Teoria

Qzxell

February 22, 2024

1 Extensiones del Criba de Eratóstenes

Por Hikari9, historia, hace 8 años, en inglés

Ahora que sabemos cómo cribar adecuadamente, veamos algunos trucos para la criba de Eratóstenes para obtener algunas otras cribas interesantes.

Presentando un problema

Digamos que queremos contar el número de divisores de un número. Una forma es verificar todos los números hasta $\sqrt{n}$ y comprobar si n divide a ese número. Otra forma es obtener su factorización prima y obtener el producto de $(exponente + 1)$ a través de combinatoria. Cualquiera de los métodos es $O(\sqrt{n})$ en promedio, por lo tanto, $O(n\sqrt{n})$ si se hace para todos los números hasta n.

Pero, ¿qué pasa si un problema nos pide imprimir el número de divisores de todos los números del 1 al 10^7 en menos de 3 segundos? ¡Un algoritmo $O(n\sqrt{n})$ será demasiado lento! Cuando intenté contar los divisores usando el método de la raíz cuadrada, se ejecutó durante aproximadamente 61 segundos en mi computadora. Eso definitivamente no se ejecutará a tiempo.

Afortunadamente, podemos ajustar la criba de Eratóstenes para contar el número de divisores de manera más eficiente y elegante. Y verás que esta técnica funciona no solo para el número de divisores, sino también para generar la suma de divisores, la función totient, el mayor divisor primo, básicamente todas las funciones que tienen que ver con divisores.

Criba de Divisores $O(n \log n)$

```
int divisors[n + 1];
for (int i = 1; i <= n; ++i)
 for (int j = i; j <= n; j += i)
  ++divisors[j];
```

Entonces, ¿qué pasa con este código bastante corto? Este código corto genera el número de divisores de todos los números menores o iguales a n. ¡Oh wow, acabamos de

resolver nuestro problema! Pero espera, ¿no estamos demasiado apurados? ¿Este código incluso funciona? ¿Y cómo funciona en primer lugar? ¿Cómo llegamos a $O(n \log n)$? No te preocupes, responderemos esas preguntas una por una.

Corrección

Queremos contar el número de divisores de un número. Desde otra perspectiva, en lugar de eso, podemos empezar desde el divisor e incrementar la cuenta de todos sus múltiplos. Haciendo eso para todos los divisores, ahora tenemos todos los recuentos de divisores de todos los números hasta n. Hurray.

Complejidad

Ahora que hemos demostrado que el algoritmo es correcto, ¿cómo estamos seguros de que la complejidad es $O(n \log n)$ cuando parece $O(n^2)$? La respuesta es porque estamos sumando una serie armónica. El bucle interno se ejecuta $\lfloor n/i \rfloor$ iteraciones, por lo tanto, el número total de iteraciones es:

$$n\left(1 + \frac{1}{2} + \frac{1}{3} + \ldots + \frac{1}{n}\right)$$

Si conoces cálculo, el recuento simplemente se aproxima a la suma de Riemann de la serie armónica, que podemos integrar para obtener $n \log n$. Magia matemática en su máxima expresión.

En general, $O(n \log n)$ para $n = 10^7$ se ejecuta aproximadamente en 1.700 segundos en una computadora normal, y aún más rápido en jueces en línea que hacen computación en la nube. Esto ya resuelve nuestro problema, ¡y vaya, con un código realmente corto!

Criba de Suma de Divisores $O(n \log n)$

```
int sumdiv[n + 1];
for (int i = 1; i <= n; ++i)
 for (int j = i; j <= n; j += i)
  sumdiv[j] += i;
```

También podemos usar esta técnica para obtener la suma de divisores. Solo incrementa por el divisor en lugar de solo incrementar por 1.

Criba de Euler Totient $O(n \log \log n)$

```
int totient[n + 1];
for (int i = 1; i <= n; ++i) totient[i] = i;
for (int i = 2; i <= n; ++i)
 if (totient[i] == i)
  for (int j = i; j <= n; j += i)
   totient[j] -= totient[j] / i;
```

Esta técnica también podría generar la función totient de Euler, donde `totient[a]` es el número de enteros positivos menores que a que son primos relativos a a. Es $O(n \log \log n)$ porque realiza el bucle interno solo si el número es primo (ver serie armónica de primos).

Podrías preguntarte por qué es `totient[j] -= totient[j] / i`. Esto se debe a la naturaleza de la función totient de Euler, que necesita un poco de antecedentes en teoría de números para demostrarlo. Esta maravillosa entrada de blog de PraveenDhinwa proporciona una buena explicación si deseas la prueba extensa.

