ thừa của các số nguyên tố tương ứng.

2.1 Đếm số lượng ước của một số

Định lí 2.1 Công thức đếm ước

Gọi $\tau(n)$ là số lượng ước của số n, ta có công thức:

$$\tau(n) = \prod_{i=1}^{k} (a_i + 1)$$

Chứng minh. Ở phần trước, ta đã biết một số nguyên m là ước của n khi mọi số mũ của m trong phân tích thừa số nguyên tố đều không quá số mũ tương ứng của n. Như vậy, với thừa số nguyên tố p_i của n, ta có thể chọn số mũ cho m là $0, 1, 2, \cdots, a_i$, tổng cộng là $a_i + 1$ cách chọn.

Vì các ước nguyên tố là độc lập với nhau, áp dụng quy tắc nhân, ta sẽ có công thức tính số ước của n như trên. ■

Code mẫu:

```
int countFactor (ll n) {
    int ans = 1;
    for (int i = 0; i < prList.size() && 1LL * prList[i] * prList[i] <= n; i++) {
        if (n % prList[i]) continue;
        int pw = 0;
        while (n % prList[i] == 0) pw++, n /= prList[i];
        ans *= (pw + 1);
    }
    if (n > 1) ans *= 2;
    return ans;
}
```

2.2 Tổng các ước của một số

Định lí 2.2 Công thức tính tổng ước

Gọi $\sigma(n)$ là tổng các ước của n, ta có công thức:

$$\sigma(n) = \prod_{i=1}^{k} (1 + p_i + p_i^2 \cdots + p_i^{a_i}) = \prod_{i=1}^{k} \frac{p_i^{a_i+1} - 1}{p_i - 1}$$

Chứng minh. Phát triển từ ý tưởng xây dựng hàm tính tích, thay vì ta đếm số lượng số mũ hợp lệ của các ước, ta tính đóng góp của từng số nguyên tố ứng với từng số mũ hợp lệ vào hàm $\sigma(n)$. Cụ thể, mọi ước nguyên tố p_i của n sẽ đóng góp một tổng $1 + p_i + p_i^2 + \cdots + p_i^{a_i}$. Khi nhân các tổng lại với nhau rồi nhân phân phối từng số hạng, ta sẽ thu được một tổng mà mỗi số hạng được biểu diễn dưới dạng phân tích thừa số nguyên tố của một ước của n.

Ngoài ra, ta nhận thấy rằng dãy $1, p_i, p_i^2, \cdots$ là một cấp số nhân với số hạng đầu là 1 và hệ số là p_i. Tổng của k số hạng đầu tiên được xác định bằng công thức:

$$1 + p_i + p_i^2 + \cdots + p_i^{k-1} = \frac{p_i^k - 1}{p_i - 1}$$

Kết hợp các phân tích trên, ta có công thức cho hàm $\sigma(n)$. ■

Code mẫu:

```
ll sumFactors (ll n) {
    ll ans = 1;
```

```
    for (int i = 0; i < prList.size() && 1LL * prList[i] * prList[i] <= n; i++) {
        if (n % prList[i]) continue;
        ll prPow = prList[i];
        while (n % prList[i] == 0) prPow *= prList[i], n /= prList[i];
        ans *= (prPow - 1) / (prList[i] - 1);
    }
    if (n > 1) ans *= (n * n - 1) / (n - 1);
    return ans;
}
```

2.3 Tích các ước của một số

Định lí 2.3 Công thức tính tích ước

Gọi $\mu(n)$ là tích các ước của số n, ta có công thức:

$$\mu(n) = n^{\tau(n)/2}$$

Chứng minh. Với một ước bất kì của n, gọi là m, ta biết rằng $\frac{n}{m}$ cũng là ước của n. Có thể thấy các ước của n tồn tại theo cặp có tích là n, ngoại trừ số chính phương có một ước lẻ là căn bậc hai của nó. Như vậy:

- Với n là số chính phương, có $\lfloor\frac{\tau(n)}{2}\rfloor$ cặp ước có tích là n và một ước lẻ là $\sqrt{n}$. Do đó, tích của các ước của n là $n^{\lfloor\tau(n)/2\rfloor} \cdot \sqrt{n}$. Biết rằng $\tau(n)$ là số lẻ nên ta có thể viết $\sqrt{n}$ lại thành $n^{0.5}$. Khi đó $\mu(n) = n^{\tau(n)/2}$.
- Với n không phải số chính phương, có đúng $\frac{\tau(n)}{2}$ cặp ước có tích là n. Như vậy $\mu(n) = n^{\tau(n)/2}$.

Cả hai trường hợp nêu trên đều cho ra công thức chung là $\mu(n) = n^{\tau(n)/2}$. ■

Code mẫu:

```
ll sqRoot (ll n) { // Tránh sai số gây ra bởi hàm sqrt()
    ll tmp = (ll)sqrt(n);
    for (ll i = tmp + 2; i >= tmp - 2; i--)
        if (i * i <= tmp) return i;
}

ll prodFactors (ll n) {
    ll ans = n, srt = sqRoot(n);
    bool isSquare = 0;
    if (srt * srt == n) ans = srt, isSquare = 1;

    for (int i = 0; i < prList.size() && 1LL * prList[i] * prList[i] <= n; i++) {
        if (n % prList[i]) continue;
        int pw = 1;
        while (n % prList[i] == 0) pw++, n /= prList[i];
        if (!isSquare) pw /= 2;

        ll cur = ans;
        for (int i = 1; i < pw; i++) ans *= cur;
    }
    if (n > 1) ans *= ans;
    return ans;
}
```

Tuy vậy, Phi hàm Euler là một kiến thức khó trong số học, không quá phổ biến trong các bài toán Lập trình thi đấu ở cấp 2.

