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

1. Introducción

La Criba de Eratóstenes es un método antiguo y eficaz para encontrar todos los números primos menores que un número dado es decir encontrar todos los números primos en un rango de $[1; n]$. 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.

2. Conocimientos previos

2.1. Eratóstenes

Eratóstenes de Cirene fue un matemático, astrónomo, geógrafo, poeta y filósofo griego que vivió en el siglo III a.C. Es conocido principalmente por su método para encontrar números primos, llamado la Criba de Eratóstenes.

Eratóstenes fue un polímata griego cuyas contribuciones en matemáticas, geografía y otras disciplinas han dejado un legado duradero en la historia del conocimiento humano.

2.2. Números primos

Los números primos son aquellos números naturales mayores que 1 que solo tienen dos divisores: el 1 y ellos mismos. En otras palabras, un número primo es aquel que no puede ser dividido de manera exacta por ningún otro número distinto de 1 y de sí mismo.

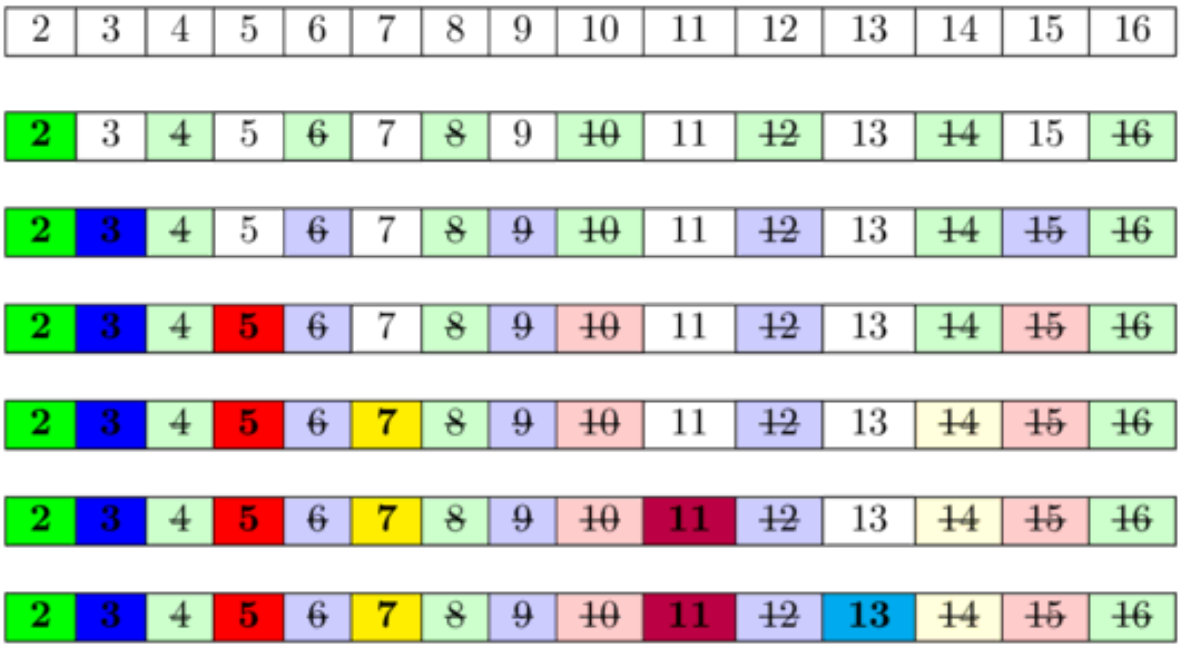
2.3. Números compuestos

Los números compuestos son aquellos enteros positivos que tienen mas de dos divisores distintos, es decir, ue pueden ser divididos por uno, por si mismos y por al menos otro número.