Criba del Mayor Divisor Primo $O(n \log \log n)$

```
int big[n + 1] = {1, 1};
for (int i = 1; i <= n; ++i)
 if (big[i] == 1)
  for (int j = i; j <= n; j += i)
   big[j] = i;
```

Podemos tabular el mayor divisor primo por número. Esto es útil para la verificación rápida de primos (cuando `big[p] == p`) y para la factorización prima más fácil. Ya no necesitas iterar a través de todos los primos para factorizar primos, solo necesitas un solo bucle while, algo así como `while (n > 1) { factores.push_back(big[n]); n /= big[n]; }`.

Hay muchas más extensiones de la maravillosa criba de Eratóstenes. Si conoces algunas interesantes, házmelo saber también para que pueda agregarlas (y darte crédito) en este blog.

Descargo de responsabilidad: Fui yo quien le puso nombre a las "cribas", así que no son nombres oficiales ni nada por el estilo. Además, técnicamente ya no son cribas (las cribas son filtros, pero las funciones anteriores son generadores de números), pero pensé que la etiqueta de criba era genial, así que la puse, si no te importa.

Aplicaciones de la Criba Lineal en Programación Competitiva

Este breve artículo se centra principalmente en la criba lineal y sus diversas aplicaciones en programación competitiva, incluyendo una breve introducción sobre cómo seleccionar números primos y una forma de calcular valores múltiples de funciones multiplicativas.

Criba de Eratóstenes

Aunque este nombre pueda sonar aterrador, la criba de Eratóstenes es probablemente la forma más simple de seleccionar todos los números primos en un rango dado de 1 a n. Como ya sabemos, una de las propiedades que tienen todos los primos es que no tienen factores excepto 1 y ellos mismos. Por lo tanto, si tachamos todos los compuestos, que tienen al menos un factor, podemos obtener todos los primos. El siguiente código muestra una implementación simple de dicho algoritmo:

Listing 1: Criba de Eratóstenes

```
#include <vector>
#include <algorithm>

std::vector<int> prime;
bool is_composite[MAXN];

void sieve(int n) {
    std::fill(is_composite, is_composite + n, false);
    for (int i = 2; i < n; ++i) {
        if (!is_composite[i]) prime.push_back(i);
        for (int j = 2; i * j < n; ++j)
            is_composite[i * j] = true;
    }
}
```

Como podemos ver, la instrucción `is_composite[i * j] = true;` tacha todos los números que tienen un factor, ya que todos son compuestos. Todos los números restantes, por lo tanto, deben ser primos. Luego verificamos esos primos y los colocamos en un contenedor llamado `prime`.

Criba Lineal

Se puede analizar que el método anterior se ejecuta en complejidad $O(n \log \log n)$. Tomémonos un minuto para considerar el cuello de botella de dicha criba. Aunque necesitamos tachar cada compuesto una vez, en la práctica ejecutamos el bucle interno para un compuesto múltiples veces debido al hecho de que tiene múltiples factores. Por lo tanto, si podemos establecer una representación única para cada compuesto y seleccionarlos solo una vez, nuestro algoritmo será algo mejor. De hecho, es posible hacerlo. Observa que cada compuesto q debe tener al menos un factor primo, así que podemos elegir el menor factor primo p y dejar que el resto sea i, es decir, $q = ip$. Dado que p es el menor factor primo, tenemos $i \geq p$, y ningún primo menor que p puede dividir i. Ahora echemos un vistazo al código que acabamos de ver. Cuando hacemos un bucle para cada i, todos los primos que no superan a i ya están registrados en el contenedor `prime`. Por lo tanto, si solo hacemos un bucle para todos los elementos en `prime` en el bucle interno, rompiendo cuando el elemento divide a i, podemos seleccionar cada compuesto exactamente una vez.

Listing 2: Criba Lineal

```
#include <vector>
#include <algorithm>

std::vector<int> prime;
bool is_composite[MAXN];
```

```
void sieve(int n) {
    std::fill(is_composite, is_composite + n, false);
    for (int i = 2; i < n; ++i) {
        if (!is_composite[i]) prime.push_back(i);
        for (int j = 0; j < prime.size() && i * prime[j] < n; ++j) {
            is_composite[i * prime[j]] = true;
            if (i % prime[j] == 0) break;
        }
    }
}
```

Como se muestra en el código, la instrucción `if (i % prime[j] == 0) break;` termina el bucle cuando p divide a i. De acuerdo con el análisis anterior, podemos ver que el bucle interno se ejecuta solo una vez para cada compuesto. Por lo tanto, el código anterior tiene una complejidad de $O(n)$, lo que resulta en su nombre: la criba 'lineal'.

Función Multiplicativa

Hay un tipo específico de función que muestra importancia en el estudio de la teoría de números: la función multiplicativa. Por definición, una función $f(x)$ definida en todos los enteros positivos es multiplicativa si cumple la siguiente condición:

Para cada par de enteros coprimos p y q, $f(pq) = f(p)f(q)$.