3 Sàng nguyên tố và Sàng nguyên tố mở rộng

Ở phần trước, ta đã viết được thuật toán kiểm tra một số nguyên tố trong độ phức tạp $O\left(\frac{\sqrt{n}}{\ln\sqrt{n}}\right)$. Câu hỏi đặt ra là có thể kiểm tra số nguyên tố cho nhiều số cùng một lúc trong một khoảng nào đó, mà không cần phải xét riêng từng số hay không? Hơn nữa, ta có thể biến đổi thuật toán này để xử lý nhiều hàm khác liên quan đến phân tích thừa số nguyên tố một cách hiệu quả hay không?

3.1 Thuật toán Sàng nguyên tố (Sieve of Eratosthenes)

Ta xét bài toán như sau: Cho số nguyên n, hãy tạo ra mảng `prime[]` với $\texttt{prime}[i] = 1$ nếu i là số nguyên tố và ngược lại thì $\texttt{prime}[i] = 0$.

Ý tưởng

Ta đưa ra một số nhận xét nhỏ như sau:

- Rõ ràng, số 2 là số nguyên tố và mọi bội lớn hơn nó là hợp số. Như vậy $\texttt{prime}[2] = 1$ và $\texttt{prime}[2i] = 0$ với $i \geq 2$.
- Hơn nữa, số 3 cũng là số nguyên tố và mọi bội lớn hơn nó cũng là hợp số. Như vậy $\texttt{prime}[3] = 1$ và $\texttt{prime}[3i] = 0$ với $i \geq 2$.
- Tương tự, số 5 là số nguyên tố và mọi bội lớn hơn nó cũng là hợp số. Như vậy $\texttt{prime}[5] = 1$ và $\texttt{prime}[5i] = 0$ với $i \geq 2$.

Cứ như vậy, với mỗi số nguyên tố p ta thấy rằng mọi bội lớn hơn nó đều là hợp số. Thuật toán Sàng nguyên tố (Sieve of Erastosthenes) đã tận dụng chính chất này để tạo ra mảng `prime[]` một cách hiệu quả như sau:

Thuật toán 3.1 Sàng nguyên tố (Sieve of Eratosthenes)

1. Khởi tạo một mảng `prime[]` với $\texttt{prime}[i] = 1$ cho mọi $i \geq 2$.
2. Duyệt mọi p từ 2 đến n. Ta nhận thấy rằng nếu p là hợp số thì các ước nguyên tố của i chắc chắn đã được duyệt qua và đánh dấu $\texttt{prime}[p] = 0$. Như vậy, nếu $\texttt{prime}[p] = 1$ thì ta có thể kết luận rằng i là số nguyên tố.
3. Với mỗi số nguyên tố p, ta thực hiện đánh dấu các bội lớn hơn nó là hợp số, tức là gán $\texttt{prime}[p \cdot i] = 0$ với $i \geq 2$.

   Sau khi thực hiện thuật toán như trên, ta đã có mảng `prime[]` đúng như yêu cầu đề bài.

Ví dụ với $n = 50$, đầu tiên, ta khởi tạo mảng `prime[]` với $\texttt{prime}[i] = 1$ cho mọi $i \geq 2$. Sau đó, duyệt $p = 2$, do $\texttt{prime}[2] = 1$ nên 2 là số nguyên tố, ta tiến hành đánh dấu các bội $4, 6, 8, 10, 12, \cdots$ là hợp số.

Tiếp đến với $p = 3$, ta lại thấy 3 là số nguyên tố nên tiến hành đánh dấu các bội $6, 9, 12, 15, \cdots$ là hợp số. Lưu ý rằng với $= 4$, do 4 đã được đánh dấu là hợp số nên ta bỏ qua lượt duyệt này.

Lặp lại quá trình này với các số nguyên tố không quá 50 còn lại, ta sẽ có được mảng `prime[]` theo ý muốn.

1	2	3	4	5	6	7	8	9	10
11	12	13	14	15	16	17	18	19	20
21	22	23	24	25	26	27	28	29	30
31	32	33	34	35	36	37	38	39	40
41	42	43	44	45	46	47	48	49	50

Hình 3.1. Bước sàng nguyên tố với $p = 2$

1	2	3	4	5	6	7	8	9	10
11	12	13	14	15	16	17	18	19	20
21	22	23	24	25	26	27	28	29	30
31	32	33	34	35	36	37	38	39	40
41	42	43	44	45	46	47	48	49	50

Hình 3.2. Bước sàng nguyên tố với $p = 3$

Tuy cách làm này khá đơn giản nhưng tính hiệu quả của nó lại cao một cách bất ngờ. Ở phần chứng minh độ phức tạp bên dưới, ta sẽ biết rằng độ phức tạp của thuật toán Sàng nguyên tố như trên là $O(n \ln \ln n)$, với $\ln n$ là hàm logarit cơ số e.

Cài đặt

```
// Bước 1: Khởi tạo mảng prime[]
for (int i = 2; i <= n; i++) prime[i] = 1;
// Bước 2: Duyệt mọi p từ 2 đến n
for (int p = 2; p <= n; p++) {
    if (prime[p] == 0) continue;
    // Bước 3: Đánh dấu các bội lớn hơn p là hợp số
    for (int i = 2; p * i <= n; i++) prime[p * i] = 0;
}
```

Cải tiến 1: Sàng đến $\sqrt{n}$

Như đã tìm hiểu ở phần trước, mọi số tự nhiên n chỉ có tối đa một ước nguyên tố lớn hơn $\sqrt{n}$. Mở rộng hơn, ta có thể chứng minh rằng mọi hợp số n có ít nhất một ước nguyên tố không quá $\sqrt{n}$. Sử dụng nhận xét này, ta có thể cải tiến thuật toán Sàng nguyên tố bằng cách chỉ duyệt các p từ 2 đến $\sqrt{n}$. Sau khi thực hiện xong quá trình duyệt, mọi số nguyên $> \sqrt{n}$ chưa được duyệt chắc chắn sẽ là số nguyên tố.
Với cải tiến này, độ phức tạp của thuật toán sẽ được giảm xuống $O(n \ln \ln \sqrt{n})$ – một độ phức tạp rất gần tuyến tính.

