

Sàng nguyên tố

Sàng nguyên tố

Người viết:

- ▶ Nguyễn Minh Hiền - Trường Đại học Công nghệ, ĐHQGHN

Reviewer:

- ▶ Nguyễn Đức Kiên - Trường Đại học Công nghệ, ĐHQGHN
- ▶ Cao Thanh Hậu - Trường Đại học Khoa học Tự nhiên, ĐHQG-HCM
- ▶ Nguyễn Minh Nhật - Trường THPT chuyên Khoa học Tự nhiên, ĐHQGHN

Khi cần tìm ra các số nguyên tố từ 1 đến n , ta có thể duyệt từng số và kiểm tra tính nguyên tố của nó. Và ý tưởng đó cho ta một thuật toán $O(n\sqrt{n})$.

Tuy nhiên, một nhà toán học cổ Hy Lạp tên là Eratosthenes đã "phát minh" ra một "thuật toán" hiệu quả hơn. Ban đầu, Eratosthenes đã lấy lá cọ và ghi tất cả các số từ 2 cho đến 100. Sau đó, ông đã chọc thủng các hợp số và giữ nguyên các số nguyên tố. Bảng số nguyên tố còn lại trông rất giống một cái sàng. Cho đến ngày nay, "thuật toán" này được phổ biến rộng rãi với cái tên **sàng nguyên tố Eratosthenes**.

Sàng nguyên tố Eratosthenes (Sieve of Eratosthenes)

Hướng tiếp cận

- ▶ Ban đầu, ta cho tất cả các số từ 2 đến n vào sàng và đánh dấu tất cả các số. (Các số không được đánh dấu sau cùng sẽ bị loại khỏi sàng).
- ▶ Duyệt lần lượt các số từ 2 đến n . Nếu số đang xét:
 - ▶ Đã được đánh dấu \Rightarrow *số nguyên tố*: ta bỏ đánh dấu tất cả các bội (khác chính nó) của số nguyên tố này để loại các bội ấy ra khỏi sàng.
 - ▶ Không được đánh dấu \Rightarrow *hợp số*: ta bỏ qua số này.
- ▶ Sau khi duyệt xong, các số còn lại trong sàng, hay nói cách khác các số được đánh dấu là số nguyên tố.

Dưới đây là hình minh họa cho thuật toán trên. *Nguồn: [CP-Algorithm](#)*



Code C++ minh họa

```

1  const int maxn = 1000000 + 5; //10^6 + 5
2  bool is_prime[maxn]; // mảng bool khởi tạo với các giá trị fals
3  void sieve(int n){
4      // Đánh dấu các số từ 2 đến n đều là số nguyên tố
5      for (int i = 2; i <= n; i++)
6          is_prime[i] = true;
7      for (int i = 2; i <= n; i++) {
8          if (is_prime[i]) {
9              for (int j = i * 2; j <= n; j += i)
10                 // Bỏ đánh dấu tất cả các số không phải số nguyên t
11                 is_prime[j] = false;
12          }
13      }
14  }

```

Độ phức tạp thời gian là $O\left(n \times \left(\frac{1}{2} + \frac{1}{3} + \dots + \frac{1}{p}\right)\right)$ với p là số nguyên tố $\leq n$.

Đến đây, bạn đọc có thể tham khảo [Định lý Merten 2](#) để rút gọn độ phức tạp:

$$O\left(n \times \left(\frac{1}{2} + \frac{1}{3} + \dots + \frac{1}{p}\right)\right) = O(n \log(\log n))$$

Độ phức tạp thời gian: $O(n \log \log n)$

Độ phức tạp không gian: $O(n)$

Nhận xét:

Xét $X = k \cdot p$ là bội của số nguyên tố p .

Nếu như $p < X < p^2$, ta có $1 < k < p$. Ta suy ra k phải có một ước nguyên tố nhỏ hơn p .

Vì thế, $X = k \cdot p$ đã bị sàng loại đi trong các vòng lặp trước đó và ta **chỉ cần xét $X \geq p^2$** .

Dựa vào Nhận xét trên, ta có cải tiến như sau:

```

1  const int maxn = 1000000 + 5; //10^6 + 5
2  bool is_prime[maxn];
3  void Eratosthenes(int n){
4      for (int i = 2; i <= n; i++)
5          is_prime[i] = true;
6      for (int i = 2; i * i <= n; i++) {
7          if (is_prime[i]) {
8              // j sẽ bắt đầu chạy từ i * i
9              for (int j = i * i; j <= n; j += i)
10                 is_prime[j] = false;
11          }
12      }
13  }
```


Độ phức tạp thời gian sau khi cải tiến vẫn là $O(n \log \log n)$. Tuy nhiên, số phép tính đã giảm đi đáng kể.