3. Desarrollo

El algoritmo es muy simple: al principio anotamos todos los números entre 2 y n. Marcamos todos los múltiplos propios de 2 (ya que 2 es el número primo más pequeño) como compuestos. Un múltiplo propio de un número x, es un número mayor que x y divisible por x. Luego encontramos el siguiente número que no ha sido marcado como compuesto, en este caso es 3. Lo que significa que 3 es primo y marcamos todos los múltiplos propios de 3 como compuestos. El siguiente número sin marcar es 5, que es el siguiente número primo, y marcamos todos los múltiplos propios del mismo. Y continuamos este procedimiento hasta que hayamos procesado todos los números de la fila.

En la siguiente imagen puedes ver una visualización del algoritmo para calcular todos los números primos en el rango $[1; 16]$. Se puede ver que muy a menudo marcamos los números como compuestos varias veces.



La idea detrás es la siguiente: un número es primo si ninguno de los números primos más pequeños lo divide. Dado que iteramos sobre los números primos en orden, ya marcamos todos los números que son divisibles por al menos uno de los números primos como divisibles. Por lo tanto, si llegamos a una celda y no está marcada, entonces no es divisible por ningún número primo menor y, por lo tanto, tiene que ser primo.

3.1. Diferentes optimizaciones de la criba de Eratóstenes

La mayor debilidad del algoritmo es que camina a lo largo de la memoria varias veces, manipulando únicamente elementos individuales. Esto no es muy amigable con el caché. Y por eso, la constante que se esconde en $O(n \log \log n)$ es comparativamente grande.

Además, la memoria consumida es un cuello de botella para las grandes n.

Los métodos presentados a continuación nos permiten reducir la cantidad de operaciones realizadas, así como acortar notablemente la memoria consumida.

3.1.1. Iterar hasta la raíz

Obviamente, para encontrar todos los números primos hasta n, bastará con realizar el cribado únicamente por los números primos, que no superen la raíz de n.

3.1.2. Iterar solo por los números impares

Dado que todos los números pares (excepto 2) son compuestos, podemos dejar de verificar los números pares. En cambio, necesitamos operar sólo con números impares.

3.1.3. Reducir la memoria consumida

Debemos notar que estas dos optimizaciones de la criba de Eratóstenes usan n bits de memoria mediante el uso de la estructura de datos **vector<bool>**. **vector<bool>** no es un contenedor normal que almacena una serie de **bool** (como en la mayoría de las arquitecturas de computadora, **bool** ocupa un byte de memoria). Es una especialización de optimización de memoria de **vector<T>**, que solo consume $\frac{N}{8}$ bytes de memoria.

Las arquitecturas de procesadores modernas funcionan mucho más eficientemente con bytes que con bits, ya que normalmente no pueden acceder a los bits directamente. Entonces, debajo **vector<bool>** almacena los bits en una gran memoria continua, accede a la memoria en bloques de unos pocos bytes y extrae/establece los bits con operaciones de bits como enmascaramiento de bits y desplazamiento de bits.

Debido a eso, hay una cierta sobrecarga cuando lees o escribes bits con a **vector<bool>** y, muy a menudo, usar a **vector<char>** (que usa 1 byte para cada entrada, por lo que 8 veces la cantidad de memoria) es más rápido.

Sin embargo, para las implementaciones simples de la criba de Eratóstenes, usar a **vector<bool>** es más rápido. Está limitado por la rapidez con la que puede cargar los datos en la memoria caché y, por lo tanto, utilizar menos memoria ofrece una gran ventaja. Un punto de referencia muestra que usar a **vector<bool>** es entre 1,4 y 1,7 veces más rápido que usar a vector<char>.

Las mismas consideraciones se aplican también a **bitset**. También es una forma eficaz de almacenar bits, similar a **vector<bool>**, por lo que sólo se necesita $\frac{N}{8}$ bytes de memoria, pero es un poco más lento en el acceso a los elementos. En el punto de referencia anterior **bitset** se desempeña un poco peor que **vector<bool>**. Otro inconveniente **bitset** es que es necesario conocer el tamaño en el momento de la compilación.

3.1.4. Iterar por bloque

De la optimización *iterar hasta la raíz* se deduce que no es necesario conservar toda la matriz is_prime[$1 \ldots n$]en todo momento. Para tamizar basta con mantener los números primos hasta la raíz de n, es decir is_prime[$1 \ldots \sqrt{n}$], dividir la gama completa en bloques y iterar cada bloque por separado.