1	2	3	4	5	6	7	8	9	10
11	12	13	14	15	16	17	18	19	20
21	22	23	24	25	26	27	28	29	30
31	32	33	34	35	36	37	38	39	40
41	42	43	44	45	46	47	48	49	50

Hình 3.3. Kết quả cuối cùng của thuật toán Sàng nguyên tố

Cải tiến 2: Chỉ xét các bội $p \cdot i$ với $p \leq i$

Cũng từ nhận xét như trên, ta còn có thể cải tiến thêm bằng cách chỉ duyệt $i \geq p$. Lý do là khi $p > i \Leftrightarrow p > \sqrt{p \cdot i}$, mà ta đã biết rằng sẽ tồn tại một số nguyên tố $p' \leq \sqrt{p \cdot i}$ là ước của $p \cdot i$, nên việc duyệt các bội $p \cdot i$ với $i < p$ là dư thừa.

Việc làm này không trực tiếp làm giảm độ phức tạp của thuật toán, tuy nhiên tốc độ thực hiện đoạn code sẽ được rút ngắn một cách đáng kể sau khi thêm cải tiến.

Kết hợp cả hai cả tiến nêu trên, ta có thuật toán Sàng nguyên tố cải tiến như sau:

Thuật toán 3.2 Sàng nguyên tố cải tiến

1. Khởi tạo một mảng `prime`[] với `prime`$[i] = 1$ cho mọi $i \geq 2$.
2. Duyệt mọi p từ 2 đến $\sqrt{n}$. Ta nhận thấy rằng nếu p là hợp số thì các ước nguyên tố của i chắc chắn đã được duyệt qua và đánh dấu `prime`$[p] = 0$. Như vậy, nếu `prime`$[p] = 1$ thì ta có thể kết luận rằng i là số nguyên tố.
3. Với mỗi số nguyên tố p, ta thực hiện đánh dấu các bội lớn hơn nó là hợp số, tức là gán `prime`$[p \cdot i] = 0$ với $i \geq p$.

   Sau khi thực hiện thuật toán như trên, ta đã có mảng `prime`[] đúng như yêu cầu đề bài.

```
for (int i = 2; i <= n; i++) prime[i] = 1;
// Chỉ duyệt các số nguyên tố không quá căn n
for (int p = 2; p * p <= n; p++) {
    if (prime[p] == 0) continue;
    // Chỉ duyệt các bội của p có dạng p * i (p <= i)
    for (int i = p; p * i <= n; i++) prime[p * i] = 0;
}
```

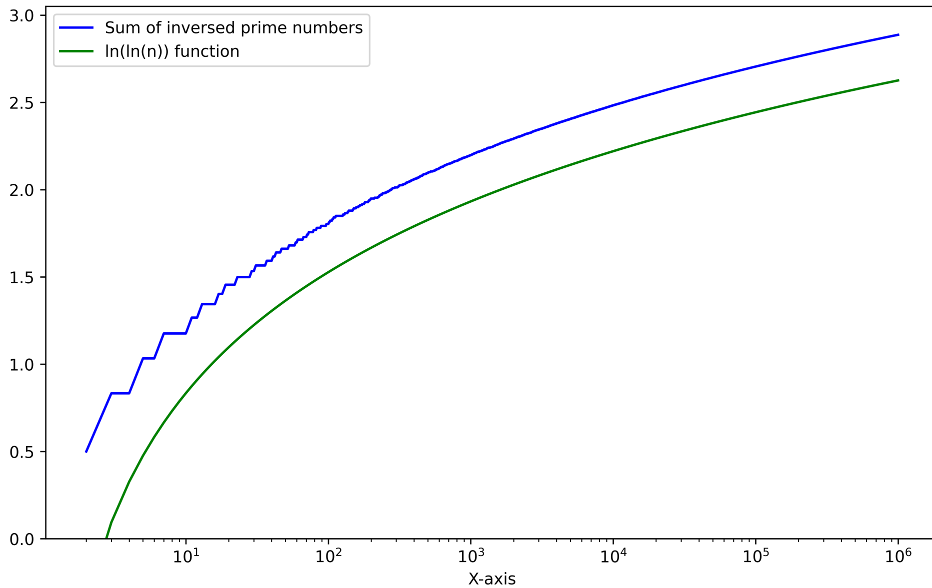
3.2 Chứng minh độ phức tạp sàng nguyên tố

Nếu phân tích độ phức tạp thuật toán trên dựa vào các vòng `for`, ta thấy rằng với mọi số nguyên tố p, độ phức tạp cho vòng lặp bên trong của biến i là $O\left(\left\lfloor \frac{n}{p} \right\rfloor\right)$. Do đó, ta có độ phức tạp sau:

$$O\left(\left\lfloor \frac{n}{2} \right\rfloor + \left\lfloor \frac{n}{3} \right\rfloor + \left\lfloor \frac{n}{5} \right\rfloor + \left\lfloor \frac{n}{7} \right\rfloor + \cdots\right) \approx O\left(n \cdot \sum_{p \leq \sqrt{n}} \frac{1}{p}\right)$$

với p là các số nguyên tố không quá $\sqrt{n}$.

