GUÍA DE APRENDIZAJE PARA CONCURSANTES ICPC Y IOI: ALGORITMO DE LA CRIBA LINEAR

1. Introducción

Dado un número N, encuentra todos los números primos en un segmento $[2, N]$.

La forma estándar de resolver una tarea es usar la criba de Eratóstenes. Este algoritmo es muy simple, pero tiene tiempo de ejecución $O(n \log \log n)$

Aunque hay muchos algoritmos conocidos con tiempo de ejecución sublineal (es decir,$O(N)$), el algoritmo descrito a continuación es interesante por su simplicidad: no es más complejo que el clásico criba de Eratóstenes.

2. Conocimientos previos

2.1. Criba de Eratóstenes

La Criba de Eratóstenes es un método antiguo y eficaz para encontrar todos los números primos menores que un número dado. Fue desarrollado por el matemático griego Eratóstenes en el siglo III a.C. y sigue siendo utilizado en la actualidad debido a su simplicidad y eficacia.

3. Desarrollo

El algoritmo se centra en calcular el factor primo mínimo para cada número en el rango $[2, N]$.

Además, necesitamos almacenar la lista de todos los números primos encontrados, llamémoslo $pr[]$

Inicializaremos los valores $lp[i]$ con ceros, lo que significa que asumimos que todos los números son primos. Durante la ejecución del algoritmo, este vector se llenará gradualmente.

Ahora repasaremos los números del 2 al N. Tenemos dos casos para el número actual i:

1. $lp[i] = 0$ significa que i es primo, es decir, no hemos encontrado factores menores para él. Por lo tanto, asignamos $lp[i] = i$ y agregamos i al final de la lista $pr[]$.
2. $lp[i]! = 0$ significa que i es compuesto y que su mínimo factor primo es $lp[i]$

En ambos casos actualizamos los valores $lp[]$ de para los números que son divisibles por i. Sin embargo, nuestro objetivo es aprender a hacerlo para establecer un valor como máximo una vez para cada número $lp[]$. Podemos hacerlo de la siguiente manera:

Consideremos los números $x_j = i \cdot p_j$ donde p_j son todos los números primos menores o iguales que $lp[i]$ (por eso necesitamos almacenar la lista de todos los números primos).

Estableceremos un nuevo valor $lp[x_j] = p_j$ para todos los números de esta forma.

3.1. Prueba de corrección

Necesitamos demostrar que el algoritmo establece todos los valores $lp[]$ correctamente y que cada valor se establecerá exactamente una vez. Por lo tanto, el algoritmo tendrá un tiempo de

ejecución lineal, ya que todas las acciones restantes del algoritmo, obviamente, funcionan para $O(n)$.

Observe que cada número i tiene exactamente una representación en la forma:

$$i = lp[i] \cdot x,$$

dónde $lp[i]$ es el factor primo mínimo de i, y el número x no tiene factores primos menores que $lp[i]$, es decir:

$$lp[i] \leq lp[x]$$

Ahora, comparemos esto con las acciones de nuestro algoritmo: de hecho, por cada x pasa por todos los números primos por los que se puede multiplicar, es decir, todos los números primos hasta$lp[x]$ inclusive, para obtener los números en la forma dada anteriormente.

Por lo tanto, el algoritmo revisará cada número compuesto exactamente una vez, estableciendo los valores correctos $lp[]$ allá.

4. Implementación

4.1. C++

```
const int N = 10000000;
vector<int> lp(N+1);
vector<int> pr;

for (int i=2; i <= N; ++i) {
   if (lp[i] == 0) {
      lp[i] = i;
      pr.push_back(i);
   }
   for (int j=0; j < (int)pr.size() && pr[j] <= lp[i] && i*pr[j] <= N; ++j) {
      lp[i * pr[j]] = pr[j];
   }
}
```

4.2. Java

```
final int N = 10000000;
int [] lp = new int[N+1];
Arrays.fill(lp,0);
List<Integer> pr = new ArrayList<Integer>();
for (int i=2; i <= N; ++i) {
   if (lp[i] == 0) {
```

```
            lp[i] = i;
            pr.add(i);
        }
        for (int j=0; j < (int)pr.size() && pr.get(j) <= lp[i] && i*pr.get(j) <= N;
             ++j) {
            lp[i * pr.get(j)] = pr.get(j);
        }
    }
```

5. Aplicaciones

La cualidad redentora es que este algoritmo calcula una matriz $lp[]$, que nos permite encontrar la factorización de cualquier número en el rango $[2; n]$ en el momento del orden de tamaño de esta factorización. Además, usar solo una matriz extra nos permitirá evitar divisiones cuando busquemos factorización.

Conocer las factorizaciones de todos los números es muy útil para algunas tareas, y este algoritmo es uno de los pocos que permite encontrarlos en tiempo lineal.

La debilidad del algoritmo dado es que usa más memoria que la criba clásica de Eratóstenes: requiere una matriz de números, mientras que para criba clásico de Eratóstenes basta con tener bits de memoria (que es 32 veces menos).

Por lo tanto, tiene sentido usar el algoritmo descrito solo hasta que para números de orden y no mayor 10^7.

6. Complejidad

Aunque el tiempo de ejecución de $O(n)$ es mejor que $O(n \log \log n)$ de la criba de Eratóstenes, la diferencia entre ellos no es tan grande. En la práctica, la criba lineal corre tan rápido como una implementación típica del criba de Eratóstenes.

En comparación con las versiones optimizadas de la criba de Eratóstenes, por ejemplo la criba por bloques, es mucho más lento. Teniendo en cuenta los requisitos de memoria de este algoritmo: una matriz $lp[]$ de longitud n, y una variedad de $pr[]$ de longitud $\frac{n}{\ln n}$, este algoritmo parece ser peor que la criba clásico en todos los sentidos.