Lưu ý:

- Trong đoạn code trên, vì j chạy từ $i \times i$ đến n , nên i chỉ chạy từ 1 đến \sqrt{n} . Ngoài ra, ở đây, ta sử dụng điều kiện $i * i \leq n$ thay vì sử dụng $i \leq \text{sqrt}(n)$ bởi hàm `sqrt()` chạy lâu hơn so với phép nhân số nguyên.
- Nếu vẫn muốn sử dụng hàm `sqrt()`, ta phải tránh việc phải tính lại `sqrt(n)` mỗi lần lặp:

```

1  int nsqrt = sqrt(n);
2  for (int i = 2; i <= nsqrt; i++)
```

Dưới đây là hình minh họa cho cải tiến trên. Nguồn: [Wikipedia](#) 

	2	3	4	5	6	7	8	9	10	(số nguyên tố)
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	
51	52	53	54	55	56	57	58	59	60	
61	62	63	64	65	66	67	68	69	70	
71	72	73	74	75	76	77	78	79	80	
81	82	83	84	85	86	87	88	89	90	
91	92	93	94	95	96	97	98	99	100	
101	102	103	104	105	106	107	108	109	110	
111	112	113	114	115	116	117	118	119	120	

Ứng dụng

Phân tích thừa số nguyên tố

Phân tích số nguyên nhỏ

Khi phân tích các số nhỏ $i \approx 10^6$, thay vì lưu kết quả kiểm tra tính nguyên tố của i ở mảng `is_prime[i]`, ta có thể sử dụng `min_prime[i]` lưu ước nguyên tố nhỏ nhất của số i .

```

1  const int maxn = 1000000 + 5; //10^6 + 5
2  int min_prime[maxn];
3  void sieve(int n){
4      for (int i = 2; i * i <= n; ++i) {
5          if (min_prime[i] == 0) { //nếu i là số nguyên tố
6              for (int j = i * i; j <= n; j += i) {
7                  if (min_prime[j] == 0) {
8                      min_prime[j] = i;
9                  }
10             }
11         }
12     }
13     for (int i = 2; i <= n; ++i) {
14         if (min_prime[i] == 0) {
15             min_prime[i] = i;
16         }
17     }
18 }

```

Bây giờ ta có thể phân tích một số ra thừa số nguyên tố:

```

1 | vector<int> factorize(int n) {
2 |     vector<int> res;
3 |     while (n != 1) {
4 |         res.push_back(minPrime[n]);
5 |         n /= minPrime[n];
6 |     }
7 |     return res;
8 | }

```

Mỗi lần ta chia n cho ước nguyên tố nhỏ nhất $\text{min_prime}[n]$ đến khi nào n giảm về 1. Trong trường hợp xấu nhất thì mỗi lần chia $\text{min_prime}[n]$ đều bằng 2. Vì vậy, hàm phân tích trên **độ phức tạp thời gian** trường hợp xấu nhất là $O(\log n)$.

Tuy nhiên, phương pháp này có **độ phức tạp không gian** $O(n)$ và thường sử dụng trong trường hợp cần phân tích nhiều số nguyên ra thừa số nguyên tố.

Đặc biệt, khi phân tích tất cả các số nguyên từ 1 đến n , tổng độ phức tạp chỉ còn lại $O(n \log \log n)$.

► Chứng minh tổng độ phức tạp khi phân tích tất cả các số nguyên từ 1 đến n

Phân tích số nguyên lớn hơn



Nhận xét: Nếu tất cả các số nguyên trong đoạn $[2; \sqrt{n}]$ đều không phải là ước của n thì n là số nguyên tố.

Dựa vào **Nhận xét** trên, để phân tích một số nguyên n lớn (khoảng 10^9 hay 10^{12}), ta xây dựng được thuật toán với độ phức tạp $O(\sqrt{n})$ dưới đây:

```

1 | vector<long long> factorize(long long n) {
2 |     vector<long long> res;
3 |     for (long long i = 2; i * i <= n; i++){
4 |         while (n % i == 0){
5 |             res.push_back(i);
6 |             n /= i;
7 |         }
8 |     }
9 |     if (n > 1) res.push_back(n);
10 |    return res;
11 | }

```

Đến đây, ta dễ dàng tìm được một cách cải tiến thuật toán này: ta chỉ cần xét các số nguyên tố trong đoạn $[2; \sqrt{n}]$. Thật vậy, nếu n không chia hết cho số nguyên tố p thì chắc chắn n sẽ không chia hết cho bội của p .

Trước hết, ta tạo mảng các số nguyên tố trong đoạn $[2; \sqrt{n}]$. Sau đó, chúng ta làm như sau:

```

1  vector<int> primes;
2
3  vector<long long> factorize(long long n) {
4      vector<long long> res;
5      for (int p : primes) {
6          if (1LL * p * p > n) break;
7          while (n % p == 0) {
8              res.push_back(p);
9              n /= p;
10         }
11     }
12     if (n > 1) res.push_back(n);
13     return res;
14 }
```

Phân tích sẽ mất độ phức tạp cho trường hợp xấu nhất là $O(\pi(\sqrt{n})) \sim O\left(\frac{\sqrt{n}}{\log \log n}\right)$.

Trong đó $\pi(x)$ là số số nguyên tố trong đoạn $[2; x]$. Bạn đọc tham khảo thêm hàm này ở phần **Mở rộng** của bài viết.