Merten đã chứng minh được khi x đủ lớn, khoảng cách giữa $\sum_{p \le x} \frac{1}{p}$ và $\ln \ln x$ là không đáng kể và gần bằng hằng số $M \approx 0.261497$ (Hardy & Wright, 2008). Thật vậy, qua đồ thị dưới đây, ta thấy rằng hai hàm này tăng rất đều nhau và khi x lên đến khoảng 10^5 hay 10^6, khoảng cách của chúng gần như bằng hằng số M.



Hình 3.4. So sánh hàm tính tổng nghịch đảo số nguyên tố và $\ln \ln(x)$ (các giá trị trục x đã được căn chỉnh theo hàm $\log_{10}(x)$)

Như vậy, nếu bỏ qua hằng số M, ta thấy giá trị của hàm tính tổng nghịch đảo số nguyên tố và $\ln \ln(x)$ là xấp xỉ nhau. Từ đó, ta cũng rút ra rằng độ phức tạp của thuật toán Sàng nguyên tố cơ bản là $O(n \ln \ln \sqrt{n})$. Đối với một số tài liệu Tin học, người ta chấp nhận rằng độ phức tạp này tương đương $O(n \log \log n)$ (hàm log dùng hệ cơ số 2) do chênh lệch giữa hai độ phức tạp này là không quá lớn, hơn nữa hàm log cơ số 2 là một hàm rất phổ biến trong Tin học nói chung.

3.3 Sàng nguyên tố trên đoạn

Thay vì Sàng nguyên tố trên đoạn $[2; n]$, giả sử ta cần sàng trên đoạn $[L; R]$ với $R - L \le 10^6$ và $2 \le L \le R \le 10^{12}$. Với thuật toán Sàng nguyên tố truyền thống, ta duyệt mọi số nguyên tố trong khoảng $[2; \sqrt{R}]$. Với mỗi số nguyên tố p, ta đánh dấu các bội của p trong đoạn $[p^2; R]$ là hợp số. Như vậy, rõ ràng nếu $p^2 < L$ thì ta đang duyệt đoạn $[p^2; L)$ một cách phung phí.

Để khắc phục điều này, ở mỗi lượt sàng, ta sẽ bắt đầu với bội bé nhất của p nhưng không bé hơn $\max(L, p^2)$. Cụ thể hơn, ta bắt đầu duyệt i từ $\max(\lceil \frac{L}{p} \rceil, p)$. Như vậy, mỗi vòng `for` của biến i có độ phức tạp $O(\lfloor \frac{R-L+1}{p} \rfloor)$ và tổng độ phức tạp là $O((R - L + 1) \ln \ln \sqrt{R})$.

Ngoài ra, với thuật toán Sàng nguyên tố trên đoạn, do ta có thể không có đầy đủ thông tin cho các số trong khoảng $[2; \sqrt{R}]$ để xác định xem các số p có phải số nguyên tố hay không, ta cần thêm một bước tiền xử lý để tạo ra danh sách các số nguyên tố không quá $\sqrt{R}$. Điều này có thể được thực hiện bằng thuật toán Sàng nguyên tố truyền thống trong $O(\sqrt{R} \ln \ln \sqrt[4]{R})$.

```
// chuẩn bị danh sách các số nguyên tố <= sqrt(R)
ll sqrtR = sqrt(R);
for (int i = 2; i <= sqrtR; i++) sieve[i] = 1;
for (int p = 2; p * p <= sqrtR; p++) {
    if (!sieve[p]) continue;
    for (int i = p; p * i <= sqrtR; i++) sieve[p * i] = 0;
}
vector<ll> prList;
```

```cpp
for (int i = 2; i <= sqrtR; i++)
    if (sieve[i]) prList.push_back(i);

// sàng nguyên tố trên đoạn
for (ll i = 0; i <= R - L; i++) sieve[i] = 1; // các chỉ số được giảm đi L để tránh tràn mảng
for (ll p : prList) {
    ll divi = L / p + (L % p > 0 ? 1 : 0);
    for (ll i = max(divi, p); p * i <= R; i++) sieve[p * i - L] = 0;
}
```

3.4 Sàng nguyên tố mở rộng

Có thể thấy, bản chất thuật toán Sàng nguyên tố là duyệt qua các số nguyên tố trong khoảng cần sàng, sau đó tính đóng góp của số này vào giá trị cần tính của các bội của nó. Với nhận xét này, ta còn có thể sử dụng Sàng nguyên tố để tính nhiều giá trị khác nhau liên quan đến số nguyên tố (ví dụ như hàm $\tau(n)$, $\sigma(n)$ hay $\varphi(n)$), chứ không đơn giản là kiểm tra số nguyên tố.

Tùy vào đặc điểm của giá trị cần tính, ta sẽ chia Sàng nguyên tố mở rộng thành 3 dạng thường gặp:

Xét các ước nguyên tố phân biệt, không lấy số mũ

Đây là dạng Sàng nguyên tố gần nhất với thuật toán sàng nguyên tố kinh điển, ý tưởng chính vẫn là duyệt qua các số nguyên tố p, sau đó tính đóng góp của nó vào các bội (có dạng là $p \cdot i$) của p và không cần xét đến số mũ của p.

Ví dụ 3.1

Tính **ước nguyên tố lớn nhất** cho mọi số nguyên từ 2 đến n ($n \leq 10^7$).

Để bắt đầu, ta khởi tạo một mảng pr[] với giá trị ban đầu là $\texttt{pr}[i] = i$. Với mọi số nguyên tố p, ta cập nhật giá trị cho các bội của p, tức gán $\texttt{pr}[2p]$, $\texttt{pr}[3p]$,... cho p. Dễ thấy, số nguyên tố cuối cùng làm thay đổi một giá trị $\texttt{pr}[i]$ nào đó cũng chính là ước nguyên tố lớn nhất của nó.