Dejar s ser una constante que determina el tamaño del bloque, entonces tenemos $\lceil \frac{n}{s} \rceil$ bloques por completo, y el bloque k ($k = 0 \ldots \lfloor \frac{n}{s} \rfloor$) contiene los números en un segmento $[ks; ks + s - 1]$. Podemos trabajar en bloques por turnos, es decir, por cada bloque k repasaremos todos los números primos (desde 1 a $\sqrt{n}$) y realizar el tamizado con ellos. Vale la pena señalar que tenemos que modificar un poco la estrategia cuando manejamos los primeros números: primero, todos los números primos de $[1; \sqrt{n}]$ no deberían retirarse; y segundo, los números 0 y 1 deben marcarse

como números no primos. Mientras trabaja en el último bloque no debe olvidar que el último número necesario n no necesariamente está ubicado al final de la cuadra.

Como se analizó anteriormente, la implementación típica del criba de Eratóstenes está limitada por la velocidad con la que se pueden cargar datos en las cachés de la CPU. Dividiendo el rango de números primos potenciales $[1; n]$ en bloques más pequeños, nunca tenemos que mantener varios bloques en la memoria al mismo tiempo y todas las operaciones son mucho más amigables con el caché. Como ahora ya no estamos limitados por las velocidades de caché, podemos reemplazar **vector<bool>** con a **vector<char>** y obtener algo de rendimiento adicional, ya que los procesadores pueden manejar lecturas y escrituras con bytes directamente y no necesitan depender de operaciones de bits para extraer bits individuales. El punto de referencia muestra que usar a **vector<char>** es aproximadamente 3 veces más rápido en esta situación que usar a **vector<bool>**. Una advertencia: esos números pueden diferir según la arquitectura, el compilador y los niveles de optimización.

3.2. Encontrar los primos en un rango

A veces necesitamos encontrar todos los números primos en un rango $[L, R]$ de pequeño tamaño (por ej. $R - L + 1 \approx 1e7$), dónde R puede ser muy grande (por ejemplo $1e12$).

Para resolver este problema, podemos utilizar la idea del tamiz segmentado. Pregeneramos todos los números primos hasta $\sqrt{R}$ y usa esos números primos para marcar todos los números compuestos en el segmento $[L, R]$.

4. Implementación

4.1. C++

4.1.1. Implementación clásica

```
int n;
vector<bool> is_prime(n+1, true);
is_prime[0] = is_prime[1] = false;
for (int i = 2; i <= n; i++) {
   if (is_prime[i] && (long long)i * i <= n) {
      for (int j = i * i; j <= n; j += i)
      is_prime[j] = false;
   }
}
```

4.1.2. Iterar hasta la raíz

```
int n;
vector<bool> is_prime(n+1, true);
is_prime[0] = is_prime[1] = false;
for (int i = 2; i * i <= n; i++) {
```

```
    if (is_prime[i]) {
        for (int j = i * i; j <= n; j += i)
            is_prime[j] = false;
    }
}
```

4.1.3. Iterar por bloques

```
int count_primes(int n) {
    const int S = 10000;
    vector<int> primes;
    int nsqrt = sqrt(n);
    vector<char> is_prime(nsqrt + 2, true);
    for (int i = 2; i <= nsqrt; i++) {
        if (is_prime[i]) {
            primes.push_back(i);
            for (int j = i * i; j <= nsqrt; j += i)
            is_prime[j] = false;
        }
    }

    int result = 0;
    vector<char> block(S);
    for (int k = 0; k * S <= n; k++) {
        fill(block.begin(), block.end(), true);
        int start = k * S;
        for (int p : primes) {
            int start_idx = (start + p - 1) / p;
            int j = max(start_idx, p) * p - start;
            for (; j < S; j += p)
                block[j] = false;
        }
        if (k == 0) block[0] = block[1] = false;
        for (int i = 0; i < S && start + i <= n; i++) {
            if (block[i]) result++;
        }
    }
    return result;
}
```