Tìm số nguyên tố trong đoạn $[L; R]$

Đôi khi bạn phải tìm tất cả các số không phải trên đoạn $[1; N]$ mà là trên đoạn $[L; R]$ có kích thước nhỏ nhưng R lớn.

Ví dụ như $R - L + 1 \approx 1e6$ và $R \approx 10^{12}$.

Ta đặt $N = R - L + 1$ là độ dài đoạn $[L; R]$ để tiện theo dõi.

Để làm được điều này, ta sẽ chuẩn bị trước một mảng gồm các số nguyên tố trong đoạn $[1; \sqrt{R}]$. Sau đó, dùng các số nguyên tố đó để đánh dấu trong đoạn $[L; R]$.

```

vector<bool> sieve(long long L, long long R) {
    long long sqrtR = sqrt(R);
    vector<bool> mark(sqrtR + 1, false);
    vector<long long> primes;
    // sinh ra các số nguyên tố <= sqrt(R)
    for (long long i = 2; i <= sqrtR; ++i) {
        if (!mark[i]) {
            primes.push_back(i);

```

```

9         for (long long j = i * i; j <= sqrtR; j += i)
10             mark[j] = true;
11     }
12 }
13
14 vector<bool> isPrime(R - L + 1, true);
15 for (long long i : primes)
16     for (long long j = max(i * i, (L + i - 1) / i * i); j <
17         isPrime[j - L] = false;
18 if (L == 1)
19     isPrime[0] = false;
20 return isPrime;
21 }
```

Độ phức tạp thời gian: $O\left(N \log \log(R) + \sqrt{R} \log \log \sqrt{R}\right)$

Độ phức tạp không gian: $O\left(N + \sqrt{R}\right)$

Trong đó:

- Tìm các số nguyên tố trong đoạn $[1; \sqrt{R}]$ mất $O\left(\sqrt{R} \log \log \sqrt{R}\right)$.
- Dùng các số nguyên tố đó để đánh dấu trong đoạn $[L; R]$ mất $O\left(N \log \log R\right)$.

Ta cũng không cần phải sinh trước các số nguyên tố trong đoạn $[1; \sqrt{R}]$:

```

vector<bool> is_prime;
void sieve(int L, int R){
    is_prime.assign(R - L + 1, true);
    // x là số nguyên tố khi và chỉ khi is_prime[x - L] == true

    for (long long i = 2; i * i <= R; ++i) {
        // Lưu ý: (L + i - 1) / i * i là bội nhỏ nhất của i mà >= L
        for (long long j = max(i * i, (L + i - 1) / i * i); j <
            is_prime[j - L] = false;
        }
    }

    if (1 >= L) { // Xét riêng trường hợp số 1
        is_prime[1 - L] = false;
    }

    for (long long x = L; x <= R; ++x) {
        if (is_prime[x - L]) {
            // x là số nguyên tố
        }
    }
}
```

```

    }
}

```

Độ phức tạp thời gian sẽ tệ hơn : $O(N \log(R) + \sqrt{R})$.

Tuy nhiên, ta lại được lợi thế hơn về độ phức tạp không gian: $O(N)$.

Nguyên nhân là ta dùng tất cả các số nguyên trong đoạn $[2; \sqrt{R}]$ đó để đánh dấu trong đoạn $[L; R]$ nên sẽ mất

$$O\left((R - L + 1) \cdot \left(\frac{1}{2} + \frac{1}{3} + \frac{1}{4} + \dots + \frac{1}{\lfloor \sqrt{R} \rfloor}\right)\right) = O(N \log(R)).$$

Một số ví dụ

VNOI - Phi hàm Euler [✎](#)

Tóm tắt đề:

Cho số nguyên dương T và T số nguyên dương n_i . Hãy tính phi hàm $\varphi(n_i)$ của T số nguyên dương đã cho.

$$\varphi(n) = p_1^{\alpha_1-1} p_2^{\alpha_2-1} \dots p_k^{\alpha_k-1} (p_1 - 1)(p_2 - 1) \dots (p_k - 1)$$

► [Gợi ý](#)

► [Lời giải](#)

Một số cải tiến của sàng nguyên tố Eratosthenes

“

Sàng nguyên tố Eratosthenes với ĐPT thời gian $O(n \log \log n)$ đã khá phù hợp với hầu hết các bài toán lập trình thi đấu. Tuy nhiên điểm yếu chí mạng của nó chính là ĐPT không gian $O(n)$.

Một số cải tiến dưới đây có thể không phù hợp với những bạn mới chỉ biết đến sàng nguyên tố. Các bạn hãy luyện tập với các bài tập luyện tập trước khi đến với các cải tiến bên dưới nha!

So sánh mảng bool và vector

- Một biến `bool` chỉ có hai giá trị `true/false` nên về mặt lý thuyết chỉ cần 1 bit để lưu trữ nó. Nhưng bình thường, các máy tính hiện nay khi lưu trữ biến `bool` sẽ sử dụng 1 byte (tương đương với 8 bits) để truy cập nhanh chóng. Vì thế một mảng `bool a[n]` sẽ cần đến n bytes.
- `vector<bool>` được tối ưu để lưu trữ 1 biến `bool` trong 1 bit thay vì 1 byte, ngoài ra còn có 40 bytes sử dụng cho khởi tạo `vector<bool>` ban đầu. Tuy nhiên, việc tối ưu về bộ nhớ khiến ta phải truy cập bit một cách gián tiếp: mỗi lần truy cập, đọc, ghi bit ta cần tách nhỏ từng bit của byte đó. Trong trường hợp bộ dữ liệu nhỏ (khoảng 10^6), truy cập như vậy sẽ chậm hơn so với việc truy cập trực tiếp.