Lưu ý, với bài toán này, do phải duyệt đến ước nguyên tố lớn nhất của từng số nguyên, ta phải duyệt mọi số nguyên tố p từ 2 đến n (thay vì đến $\sqrt{n}$), và duyệt mọi bội khác p của p (tức là duyệt $p \cdot i$ với mọi $i \geq 2$ thay vì $p \leq i$).

```cpp
    for (int i = 2; i <= n; i++) pr[i] = i;
    for (int p = 2; p <= n; p++) {
        if (pr[p] < p) continue; // p là hợp số
        for (int i = 2; p * i <= n; i++) pr[p * i] = p;
    }

```

Thuật toán trên có độ phức tạp thời gian là $O(n \ln \ln n)$ và bộ nhớ $O(n)$.
Với mảng pr[], ta hoàn toàn có thể phân tích thừa số nguyên tố cho một số nguyên a ($a \leq 10^7$) bằng cách liên tục lưu lại ước nguyên tố lớn nhất của a rồi chia a cho giá trị này, cho đến khi a bằng 1. Dễ thấy, số lần lặp lại thao tác này sẽ bằng đúng số ước nguyên tố của a.

Xét các ước nguyên tố phân biệt, lấy số mũ

Ta phát triển từ ý tưởng của dạng Sàng nguyên tố chỉ xét các ước nguyên tố phân biệt, không lấy số mũ. Nhiệm vụ của chúng ta là, khi đã cố định được một số nguyên tố p, ta cần tính số mũ của p trong mọi bội của p. Nói một cách toán học hơn, gọi $v_p(a)$ là số mũ của p trong phân tích thừa số nguyên tố của a, ta cần tính nhanh $v_p(p \cdot i)$. Dễ thấy, do $v_p(p) = 1$ nên:

$$v_p(p \cdot i) = 1 + v_p(i)$$

Như vậy, ta có thể tính trước $v_p(i)$ bằng quy hoạch động, sau đó suy ra số mũ của p trong $p \cdot i$ theo công thức liên hệ như trên.

$$v_p(i) = \begin{cases} v_p(\frac{i}{p}) + 1 & p \mid i \\ 0 & p \nmid i \end{cases}$$

Để hiểu rõ hơn, ta lại xem xét một ví dụ liên quan.

Ví dụ 3.2

Tính **tổng các ước nguyên dương** của mọi số nguyên từ 2 đến n ($n \leq 10^7$).

Như đã tìm hiểu ở phần trên, chúng ta có hai công thức xác định hàm $\sigma(n)$. Để tiện cho phần cài đặt, mình sẽ sử dụng công thức:

$$\sigma(n) = \prod_{i=1}^{k} (1 + p_i + \cdots + p_i^{a_i})$$

Để có thể tính $1 + p_i + \cdots + p_i^{a_i}$ trong $O(1)$, trước khi duyệt các bội của p, ta cần tính trước tổng này cho số mũ lớn nhất có thể và lưu lại trong một mảng/vector. Khi duyệt bội của p, ta chỉ việc trích xuất lại dữ liệu trong $O(1)$.

```
    for (int i = 1; i <= n; i++) sumFactor[i] = 1;
    for (ll p = 2; p <= n; p++) {
        if (sumFactor[p] != 1) continue; // p là hợp số

        // tính trước các giá trị 1 + p + p^2 + ...
        vector<ll> power(1, 1), sumPower(1, 1);
        while (power.back() * p <= n) {
            power.push_back(power.back() * p);
            sumPower.push_back(sumPower.back() + power.back());
        }

        // tính đóng góp của số nguyên tố p vào các bội của nó
        for (ll i = 1; p * i <= n; i++) {
            vp[i] = (i % p ? 0 : vp[i / p] + 1);
            sumFactor[p * i] *= sumPower[vp[i] + 1];
        }
    }

```

Thuật toán trên có độ phức tạp thời gian $O(n \ln \ln n)$ và bộ nhớ $O(n)$.

Xét ước nguyên dương

Dạng thứ ba của Sàng nguyên tố mở rộng là dạng xét mọi ước nguyên dương (không nhất thiết là số nguyên tố) của một số bất kì. Đối với dạng này, ta chỉ đơn giản là duyệt là duyệt mọi số nguyên p rồi duyệt các bội của p. Với dạng Sàng nguyên tố này, ta thường phải xử lý riêng một trường hợp đặc biệt là số chính phương, vì căn bậc hai của chúng không tồn tại theo cặp có tích là số chính phương đó.

Ví dụ 3.3

Dạng Sàng nguyên tố này có thể được sử dụng để xử lý bài toán đã được tìm hiểu ở phần trên, đó là tính tổng các ước nguyên dương của mọi số nguyên từ 2 đến n ($n \leq 3 \cdot 10^6$). Điểm khác biệt là thuật toán này có độ hiệu quả cũng như khả năng áp dụng vào nhiều dạng bài tập khác nhau kém hơn thuật toán nêu trên, bù lại, phần code sẽ ngắn gọn và dễ hiểu hơn.