4.1.4. Encontrar los primos en un rango

```
vector<char> segmentedSieve(long long L, long long R) {
// generar todos los numeros primos hasta sqrt(R)
    long long lim = sqrt(R);
    vector<char> mark(lim + 1, false);
    vector<long long> primes;
```

```
   for (long long i = 2; i <= lim; ++i) {
      if (!mark[i]) {
         primes.emplace_back(i);
         for (long long j = i * i; j <= lim; j += i) mark[j] = true;
      }
   }
   vector<char> isPrime(R - L + 1, true);
   for (long long i : primes)
      for (long long j = max(i * i, (L + i - 1) / i * i); j <= R; j += i)
         isPrime[j - L] = false;
   if(L == 1)
      isPrime[0] = false;
   return isPrime;
}
```

También es posible que no generemos previamente todos los números primos:

```
vector<char> segmentedSieveNoPreGen(long long L, long long R) {
   vector<char> isPrime(R - L + 1, true);
   long long lim = sqrt(R);
   for (long long i = 2; i <= lim; ++i)
      for (long long j = max(i * i, (L + i - 1) / i * i); j <= R; j += i)
            isPrime[j - L] = false;
   if (L == 1) isPrime[0] = false;
   return isPrime;
}
```

4.2. Java

4.2.1. Implementación clásica

```
int n;
boolean [] is_prime = new boolean [n+1];
Arrays.fill(is_prime,true);
is_prime[0] = is_prime[1] = false;
for (int i = 2; i <= n; i++) {
   if (is_prime[i] && i * i <= n) {
      for (int j = i * i; j <= n; j += i)
         is_prime[j] = false;
   }
}
```

4.2.2. Iterar hasta la raíz

```
int n;
boolean[] is_prime= new boolean[n+1];
```

```
Arrays.fill(is_prime,true);
is_prime[0] = is_prime[1] = false;
for (int i = 2; i * i <= n; i++) {
   if (is_prime[i]) {
      for (int j = i * i; j <= n; j += i)
         is_prime[j] = false;
   }
}
```

4.2.3. Iterar por bloques

```
public static int count_primes(int n) {
   final int S = 10000;
   List<Integer> primes =new ArrayList<Integer>();
   int nsqrt = (int)Math.sqrt(n);
   char [] is_prime=new char [nsqrt + 2];
   Arrays.fill(is_prime,'T');
   for (int i = 2; i <= nsqrt; i++) {
      if (is_prime[i]=='T') {
         primes.add(i);
         for (int j = i * i; j <= nsqrt; j += i)
            is_prime[j] = 'F';
      }
   }
   int result = 0;
   char [] block =new char [S];
   for (int k = 0; k * S <= n; k++) {
      Arrays.fill(block, 'T');
      int start = k * S;
      for (int p : primes) {
         int start_idx = (start + p - 1) / p;
         int j = Math.max(start_idx, p) * p - start;
         for (; j < S; j += p)
            block[j] = 'F';
      }
      if (k == 0) block[0] = block[1] = 'F';
         for (int i = 0; i < S && start + i <= n; i++) {
            if (block[i]=='T') result++;
      }
   }
   return result;
}
```

4.2.4. Encontrar los primos en un rango

```
public static char [] segmentedSieve(int L, int R) {
   // generar todos los numeros primos hasta sqrt(R)
```

```
   int lim = (int)Math.sqrt(R);
   char[] mark= new char[lim + 1];
   Arrays.fill(mark,'F');
   ArrayList<Integer> primes = new ArrayList<Integer>();
   for (int i = 2; i <= lim; ++i) {
      if (mark[i]=='F') {
         primes.add(i);
         for (int j = i * i; j <= lim; j += i) mark[j] = 'T';
      }
   }
   char [] isPrime = new char[R - L + 1];
   Arrays.fill(isPrime,'T');
   for (int i : primes)
      for (int j = Math.max(i * i, (L + i - 1) / i * i); j <= R; j += i)
         isPrime[j - L] = 'F';
   if(L == 1) isPrime[0] = 'F';
   return isPrime;
}
```