- Tóm lại, ta có bảng dưới đây

	bool a[n]	vector
Không gian lưu trữ	n bytes	$40 + \left\lceil \frac{n}{8} \right\rceil$ bytes
Truy cập, đọc, ghi bit	Trực tiếp	Gián tiếp

Chỉ sàng số lẻ

Một cải tiến khác cũng có thể được sử dụng đó là chỉ tiến hành kiểm tra với số lẻ (số chẵn chỉ có 2 là số nguyên tố).

Điều này có thể giảm cả không gian để lưu trữ lẫn số bước tính toán đi một nửa.

Code C++ minh họa

```

1  vector<bool> is_prime;
2  void sieve_odd(int n){
3      is_prime.assign(n / 2 + 1, true);
4      //is_prime[t] = true nghĩa là 2*t+1 là số nguyên tố
5      is_prime[0] = false;
6      for (int t = 1; t * t <= n / 4; t++) {
7          int i = 2 * t + 1;
8          if (is_prime[t]){
9              for (int j = i * i; j <= n; j += i * 2)
10                 is_prime[j / 2] = false;
11          }
12      }
13  }
```

Độ phức tạp thời gian: $O\left(\frac{n}{2} \cdot \log \log n\right)$

Độ phức tạp không gian: $O\left(\frac{n}{2}\right)$

Sử dụng bitset

Trong C++, `std::bitset` là một công cụ hữu hiệu trong việc lưu trữ và xử lý dãy nhị phân.

`std::bitset` sử dụng cách lưu bit tương tự `std::vector<bool>` và nhanh hơn `std::vector<bool>` một chút. Tuy nhiên kích thước `MAX` của `std::bitset<MAX>` phải được biết lúc biên dịch.

Code C++ minh họa


```

1  | const int maxn = 1e6;
2  | bitset<maxn + 1> is_prime;
3  | void sieve_bitset(int n){
4  |     is_prime.set(); // gán tất cả các bit là true
5  |     is_prime[0] = is_prime[1] = 0;
6  |     for (int i = 2; i*i <= n; i = is_prime._Find_next(i)) {
7  |         if (is_prime[i]) {
8  |             for (int j = i*i; j <= n; j += i) {
9  |                 is_prime[j] = 0;
10 |             }
11 |         }
12 |     }
13 | }

```

Độ phức tạp thời gian: $O(n \log \log n)$

Độ phức tạp không gian: $O\left(\frac{n}{32}\right)$

Một cách khác, vì biến `bool` lưu trong bộ nhớ thường là 1 byte (8 bits), tuy nhiên thực chất chỉ cần sử dụng 1 bit. Vì thế ta có thể sử dụng một biến `int` để lưu nhiều biến `bool`. Để code được nhanh chóng, ở đây ta nên sử dụng các [phép toán trên bit](#) .

Code C++ minh họa

```

1  | #define doc(n) (prime_bits[n >> 3] & (1 << (n & 7)))
2  | #define set(n) {prime_bits[n >> 3] |= (1 << (n & 7));}
3  | vector<int> prime_bits;
4  | void sieve_bits(int n){
5  |     prime_bits.assign((n >> 3) + 5, 0);
6  |     set(0); set(1);
7  |     for(int i = 2; i * i <= n; i++){
8  |         if (!doc(i)){
9  |             for(int j = i * i; j <= n; j += i){
10 |                 set(j);
11 |             }
12 |         }
13 |     }
14 | }

```

Độ phức tạp thời gian: $O(n \log \log n)$

Độ phức tạp không gian: $O\left(\frac{n}{8}\right)$

Trong code bên trên, `int` được sử dụng để lưu 8 giá trị `bool`.

Trên thực tế, `int/unsigned int` chứa 4 bytes hay 32 bits. Nhờ đó, một số

`int/unsigned int` có thể lưu trữ đến 32 giá trị `bool`. Và bạn đọc có thể thử cách lưu 32 giá trị thay vì 8 vào code bên trên.

Sàng nguyên tố tuyến tính - Linear Sieve

“

- Sàng nguyên tố này được cải tiến từ Sàng Eratosthenes. Tuy có ĐPT thời gian là $O(n)$ nhưng với những bộ dữ liệu khoảng 10^6 thì không nhanh hơn Sàng Eratosthenes là mấy.
- Sàng $O(n)$ này có lưu lại các ước nguyên tố nhỏ nhất của các số không vượt quá n nên sẽ phù hợp cho các bài toán liên quan đến phân tích thừa số nguyên tố.