Sử dụng lại nhận xét đã chứng minh ở phần trước: Với một số n bất kì, mọi ước của chúng đều tồn tại theo cặp có tích là n (trừ trường hợp căn bậc hai của số chính phương), trong đó, một trong hai ước đó chắc chắn không vượt quá $\sqrt{n}$. Từ đó, ta có thể duyệt các số nguyên p từ 1 đến $\sqrt{n}$, tính riêng trường hợp số chính phương p^2 rồi xét đóng góp của p vào các bội có dạng $p \cdot q$ với $p < q$.

```
for (int p = 1; p * p <= n; p++) {
    sumFactor[p * p] += p;
    for (int q = p + 1; p * q <= n; q++) sumFactor[p * q] += p + q;
}

```

Có thể lấy, độ phức tạp của thuật toán là:

$$O\left(\frac{n}{1} + \frac{n}{2} + \frac{n}{3} + \cdots + \frac{n}{\sqrt{n}}\right)$$

Để rút gọn độ phức tạp trên, ta nhận thấy công thức trên chính là tích của n và số harmonic thứ $\lfloor\sqrt{n}\rfloor$. Công thức xác định số harmonic thứ k là:

$$H_k = \sum_{1 \leq i \leq k} \frac{1}{i}$$

Người ta đã chứng minh được cận trên của số harmonic thứ k là $\ln k$. Như vậy, có thể thấy độ phức tạp của thuật toán trên chính là $O(n \cdot H_{\lfloor\sqrt{n}\rfloor}) \approx O(n \ln \sqrt{n})$.

Ngoài ra, ta còn có thể biến đổi Sàng nguyên tố để tạo ra danh sách các ước nguyên dương trong khoảng từ 1 đến n ($n \leq 10^6$). Ý tưởng cho thuật toán này cũng tương tự thuật toán tính tổng các ước nguyên dương được đã được nêu ở phần trước nhưng thay vì cộng `sumFactor`$[pq]$ cho $p + q$, ta thêm p và q vào danh sách ước của pq.

Hệ quả 3.1

Qua việc chứng minh độ phức tạp của thuật toán trên, ta cũng chứng minh được rằng tổng số ước của các số từ 1 đến n là xấp xỉ $O(n \ln \sqrt{n})$. Nói một cách toán học hơn:

$$\sum_{1 \leq i \leq n} \tau(i) \approx O(n \ln \sqrt{n})$$

4 Ứng dụng

Ngoài các bài toán liên hệ trực tiếp, số nguyên tố còn có nhiều ứng dụng khác trong số học cũng như toán học/tin học nói chung. Trong phần này, ta sẽ cùng tìm hiểu những ứng dụng đó.

4.1 Tính ước chung lớn nhất

Để tính ước chung lớn nhất của hai số nguyên dương a, b, ta xét từng ước nguyên tố của a và b, gọi là p. Giả sử số mũ của p trong a, b lần lượt là $v_p(a), v_p(b)$, số mũ của p trong $\gcd(a, b)$ sẽ là $\min(v_p(a), v_p(b))$.
Tương tự, với ước chung lớn nhất của n số nguyên $a_1, a_2, \cdots, a_n$, số mũ của p trong $\gcd(a_1, a_2, \cdots, a_n)$ là $\min_{1\le i\le n} v_p(a_i)$.

Ví dụ 4.1

Tính $\gcd(180, 4200, 27000)$.

Phân tích thừa số nguyên tố của $180, 4200, 27000$ lần lượt là:

$$180 = 2^2 \cdot 3^2 \cdot 5^1$$
$$4200 = 2^3 \cdot 3^1 \cdot 5^2 \cdot 7^1$$
$$27000 = 2^3 \cdot 3^3 \cdot 5^3$$

Với mỗi ước nguyên tố, ta lấy số mũ nhỏ nhất, như vậy, $\gcd(180, 4200, 27000) = 2^3 \cdot 3^1 \cdot 5^1 \cdot 7^0 = 120$.

4.2 Tính bội chung nhỏ nhất

Cách tính bội chung nhỏ nhất bằng phân tích thừa số nguyên tố khá giống ước chung lớn nhất, nhưng ta sẽ lấy số mũ lớn nhất thay vì là bé nhất. Cụ thể, với bội chung nhỏ nhất của n số nguyên $a_1, a_2, \cdots, a_n$, số mũ của p trong $\text{lcm}(a_1, a_2, \cdots, a_n)$ là $\max_{1\le i\le n} v_p(a_i)$.

Ví dụ 4.2

Tính $\text{lcm}(56, 180, 300)$

Phân tích thừa số nguyên tố của $56, 180, 300$ lần lượt là:

$$56 = 2^3 \cdot 7^1$$
$$180 = 2^2 \cdot 3^2 \cdot 5^1$$
$$27000 = 2^2 \cdot 3^1 \cdot 5^2$$

Với mỗi ước nguyên tố, ta lấy số mũ lớn nhất, như vậy, $\text{lcm}(56, 180, 300) = 2^3 \cdot 3^2 \cdot 5^2 \cdot 7^1$.

5 Bài tập ví dụ

5.1 Bài tập: Savrsen

- Link đề gốc: `https://oj.uz/problem/view/COCI17_savrsen`

Mô tả bài toán

Gọi $\sigma(n)$ là tổng các ước của n, và $f(i) = |2 \cdot n - \sigma(n)|$. Cho $l \le r \le 10^7$, tính $\sum_{i=l}^{r} f(i)$.

Ý tưởng

- Khi gặp $\sigma(n)$, ta cần tìm cách tính nhanh cho 10^7 số này. Thuật toán $O(\sqrt{n})$ sẽ quá chậm khi thực hiện 10^7 lần.
- Thay vào đó, có thể dùng công thức tính tổng trong $O(\log n)$ để tối ưu hóa.