También es posible que no generemos previamente todos los números primos:

```
public static char[] segmentedSieveNoPreGen(int L, int R) {
   char [] isPrime=new char[R - L + 1];
   Arrays.fill(isPrime,'T');
   int lim = (int)Math.sqrt(R);
   for (int i = 2; i <= lim; ++i)
      for (int j = Math.max(i * i, (L + i - 1) / i * i); j <= R; j += i)
         isPrime[j - L] = 'F';
   if (L == 1) isPrime[0] = 'F';
   return isPrime;
}
```

La implementación clásica primero marca todos los números excepto el cero y el uno como números primos potenciales, luego comienza el proceso de cribado de números compuestos. Para esto itera sobre todos los números de 2 a n. Si el número actual i es un número primo, marca todos los números que son múltiplos de i como números compuestos, a partir de i^2. Esto ya es una optimización sobre una forma ingenua de implementarlo y está permitido ya que todos los números más pequeños que son múltiplos de i necesario también tener un factor primo que sea menor que i, por lo que todos ellos ya fueron tamizados antes. Desde i^2 puede desbordar fácilmente el tipo **int**, la verificación adicional se realiza usando el tipo **long long** antes del segundo bucle anidado.

5. Aplicaciones

La criba de Eratóstenes es un algoritmo utilizado para encontrar todos los números primos hasta un número dado. Algunas aplicaciones de este algoritmo son:

1. **Encriptación de datos:** La criba de Eratóstenes se puede utilizar en criptografía para generar claves públicas y privadas basadas en números primos.
2. **Optimización de algoritmos:** Este algoritmo se puede utilizar para optimizar algoritmos que requieran el uso de números primos, como el algoritmo de Euclides para encontrar el máximo común divisor.
3. **Seguridad informática:** La criba de Eratóstenes se puede utilizar en la generación de números primos aleatorios para mejorar la seguridad en sistemas informáticos.
4. **Matemáticas computacionales:** Este algoritmo se utiliza en la teoría de números computacionales para encontrar números primos de manera eficiente.

En resumen, la criba de Eratóstenes tiene diversas aplicaciones en campos como la criptografía, la seguridad informática, la optimización de algoritmos y las matemáticas computacionales.

6. Complejidad

La implementación clásica consume $O(n)$ de la memoria (obviamente) y realiza $O(n \log \log n)$ operaciones. Es sencillo demostrar un tiempo de ejecución de $O(n \log n)$ sin saber nada sobre la distribución de números primos: ignorando la *is_prime* verificación, el bucle interno se ejecuta (como máximo) n/i tiempos para $i = 2, 3, 4, \ldots$, lo que hace que el número total de operaciones en el bucle interno sea una suma armónica como $n(1/2 + 1/3 + 1/4 + \cdots)$, que está delimitada por $O(n \log n)$.

Demostremos que el tiempo de ejecución del algoritmo es $O(n \log \log n)$. El algoritmo realizará $\frac{n}{p}$ operaciones para cada primo $p \leq n$ en el bucle interior. Por tanto, necesitamos evaluar la siguiente expresión:

$$\sum_{\substack{p \leq n, \\ p \text{ prime}}} \frac{n}{p} = n \cdot \sum_{\substack{p \leq n, \\ p \text{ prime}}} \frac{1}{p}.$$

Recordemos dos hechos conocidos.

- El número de números primos menores o iguales a n es aproximadamente $\frac{n}{\ln n}$.
- El k-ésimo número primo aproximadamente igual $k \ln k$ (esto se sigue inmediatamente del hecho anterior).

Así podemos escribir la suma de la siguiente manera:

$$\sum_{\substack{p \leq n, \\ p \text{ prime}}} \frac{1}{p} \approx \frac{1}{2} + \sum_{k=2}^{\frac{n}{\ln n}} \frac{1}{k \ln k}.$$

Aquí extrajimos el primer número primo 2 de la suma, porque $k = 1$ en aproximación $k \ln k$ es 0 y provoca una división por cero.