Hướng tiếp cận

Xét `min_prime[i]` là ước nguyên tố nhỏ nhất của i
 Mảng `primes[]` sẽ lưu tất cả các số nguyên tố đã tìm được.
 Duyệt các số từ 2 đến n . Ta có 2 trường hợp:

- `min_prime[i] = 0` $\Rightarrow i$ là số nguyên tố. Vì thế, ta gán `min_prime[i] = i` và thêm i vào cuối mảng `primes[]`.
- Ngược lại, `min_prime[i] \neq 0`, ta phải tính được `min_prime[i]` trong các vòng lặp trước đó.

Trong cả hai trường hợp, ta đều cần cập nhật giá trị của `min_prime[]` cho các bội của i . Và mục tiêu của ta là gán giá trị `min_prime[]` tối đa một lần cho mỗi số.

Chúng ta có thể làm như sau: Duyệt các số nguyên i từ 2 đến n . Với mỗi số nguyên i , ta sẽ gán `min_prime[i * p_j] = p_j` với p_j là các số nguyên tố $\leq \text{min_prime}[i]$.

Code C++ minh họa

```

1  vector<int> min_prime, primes;
2  void linear_sieve(int n){
3      min_prime.assign(n + 1, 0);
4
5      for (int i = 2; i <= n; ++i) {
6          if (min_prime[i] == 0) {
7              min_prime[i] = i;
8              primes.push_back(i);
9          }
10         for (int j = 0; i * primes[j] <= n; ++j) {
11             min_prime[i * primes[j]] = primes[j];
12             if (primes[j] == min_prime[i]) {
13                 break;
14             }
15         }
16     }
17 }
```

```
    }
}
```

Độ phức tạp thời gian: $O(n)$

Độ phức tạp không gian: $O(n)$

► Giải thích về ĐPT của thuật toán

Sàng phân đoạn - Block Sieve / Segmented Sieve

“

Đây là một trong số những phương pháp hữu hiệu khắc phục điểm yếu về không gian của sàng nguyên tố Eratosthenes.

Xét code sàng Erathosenes sau:

```
1  for (int i = 2; i * i <= n; i++) {
2      if (is_prime[i]) {
3          // j sẽ bắt đầu chạy từ i * i
4          for (int j = i * i; j <= n; j += i)
5              is_prime[j] = false;
6      }
7  }
```

Vì vòng lặp `j` bắt đầu từ `i * i` nên ta không cần phải giữ lại toàn bộ mảng `is_prime[1..n]` trong suốt quá trình sàng. Khi đó:

- Giữ lại các số nguyên tố p trong đoạn $[1; \sqrt{n}]$: `prime[1..sqrt(n)]`
- Chia $[1; n]$ thành các đoạn con (block) và sàng riêng từng đoạn (block).

Gọi S là kích thước của mỗi đoạn. Như thế, chúng ta sẽ có $\left\lceil \frac{n}{S} \right\rceil$ đoạn. Đoạn thứ k ($k = 0 \dots \left\lceil \frac{n}{S} \right\rceil - 1$) là $[kS; \min(kS + S - 1, n)]$.

Với mỗi đoạn, vòng lặp `for (int j = i * i; j <= n; j += i)` sẽ thay đổi sao cho `j` chỉ chạy trong đoạn đang xét.

Code C++ minh họa

```
vector<int> primes;
void segmented_sieve(int n) {
    const int S = 10000;

    int nsqrt = sqrt(n);
    vector<char> is_prime(nsqrt + 2, true);
```

```

8     for (int i = 2; i <= nsqrt; i++) {
9         if (is_prime[i]) {
10            primes.push_back(i);
11            for (int j = i * i; j <= nsqrt; j += i)
12                is_prime[j] = false;
13        }
14    }
15
16    int result = 0;
17    vector<char> block(S);
18    for (int k = 0; k * S <= n; k++) {
19        fill(block.begin(), block.end(), true);
20        int start = k * S;
21        for (int p : primes) {
22            // start_idx * p là bội nhỏ nhất của p mà start_idx
23            int start_idx = (start + p - 1) / p;
24            int j = max(start_idx, p) * p - start;
25            for (; j < S; j += p)
26                block[j] = false;
27        }
28        if (k == 0)
29            block[0] = block[1] = false;
30        for (int i = 0; i < S && start + i <= n; i++) {
31            if (block[i])
32                result++;
33        }
34    }
35    cout << result << '\n'; // In ra số số nguyên tố tìm được
}

```

Độ phức tạp thời gian: $O\left(n \log \log n + \frac{n \cdot \pi(\sqrt{n})}{S}\right)$