Lời giải

- Áp dụng sàng nguyên tố để phân tích số x thành các thừa số nguyên tố trong $O(\log x)$, sau đó sử dụng công thức tổng ước.
- Nhắc lại công thức: với $n = p_1^{e_1} \cdot p_2^{e_2} \cdots p_k^{e_k}$, ta có:

$$\sigma(n) = \frac{p_1^{e_1+1} - 1}{p_1 - 1} \cdot \frac{p_2^{e_2+1} - 1}{p_2 - 1} \cdots \frac{p_k^{e_k+1} - 1}{p_k - 1}$$

Phân tích độ phức tạp

- Hàm $\sigma(n)$ có thể tính trong $O(\log n)$ dẫn đến việc tính được hàm $f(n)$ trong $O(\log n)$, và ta sẽ tính $r - l$ lần hàm này.
- Độ phức tạp sẽ là:
  - Tiền xử lý: $O(n \log n)$ cho sàng nguyên tố
  - Bài toán chính: $O((r - l + 1) \cdot \log n)$

Cài đặt

```cpp
#include <bits/stdc++.h>
using namespace std;

const int MAXN = 1e7;
int sieve[MAXN + 5];

void init()
{
    for(int i = 1; i <= MAXN; i++) sieve[i] = i;

    for(int i = 2; i*i <= MAXN; i++) {
        if(sieve[i] == i) {
            for(int j = i * i; j <= MAXN; j += i)
                if(sieve[j] == j) sieve[j] = i;
        }
    }
}
long long f(int n) {
    long long res = 1;

```

```cpp
    while(n > 1) {
        int d = sieve[n];
        int e = 1;
        while(n % d == 0) e++, n /= d;
        long long sum = 0, pw = 1;
        while(e--) {
            sum += pw;
            pw *= d;
        }
        res *= sum;
    }
    return res;
}

int main()
{
    init();
    int l, r; cin >> l >> r;
    long long res = 0;
    for(int x = l; x <= r; x++)
        res += abs(x - (f(x) - x));
    cout << res << '\n';
    return 0;
}
```

Mở rộng (một cách làm nhanh hơn)

Thay vì sử dụng hàm để sàng nguyên tố bình thường, ta có thể sử dụng sàng mở rộng thành sàng tổng ước (được nêu ở trong phần 3).

- Sàng tổng ước trong $O(n \ln \sqrt{n})$
- Nếu như vậy, khi tính hàm $f(x)$ chỉ cần độ phức tạp là $O(1)$

Vậy độ phức tạp của lời giải là, tiền xử lý: $O(n \ln \sqrt{n})$, bài toán chính là $O(r - l)$.

So sánh hai lời giải

- Lời giải 1: `https://oj.uz/submission/1105990`
- Lời giải 2: `https://oj.uz/submission/1106000`

Qua đây ta có thể thấy được lời giải sau (`201ms`) có sự tối ưu để nhanh hơn 2 lần so với lời giải cũ (`420ms`). Trong một số bài cần tối ưu chặt chẽ thì có thể sử dụng lời giải 2 nên tiết kiệm nhiều thời gian nhất có thể.

5.2 Bài tập: Chia hết

Mô tả bài toán

Cho số nguyên dương m và dãy a gồm n số nguyên dương $a_1, a_2, \ldots, a_n$. Nếu $m!$ chia hết cho $\prod_{i=1}^{n} a_i$ thì in ra YES ngược lại thì in ra NO.

Giới hạn:

- $m \leq 10^9$.
- $n \leq 10^6$.
- $a_i \leq 10^6$ $(1 \leq i \leq n)$.

Lời giải

Một số nguyên dương x có thể được phân tích thành các thừa số nguyên tố: $x = {p_1}^{e_1} \times {p_2}^{e_2} \times \ldots \times {p_k}^{e_k}$. Gọi $v_p(n)$ là số mũ đúng của số nguyên tố p trong phân tích thừa số nguyên tố của số n. Với 2 số nguyên dương x, y, xét riêng từng thừa số số nguyên tố p thì y chia hết cho x **khi và chỉ khi** $v_p(y) \geq v_p(x)$.

Do đó ta đưa bài toán trên thành kiểm tra xem $m!$ có chia hết cho các thừa số nguyên tố của $\prod_{i=1}^{n} a_i$ hay không.

Ta phân tích tích các phần tử của mảng a thành các thừa số nguyên tố: $\prod_{i=1}^{n} a_i = {p_1}^{q_1} \times {p_2}^{q_2} \times \ldots \times {p_k}^{q_k}$. Với mỗi $p_i^{q_i}$ $(1 \leq i \leq k)$ ta kiểm tra xem $m!$ có chia hết cho $p_i^{q_i}$ hay không.

Các bước giải:

- Phân tích $\prod_{i=1}^{n} a_i$ thành thừa số nguyên tố bằng sàng nguyên tố, với mỗi số nguyên tố p, tính tổng cơ số mũ q của p trong mọi phần tử a_i. Độ phức tạp $O(n \cdot \ln(n))$.
- Với mỗi số nguyên tố p_i, tính cơ số mũ q' của nó trong $m!$ bằng duyệt các số $p_i, 2p_i, 3p_i, \cdots$ rồi tính số mũ của p_i trong từng số đó. Do mọi bội của p_i đều chứa ít nhất một thừa số nguyên tố p_i nên ta cần duyệt qua không quá q_i bội của p_i. Như vậy, tổng độ phức tạp cho bước này là $O(\sum q_i)$, trường hợp tệ nhất thì độ phức tạp này tương đương $O(n \log n)$.
- Nếu số mũ q' của p trong $m!$ lớn hơn hoặc bằng số mũ q của p trong $\prod_{i=1}^{n} a_i$ thì $m!$ chia hết cho p^q.

