Nivel básico: Teoría de Números

Instructor: Leonardo Valverde Lara

Febrero 20, 2024

§1. Preliminares

Sean a y b enteros con $a \neq 0$, decimos que « a divide a b » si existe un entero k tal que $ak = b$ y se representa como $a \mid b$. Si a **no** divide a b escribimos $a \nmid b$.

> **Observación.** Otras formas de expresar que « a divide a b » son « a es divisor de b », « a es factor de b» o « b es múltiplo de a».

Ejemplos:

- $-5 \mid 15$, (aquí $k = -3$).
- $1 \mid -13$, (aquí $k = -13$).
- $8 \mid 0$, (aquí $k = 0$).
- $4 \mid 36$, (aquí $k = 9$).

Sean a, b y $c \in \mathbb{Z}$, por la definición, se concluyen las siguientes propiedades.

1. Si $a \neq 0$ entonces $a \mid a$.
2. Si $a \mid b$ y $a \mid c$, entonces $a \mid b \pm c$.
3. Si $a \mid b$, entonces $a \mid bc$ para todo $c \in \mathbb{Z}$.
4. Si $a \mid b$ y $a \mid b \pm c$, entonces $a \mid c$.
5. Si $a \mid b$ y $b \mid c$, entonces $a \mid c$.

Dados a, b enteros, y n un entero positivo, decimos que «a y b son *congruentes módulo* n» sí y solo sí a y b dejan el mismo resto al ser divididos entre n, y se denota por $a \equiv b \pmod{n}$.

> **Observación.** Otra forma común de definir la congruencia es $a \equiv b \pmod{n} \iff n \mid a - b$

Ejemplos:

- $9 \equiv 25 \pmod{4}$
- $2022 \equiv -1 \pmod{7}$
- $67 \equiv 4 \pmod{21}$
- $-12 \equiv 8 \pmod{5}$
- $15 \equiv 0 \pmod{3}$
- $5 \equiv -3 \pmod{2}$

Por la definición se concluyen las siguientes propiedades:

1. (Reflexividad) $a \equiv a \pmod{n}$.

2. (Transitividad) Si $a \equiv b \pmod{n}$ y $b \equiv c \pmod{n}$, entonces $a \equiv c \pmod{n}$.
3. (Simetría) Si $a \equiv b \pmod{n}$, entonces $b \equiv a \pmod{n}$.
4. Para todo k entero, si $a \equiv b \pmod{n}$, entonces $ak \equiv bk \pmod{n}$.
5. Si $a \equiv b \pmod{n}$ y $c \equiv d \pmod{n}$, entonces:

$$a + c \equiv b + d \pmod{n} \text{ y } ab \equiv cd \pmod{n}$$

6. Si $a \equiv b \pmod{n}$, entonces $a^m \equiv b^m \pmod{n}$, para todo entero positivo m.
7. Si $ab \equiv 0 \pmod{n}$ y $\gcd(b, n) = 1$, entonces $a \equiv 0 \pmod{n}$.

§2. Divisores de un entero positivo N

Dado un entero positivo N nuestro objetivo es hallar todos sus divisores. El primer enfoque sería iterar todos los números del 1 al N y verificar cuál de ellos es divisor de este número.

```
vector<int> divisor(int N){
    vector<int> div;
    for(int d = 1; d <= N; d++){
        if(N % d == 0){
            div.push_back(d);
        }
    }
    return d;
}
```

Sin embargo la complejidad de esta función es $O(N)$. Veamos una forma de optimizarlo empleando el siguiente lema.

Lema 1.

Sea $d \in \mathbb{Z}$ un divisor de N, entonces se cumple que $d \leq \sqrt{N}$ o $\frac{N}{d} \leq \sqrt{N}$.

De este modo es posible agrupar todos los divisores de N en parejas de la forma $(d, \frac{N}{d})$, de modo que al menos uno de ellos sea menor o igual a $\sqrt{N}$. Por ende basta con iterar todos los divisores $d \leq \sqrt{N}$ e insertar también $\frac{N}{d}$ (solo si es distinto de d).

```
vector<int> divisor(int N){
    vector<int> div;
    for(int d = 1; d * d <= N; d++){
        if(N % d == 0){
            div.push_back(d);
            if(d != N / d){
                div.push_back(N / d);
            }
        }
    }
    return d;
}
```

Esto reduce la complejidad a $O(\sqrt{N})$.

§3. Test de Primalidad

Nuestro objetivo ahora es verificar si un entero positivo N es primo o no. Un primer enfoque al igual que en la sección anterior sería iterar todos los números $2 \leq p \leq N-1$. Si algún p cumple que $p \mid N$ entonces N no es primo, caso contrario N es primo.

```
bool is_prime(int N){
    bool prime = true;
    for(int p = 2; p <= N - 1; p++){
        if(N % p == 0){
            prime = false;
            break;
        }
    }
    return prime;
}
```

La complejidad de esta función es $O(N)$. Para optimizarla usaremos el siguiente lema.

Lema 2.

Sea N un entero positivo compuesto y sea p su menor divisor primo, entonces se cumple que $p \leq \sqrt{N}$.