Ahora, evalúemos esta suma usando la integral de una misma función sobre k de 2 a $\frac{n}{\ln n}$ (Podemos hacer tal aproximación porque, de hecho, la suma está relacionada con la integral como su aproximación usando el método del rectángulo):

$$\sum_{k=2}^{\frac{n}{\ln n}} \frac{1}{k \ln k} \approx \int_{2}^{\frac{n}{\ln n}} \frac{1}{k \ln k} dk$$

La antiderivada del integrando es $\ln \ln k$. Usando una sustitución y eliminando términos de orden inferior, obtendremos el resultado:

$$\int_{2}^{\frac{n}{\ln n}} \frac{1}{k \ln k} dk = \ln \ln \frac{n}{\ln n} - \ln \ln 2 = \ln(\ln n - \ln \ln n) - \ln \ln 2 \approx \ln \ln n$$

Ahora, volviendo a la suma original, obtendremos su valoración aproximada:

$$\sum_{\substack{p \leq n, \\ p \ is \ prime}} \frac{n}{p} \approx n \ln \ln n + o(n)$$

Puede encontrar una prueba más estricta (que brinda una evaluación más precisa y precisa dentro de multiplicadores constantes) en el libro escrito por Hardy & Wright *Una introducción a la teoría de los números* (p. 349).

La optimización de iterar hasta la raíz no afecta la complejidad (de hecho, al repetir la prueba presentada anteriormente obtendremos la evaluación $n \ln \ln \sqrt{n} + o(n)$, que es asintóticamente igual según las propiedades de los logaritmos), aunque el número de operaciones se reducirá notablemente.

La optimización de iterar los impares primero, nos permitirá reducir a la mitad la memoria necesaria. En segundo lugar, reducirá el número de operaciones realizadas por el algoritmo aproximadamente a la mitad.

El tiempo de ejecución de iterar por bloques es el mismo que el del tamiz normal de Eratóstenes (a menos que el tamaño de los bloques sea muy pequeño), pero la memoria necesaria se reducirá a $O(\sqrt{n} + S)$ y tenemos mejores resultados de almacenamiento en caché. Por otro lado, habrá una división para cada par de bloque y número primo de $[1; \sqrt{n}]$, y eso será mucho peor para bloques de menor tamaño. Por lo tanto, es necesario mantener el equilibrio al seleccionar la constante S. Logramos los mejores resultados para tamaños de bloque entre 10^4 y 10^5.

La complejidad temporal del primer enfoque de encontrar los primos en un rango es $O((R - L + 1) \log \log(R) + \sqrt{R} \log \log \sqrt{R})$. Mientras el segundo enfoque la complejidad es peor, lo cual es $O((R - L + 1) \log(R) + \sqrt{R})$. Sin embargo, en la práctica sigue funcionando muy rápido.

Podemos modificar el algoritmo de tal manera que solo tenga complejidad de tiempo lineal. Este enfoque lo vamos analizar en otro momento. Sin embargo, este algoritmo también tiene sus propias debilidades.

7. Ejercicios

A continuación una lista de ejercicios que se puede resolver aplicando el algoritmo abordado:

- DMOJ - Números Primos de nuevo
- DMOJ - Cuántos primos divide el número
- DMOJ - Primorial
- DMOJ - Conjeturas de Goldbach
- Leetcode - Four Divisors
- Leetcode - Count Primes
- SPOJ - Printing Some Primes
- SPOJ - A Conjecture of Paul Erdos
- SPOJ - Primal Fear
- SPOJ - Primes Triangle (I)
- SPOJ - Namit in Trouble
- SPOJ - Bazinga!
- SPOJ - N-Factorful
- SPOJ - Binary Sequence of Prime Numbers
- SPOJ - Prime Generator
- SPOJ - Printing some primes
- Codeforces - Almost Prime
- Codeforces - Sherlock And His Girlfriend
- Codeforces - Nodbach Problem
- Codeforces - Colliders
- Project Euler - Prime pair connection
- UVA 11353 - A Different Kind of Sorting