Độ phức tạp: $O(n \cdot \ln(n) + \sum q_i)$

Cài đặt

```
#include <bits/stdc++.h>
#define int long long
using namespace std;

const int MX = 1000005;
int n, m;
int a[MX];
int c[MX];
int q[MX];
bool isP[MX];

int32_t main(){
    ios_base::sync_with_stdio(false);
    cin.tie(0);

    cin >> n >> m;
    for(int i=1; i<=n; i++) cin >> a[i];

```

```
    for(int i=1; i<=n; i++) c[a[i]]++;

    isP[0] = isP[1] = 1;
    for(int i=2; i<MX; i++) if(!isP[i]){
        for(int j=i, t=i; j<MX; j+=i, t++){
            isP[j] = 1;
            q[i] += t/i * c[j];
        }
        isP[i] = 0;
    }

    bool flag = 1;
    for(int i=1; i<MX; i++) if(!isP[i]){
        for(int j=i; j<=m; j*=i) q[i] -= m/j;
        if(q[i] > 0) flag = 0;
    }

    if(flag) cout << "YES" << endl;
    else cout << "NO" << endl;

    return 0;
}
```

5.3 Bài tập thêm: Số nguyên tố đối xứng

Mô tả đề bài

Số nguyên tố đối xứng là một số nguyên tố bằng trung bình cộng của 2 số nguyên tố liền trước và liền sau nó. Với P_i là số nguyên tố thứ i. Một số nguyên tố đối xứng khi thỏa: $P_i = \dfrac{P_{i-1} + P_{i+1}}{2}$.

Ví dụ 5.1

Dãy số nguyên tố: 2, 3, 5, 7, 11, 13, ...

5 là số nguyên tố đối xứng. Vì số nguyên tố liền trước của 5 là 3, số nguyên tố liền sau của 5 là 7.

Mà: $\dfrac{3+7}{2} = 5$.

10 số nguyên tố đối xứng đầu tiên là: 5, 53, 157, 173, 211, 257, 263, 373, 563, 593.

Yêu cầu: Nhập vào số nguyên dương n $(0 < n \leq 20000)$. Hãy cho biết số nguyên tố đối xứng thứ n.

Gợi ý

- Xây dựng một mảng để lưu lại các số nguyên tố (Áp dụng sàng nguyên tố)
- Duyệt qua từng số nguyên tố trong mảng đã xây dựng cho đến khi tìm được số nguyên tố đối xứng thứ n.

5.4 Bài tập thêm: Khoảng cách

- Link đề gốc: `https://oj.lequydon.net/problem/dnum`

Mô tả đề bài

Với hai số nguyên dương a, b, ta định nghĩa **khoảng cách** giữa a và b là số phép **nhân với một số nguyên tố** hoặc **chia hết với một số nguyên tố** để số a thành số b.

Ví dụ: Khoảng cách giữa 100 và 360 bằng 4 vì $100/5 \times 2 \times 3 \times 3 = 360$.

Yêu cầu: Cho T $(1 \leq T \leq 10^5)$ test-cases. Tính khoảng cách giữa hai số a, b $(1 \leq a, b \leq 10^6)$ cho trước.

Gợi ý

Ví dụ xét $a = 88$, $b = 999$

Đầu tiên phân tích a và b thành thừa số nguyên tố:

- $88 = 2 \times 2 \times 2 \times 11$.
- $999 = 3 \times 3 \times 3 \times 37$.

Để biến số a thành số b, thì ta phải biến đổi thừa số nguyên tố của số a sao cho giống thừa số nguyên tố của số b.

Gọi tập A là tập lưu thừa số nguyên tố của số nguyên dương a, tập B là tập lưu thừa số nguyên tố của số nguyên dương b.

$$\Rightarrow A = \{2, 2, 2, 11\}, B = \{3, 3, 3, 11\}.$$

Khoảng cách giữa a và b sẽ bằng số phần tử của tập $A \setminus B$ + số phần tử của tập $B \setminus A$.

5.5 Bài tập thêm: Chú gấu Tommy và các bạn

Mô tả đề bài

Cho dãy n số nguyên dương $x_1, x_2, \ldots, x_n$ và m truy vấn, mỗi truy vấn được cho bởi 2 số nguyên l_i,r_i. Cho một hàm $f(p)$ trả về số lượng các số x_k là bội của p. Câu trả lời cho truy vấn l_i,r_i là tổng $\sum_{p \in S(l_i, r_i)} f(p)$, trong đó $S(l_i, r_i)$ là tập các số nguyên tố trong đoạn $[l_i, r_i]$.

Hãy giúp chú gấu Tommy giải bài toán!

Giới hạn:

- $1 \leq n \leq 10^5$.
- $2 \leq x_i \leq 10^7$.
- $1 \leq m \leq 50000$.
- $2 \leq l_i \leq r_i \leq 2 \times 10^9$.

Gợi ý

- Xây dựng mảng c[i] là số lần xuất hiện của giá trị i trong dãy số.
- Dùng sàng nguyên tố, kết hợp để tính các giá trị $f(i)$ với i là số nguyên tố trong đoạn [1,10^7].
- Áp dụng mảng cộng dồn, xây dựng mảng sum[i] lưu lại tổng của $f[1] + f[2] + \ldots + f[i]$.
- Bây giờ, ta có thể tìm ra được đáp án trong đoạn $[l_i, r_i]$ với độ phức tạp O(1), đáp án sẽ là sum[r] - sum[$l-1$].

6 Lời kết

Sau những bài học rất thú vị về số nguyên tố, mình tin rằng bạn sẽ học được nhiều thứ về số nguyên tố cũng như những ứng dụng của nó trong tin học. Và như Paul Erdos từng nói:

> *"Vẻ đẹp của các số nguyên tố là chúng luôn ở đó, chờ đợi"*

Hãy tiếp tục tìm hiểu và khám phá thêm về số nguyên tố vì những điều thú vị, tuyệt đẹp đang chờ bạn ở phía trước!

References

Hardy, G. H., & Wright, E. M. (2008). An introduction to the theory of numbers (6th ed.). Oxford University Press.