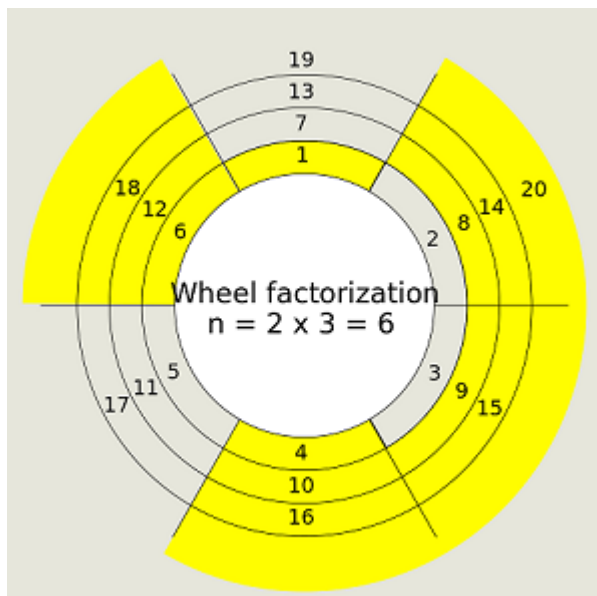
Độ phức tạp không gian: $O(\sqrt{n} + S)$

Chú ý rằng ta phải chọn S sao cho cân bằng giữa độ phức tạp không gian và thời gian.



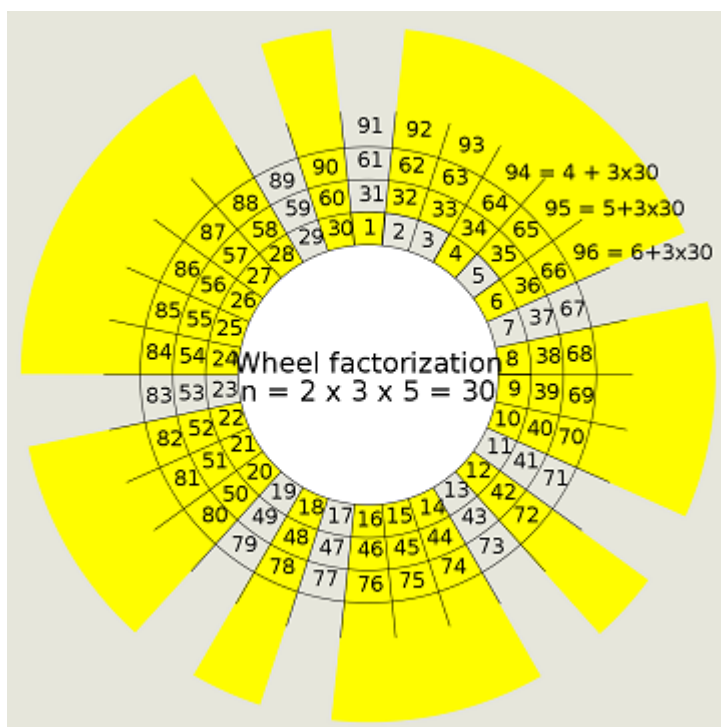
Bánh xe phân tích - Wheel Factorization

Wheel Factorization là phương pháp cải tiến có thể loại bỏ đi rất nhiều trường hợp trước khi sàng nguyên tố. Thay vì chỉ xét các số lẻ, ta có thể loại bỏ các số là bội của 2, 3, 5, 7, ... Việc này có thể giúp chúng ta giảm đi ĐPT cả thời gian lẫn không gian đi một chút.



Ví dụ khi chọn $n = 2 \cdot 3 = 6$ sẽ chỉ loại đi các số là bội của 2 hoặc 3.

Bản chất của bánh xe khi này là chỉ giữ lại các số 2, 3 và các số có dạng $6k + 1$ hoặc $6k + 5$.



Hoặc ví dụ khi chọn loại bỏ các bội của 2 hoặc 3 hoặc 5, ta chọn $n = 2 \cdot 3 \cdot 5 = 30$

Bản chất là ta chỉ giữ lại các số 2, 3, 5 và các số có dạng $30k + i$ với $i < 30$ và i không chia hết cho 2, 3, 5.

($i \in \{1, 7, 11, 13, 17, 19, 23, 29\}$)

Trong trường hợp này, ta chỉ cần sử dụng mảng kiểm tra nguyên tố `is_prime` cho các số có dạng trên.

Lý do người ta dùng bánh xe thì bạn đọc có thể xem ảnh dưới đây. Nguồn: [Wikipedia](#)



Code C++ minh họa

```
1 // Các thông số của bánh xe
2 // Bội của các số nguyên tố bé
3 const int wheel_size = 2 * 3 * 5;
4 const int num_offsets = 8;
5 // Tập các số dư
6 const int wheel_offsets[] = {1, 7, 11, 13, 17, 19, 23, 29};
7 // Thứ tự của 1 số trong offsets
8 int num_in_offsets[wheel_size];
9
10 vector<bool> is_prime;
11
12 // vị trí trong mảng is_prime
13 int pos(const int &v){
14     return v / wheel_size * num_offsets + num_in_offsets[v % wh
15 }
16
17 void sieve_with_wheel(int n){
18     for (int i = 0; i < num_offsets; i++)
19         num_in_offsets[wheel_offsets[i]] = i + 1;
20
21     is_prime.assign(pos(n) + num_offsets + 10, true);
22     is_prime[0] = false; // 1 không là số nguyên tố
23
24     // Sàng
25     for (int i = 0; i * i <= n; i += wheel_size) {
26         for (int j = 0; j < num_offsets; ++j) {
27             int u = i + wheel_offsets[j];
28             if (is_prime[pos(u)]) {
29                 for (int v = u * u; v <= n; v += u) {
30                     if (num_in_offsets[v % wheel_size]) {
31                         is_prime[pos(v)] = false;
32                     }
33                 }
34             }
35         }
36     }
37
38     // vòng lặp sau sẽ duyệt tất cả các số nguyên tố lớn hơn 5
39     for (int i = 0; i <= n; i += wheel_size) {
40         for (int j = 0; j < num_offsets; ++j) {
41             int u = i + wheel_offsets[j];
42             if (u <= n && is_prime[pos(u)]) {
43                 // u là một số nguyên tố
44             }
45         }
46     }
47 }
```