Aplicando la definición, se puede mostrar que $f(n) = f(n)f(1)$. Así que, a menos que para cada entero n tengamos $f(n) = 0$, $f(1)$ debe ser 1. Además, dos funciones multiplicativas $f(n)$ y $g(n)$ son idénticas si y solo si para cada primo p y entero no negativo k, $f(p^k) = g(p^k)$ es verdadero. Se puede inferir que para una función multiplicativa $f(n)$, será suficiente conocer su representación en $f(p^k)$.

Las siguientes funciones son funciones multiplicativas más o menos comúnmente utilizadas, según Wikipedia:

- La función constante $I(p^k) = 1$.
- La función de potencia $Ida(p^k) = p^a k$, donde a es constante.
- La función unitaria $\varepsilon(p^k) = [p^k = 1]$ ($[P]$ es 1 cuando P es verdadero y 0 en caso contrario).
- La función de los divisores, denotando la suma de las a-ésimas potencias de todos los divisores positivos del número.
- La función de Möbius $\mu(p^k) = [k = 0] - [k = 1]$.
- La función totient de Euler $\phi(p^k) = p^k - p^{k-1}$.

Es interesante que la criba lineal también se pueda utilizar para encontrar todos los valores de una función multiplicativa $f(x)$ en un rango dado $[1, n]$. Para hacerlo, debemos observar más de cerca el código de la criba lineal. Como podemos ver, cada entero x se seleccionará solo una vez, y debemos conocer una de las siguientes propiedades:

1. x es primo. En este caso, podemos determinar el valor de $f(x)$ directamente.

2. $x = ip$ y p no divide a i. En este caso, sabemos que $f(x) = f(i)f(p)$. Dado que tanto $f(i)$ como $f(p)$ ya se conocen antes, simplemente podemos multiplicarlos juntos.

3. $x = ip$ y p divide a i. Este es un caso más complicado en el que tenemos que descubrir una relación entre $f(i)$ y $f(ip)$. Afortunadamente, en la mayoría de las situaciones, existe una relación simple entre ellos. Por ejemplo, en la función totient de Euler, podemos inferir fácilmente que $\phi(ip) = p \cdot \phi(i)$.

Dado que podemos calcular el valor de la función para x que satisface cualquiera de las propiedades anteriores, simplemente podemos modificar la criba lineal para encontrar todos los valores de la función multiplicativa de 1 a n. El siguiente código implementa un ejemplo en la función totient de Euler.

Listing 3: Criba Lineal para la Función Totient de Euler

```
#include <vector>
#include <algorithm>

std::vector<int> prime;
bool is_composite[MAXN];
int phi[MAXN];

void sieve(int n) {
    std::fill(is_composite, is_composite + n, false);
    phi[1] = 1;
    for (int i = 2; i < n; ++i) {
        if (!is_composite[i]) {
            prime.push_back(i);
            phi[i] = i - 1; // i es primo
        }
        for (int j = 0; j < prime.size() && i * prime[j] < n; ++j) {
            is_composite[i * prime[j]] = true;
            if (i % prime[j] == 0) {
                phi[i * prime[j]] = phi[i] * prime[j]; // prime[j] divide i
                break;
            } else {
                phi[i * prime[j]] = phi[i] * phi[prime[j]]; // prime[j] no di
            }
        }
    }
}
```

Extensión de la Criba Lineal

Bueno, puede que no siempre sea posible encontrar una fórmula cuando p divide a i. Por ejemplo, puedo escribir alguna función multiplicativa aleatoria $f(p^k) = k$ que es difícil de inferir una fórmula. Sin embargo, aún hay una forma de aplicar la criba lineal en tal función. Podemos mantener un arreglo contador `cnt[i]` que denote la potencia del menor factor primo de i, y encontrar una fórmula usando el arreglo ya que $\phi(i) = i - \phi(i-1)$. El siguiente código da un ejemplo de la función que acabo de escribir.

Listing 4: Criba Lineal para una Función Multiplicativa Aleatoria

```
#include <vector>
#include <algorithm>

std::vector<int> prime;
bool is_composite[MAXN];
int func[MAXN], cnt[MAXN];

void sieve(int n) {
    std::fill(is_composite, is_composite + n, false);
    func[1] = 1;
    for (int i = 2; i < n; ++i) {
        if (!is_composite[i]) {
            prime.push_back(i);
            func[i] = 1;
            cnt[i] = 1;
        }
        for (int j = 0; j < prime.size() && i * prime[j] < n; ++j) {
            is_composite[i * prime[j]] = true;
            if (i % prime[j] == 0) {
                func[i * prime[j]] = func[i] / cnt[i] * (cnt[i] + 1);
                cnt[i * prime[j]] = cnt[i] + 1;
                break;
            } else {
                func[i * prime[j]] = func[i] * func[prime[j]];
                cnt[i * prime[j]] = 1;
            }
        }
    }
}
```