Độ phức tạp thời gian: $O\left(\frac{4}{15}n \log \log n\right)$


Độ phức tạp không gian: $O\left(\frac{4}{15}n\right)$

Xét kích thước "bánh xe" là $mod = 2 \cdot 3 \cdot 5 \dots$ có thể chọn mod vào khoảng \sqrt{n} thì ĐPT sẽ còn là $O\left(\frac{n}{\log \log n}\right)$. Nhìn thì ĐPT thấp hơn sàng Eratosthenes thông thường, nhưng vì phương pháp trên mỗi vòng lặp đều sử dụng phép nhân/chia nên thời gian chạy có thể chậm hơn nhiều so với sàng Eratosthenes thông thường với bộ dữ liệu nhỏ ($n \leq 10^6$).

Và vì lý do bộ nhớ cache mà người ta chỉ thường chọn modulo $mod \in 30; 210$. Các số lọc được tiếp tục kiểm tra bằng cách khác như bên trên.

Kết hợp các cải tiến


Bên trên là một số cách cải tiến thường được sử dụng. Tuy nhiên bạn có thể kết hợp các cải tiến một cách hợp lý để tạo ra một sàng nguyên tố mạnh mẽ.

Dưới đây là một số sàng được sưu tầm bởi [Code cùng RR](#) .

► Sàng phân đoạn và Chỉ sàng số lẻ

► Sàng phân đoạn và Bánh xe phân tích

► Cải tiến của Kim Walisch

So sánh độ dài code và thời gian chạy với $n = 10^9$ của một số sàng nguyên tố (Nguồn: [Code cùng RR](#) )


```

23 +-- 20 lines: Naive sieve implementation from cp-algorithms .....
43 +-- 21 lines: Naive sieve with bitset .....
64 +-- 24 lines: Linear sieve from cp-algorithms .....
88 +-- 31 lines: odd sieve with bitset .....
119 +-- 38 lines: Block sieve from cp-algorithms .....
157 +-- 72 lines: Kim Walisch .....
229 +-- 33 lines: Block sieve, only odd elements .....
262 +--104 lines: Segmented sieve with wheel factorization .....
366
367 +-- 28 lines: O(N) sieve to calculate sum of primes .....
395 +-- 42 lines: O(N) sieve to calculate sum of primes, optimized hash table .....
437
438 int main() {
439     ios::sync_with_stdio(0); cin.tie(0);
440     benchmark(sieve_cp_algorithms, "cp_algorithms_naive");
441     benchmark(sieve_bitset, "sieve_bitset");
442     benchmark(linear_sieve_cp_algorithms, "cp_algorithms_linear");
443     benchmark(odd_sieve_bitset::sieve, "odd_sieve_bitset");
444     benchmark(block_sieve_cp_algorithms, "cp_algorithms_block");
445     benchmark(kim_walisch::sieve, "kim_walisch_sieve");
446     benchmark(block_sieve_odd, "block_sieve_odd");
447     benchmark(segmented_sieve_wheel::sieve, "segmented_sieve_wheel");
448
449     cout << "==== O(N) SIEVE FOR CALCULATING SUM OF PRIMES =====> endl;
450     benchmark(sieve_sum, "sieve_sum_O(n)");
451     benchmark(sieve_sum_optimized_hash_table, "sieve_sum_optimized");
452     return 0;
453 }

```

N benchmark.cpp
:e benchmark.cpp




```

~/code/CF ./benchmark
cp_algorithms_naive      : 8364.87ms
sieve_bitset             : 8105.95ms
cp_algorithms_linear     : 5609.58ms
odd_sieve_bitset         : 4563.56ms
cp_algorithms_block      : 2520.91ms
kim_walisch_sieve       : 2435.91ms
block_sieve_odd          : 1298.88ms
segmented_sieve_wheel    : 557.632ms
==== O(N) SIEVE FOR CALCULATING SUM OF PRIMES =====>
sieve_sum_O(n)          : 128.037ms
sieve_sum_optimized      : 83.047ms

```


Một số sàng nguyên tố khác

Ngoài Sàng Eratosthenes, còn có một số sàng nguyên tố khác như:

- [Sàng nguyên tố Atkin](#)  với ĐPT $O(n)$
- [Sàng nguyên tố Sundaram](#)  với ĐPT $O(n \log n)$
- [Sàng Pritchard](#)  với ĐPT $O\left(\frac{n}{\log \log n}\right)$
- ...

Tuy nhiên, khi gặp các bộ dữ liệu n vào khoảng 10^6 thì các sàng này hầu như chạy chậm hơn so với Sàng Eratosthenes thông thường.

Mở rộng

- Sử dụng cách sàng như sàng nguyên tố chúng ta có thể xây dựng các sàng cho các số có tính chất đặc biệt khác, ví dụ như lưu ước chính phương lớn nhất, thay các số nguyên tố thành các số phân biệt có tính chất nào đó ... Ví dụ như [VNOI - Duyên Hải 2020 - Lớp 10 - Bài 2](#) 
- **Hàm $\pi(x)$** là hàm đếm số số nguyên tố không vượt quá số dương x .
Và theo định lý số nguyên tố (Prime Number Theorem), ta có một công thức để ước lượng:

$$\pi(x) \sim \frac{x}{\ln x}$$

Công thức này có thể hữu ích trong việc ước chừng các độ phức tạp liên quan đến số nguyên tố.

- ▶ Khi tính số lượng các số nguyên tố hay tổng các số nguyên tố không vượt quá n , việc sử dụng Sàng nguyên tố là một phương pháp nhanh dễ hiểu. Tuy nhiên, với những bộ dữ liệu lớn, người ta thường sử dụng sử dụng [thuật toán Meissel–Lehmer](#) ^[↗] hay [thuật toán Lucy Hedgehog](#) ^[↗], có thể chạy với n lên đến 10^{12} trong 1s. Xem code bằng C++ tại [thuật toán Lehmer - đếm số lượng số nguyên tố](#) ^[↗] và [thuật toán Lucy Hedgehog - tính tổng các số nguyên tố](#) ^[↗].
- ▶ Để phân tích thừa số nguyên tố thì có hai sàng tốt hơn sàng nguyên tố: [Quadratic Sieve](#) ^[↗] và [General number field sieve](#) ^[↗]
- ▶ Ngoài ra có thể tham khảo thêm các code sàng nguyên tố được sưu tầm bởi [Code cùng RR](#) ^[↗] tại [đây](#). ^[↗]







Bài tập luyện tập

Sàng nguyên tố cơ bản

- ▶ [VNOI - Free Contest 75 - FPRIME](#) ^[↗]
- ▶ [VNOI - Free Contest 102 - PRIME](#) ^[↗]
- ▶ [VNOI - Bedao Grand Contest 01 - KPRIME](#) ^[↗]
- ▶ [VNOI - Bedao Regular Contest 03 - PRIME](#) ^[↗]
- ▶ [VNOI - Tìm số nguyên tố](#) ^[↗]
- ▶ [VNOI - Số nguyên tố!](#) ^[↗]
- ▶ [VNOI - Vòng số nguyên tố](#) ^[↗]
- ▶ [VNOI - Bedao Regular Contest 02 - PXOR](#) ^[↗]
- ▶ [VNOI - VM 08 Bài 04 - Xóa số](#) ^[↗]
- ▶ [VNOI - Euler](#) ^[↗]
- ▶ [NTUCoder - Phi Euler](#) ^[↗]
- ▶ [NTUCoder - Phi Euler 2](#) ^[↗]
- ▶ [VNOI - COCI 2016/2017 - Contest 6 - Savrsen](#) ^[↗]
- ▶ [NTUCoder - Sum gcd](#) ^[↗]
- ▶ [VNOI - Sum of Primes](#) ^[↗]
- ▶ [Codechef - Chef and Divisor Tree](#) ^[↗]
- ▶ [SPOJ - A conjecture of Paul Erdős](#) ^[↗]
- ▶ [SPOJ - Primal Fear](#) ^[↗]
- ▶ [SPOJ - Primes Triangle](#) ^[↗]
- ▶ [Codeforces - Almost Prime](#) ^[↗]






- [Codeforces - Sherlock And His Girlfriend](#) 
- [SPOJ - Namit in Trouble](#) 
- [SPOJ - Bazinga!](#) 
- [Hackerrank - Project Euler - Prime pair connection](#) 
- [SPOJ - N-Factorful](#) 
- [SPOJ - Binary Sequence of Prime Numbers](#) 
- [UVA 11353 - A Different Kind of Sorting](#) 
- [SPOJ - Prime Generator](#) 
- [Codeforces - Nodbach Problem](#) 
- [Codefoces - Colliders](#) 

Sàng nguyên tố cải tiến. Thuật toán Meissel-Lehmer

- [VNOI - Prime Number Theorem](#) 
- [SPOJ - Primes2](#) 
- [SPOJ - KPrimes2](#) 
- [SPOJ - BSPRIME](#) 
- [ICPC 2022 vòng Quốc gia - C: Consecutive Primes](#) 
- [VOI 22 Bài 6 - Xây dựng ma trận](#) 

Nguồn tham khảo

Bài viết được tổng hợp từ các nguồn dưới đây:

- CP - Algorithms:
 - [Sieve of Eratosthenes](#) 
 - [Linear Sieve](#) 
- [Bài viết của Code cùng RR](#) 
- [Bài viết 2 của Code cùng RR](#) 
- [Wikipedia](#) 

Được cung cấp bởi [Wiki.js](#)