Basándonos en lo anterior, basta con buscar aquel primo p iterando de 2 a $\sqrt{N}$. Si no se encuentra ningún divisor en ese intervalo entonces N es primo, esto reduce la complejidad a $O(\sqrt{N})$.

```
bool is_prime(int N){
    bool prime = true;
    for(int p = 2; p * p <= N; p++){
        if(N % p == 0){
            prime = false;
            break;
        }
    }
    return prime;
}
```

§4. Factorización

Sea N un entero positivo, lo que se busca es obtener la descomposición de dicho número en sus factores primos, es decir, sea:

$$N = p_1^{\alpha_1} p_2^{\alpha_2} \cdots p_k^{\alpha_k}$$

siendo $p_1 \leq p_2 \leq \ldots \leq p_k$ sus divisores primos, el objetivo es obtener todas las parejas $\{(p_1, \alpha_1), (p_2, \alpha_2), \ldots, (p_k, \alpha_k)\}$.

Usando el lema 2 podemos notar que $p_1 \leq \sqrt{N}$, entonces podemos hallar p_1 en el intervalo $[2, \sqrt{N}]$. Luego, sea

$$N' = \frac{N}{p_1^{\alpha_1}} = p_2^{\alpha_2} \cdots p_k^{\alpha_k}$$

Usando nuevamente el lema 2 se cumple que $p_2 \leq \sqrt{N'}$, por ende p_2 se puede encontrar en el intervalo $[2, \sqrt{N'}]$. De este modo el algoritmo quedaría de la siguiente forma:

```
vector<pair<int, int>> fact(int N){
    vector<pair<int, int>> f;
    if(N == 1){
        f.push_back(make_pair(1, 1));
        return f;
    }
    for(int p = 2; p * p <= N; p++){
        if(N % p == 0){
            int exp = 0;
            while(N % p == 0){
                exp++;
                N /= p;
            }
            f.push_back(make_pair(p, exp));
        }
    }
    return f;
}
```

Sin embargo, si $\alpha_k = 1$, es decir, el exponente del mayor primo es 1, entonces este no será añadido en el algoritmo anterior, debido a que obtendríamos $N' = p_k$, sin embargo no es posible hallar p_k en el intervalo $[2, \sqrt{N'}]$. Debido a ello hay que hacer una verificación al final. Si después de todo el proceso se cumple que $N \neq 1$, entonces el N resultante es primo, por ende se agregaría N con exponente 1.

```
vector<pair<int, int>> fact(int N){
    vector<pair<int, int>> f;
    if(N == 1){
        f.push_back(make_pair(1, 1));
        return f;
    }
    for(int p = 2; p * p <= N; p++){
        if(N % p == 0){
            int exp = 0;
            while(N % p == 0){
                exp++;
                N /= p;
            }
            f.push_back(make_pair(p, exp));
        }
    }
    if(N != 1) f.push_back(make_pair(N, 1));
    return f;
}
```

§5. Criba de Eratóstenes

Este algoritmo nos permite hallar todos los números primos en el intervalo $[1, N]$. La idea consiste en descartar aquellos números que no pueden ser primos. Sea p un número primo, entonces podemos descartar todos los números X tal que $p \mid X$ y $p < X$, es decir, descartamos todos los múltiplos de p que son mayores que p.

Además, si luego de haber hecho este proceso para todos los primos en el intervalo $[2, X-1]$ el valor X no ha sido descartado, entonces X es primo, puesto que no tendría divisor en el intervalo $[2, X-1]$. Para la implementación se usará un vector booleano que definirá si un número es primo o no.

```cpp
vector<bool> is_prime;
void sieve(int N){
    is_prime.resize(N + 1, true);
    is_prime[0] = is_prime[1] = false;

    for(int p = 2; p <= N; p++){
        if(is_prime[p] == false) continue;

        // Solo se realiza este bucle si p es primo
        // descartando 2p, 3p, 4p, ...
        for(int mult = 2 * p; mult <= N; mult += p){
            is_prime[mult] = false;
        }
    }
}
```

Para determinar la complejidad de este algoritmo hay que notar que por cada primo p el segundo bucle `for` realiza aproximadamente $\frac{n}{p}$ iteraciones, por ende, la cantidad total de iteraciones es:

$$\frac{n}{2}+\frac{n}{3}+\frac{n}{5}+\frac{n}{7}+\ldots=\sum_{\substack{p\leq n\\ p\text{: primo}}}\frac{n}{p}=n\cdot\sum_{\substack{p\leq n\\ p\text{: primo}}}\frac{1}{p}$$

Esta última sumatoria se puede aproximar a $\ln\ln n$, por ende la complejidad del algoritmo es $O(N\log\log N)$.

§5.1. Pequeña optimización

Dado un número primo p, en el algoritmo anterior se mencionó que se iban a descartar todos los múltiplos de p mayores que p, sin embargo hay que notar que los múltiplos $2p, 3p, 4p, \ldots, (p-1)p$ ya han sido descartados por los primos en el intervalo $[2, p-1]$, debido a ellos podemos omitir dichos múltiplos y empezar a partir de p^2, quedando el algoritmo de la siguiente manera.

```cpp
vector<bool> is_prime;
void sieve(int N){
    is_prime.resize(N + 1, true);
    is_prime[0] = is_prime[1] = false;

    for(int p = 2; p <= N; p++){
        if(is_prime[p] == false) continue;

        for(int mult = p * p; mult <= N; mult += p){
            is_prime[mult] = false;
        }
    }
}
```

Observación. Se debe tener cuidado con el producto `p * p` en el código, puesto que puede exceder el límite de una variable `int`, en dicho caso es preferible utilizar variables `long long` adecuadamente.

§6. Criba de Eratóstenes + LP

Veremos ahora una versión más fuerte de cómo identificar si un número es primo o no, en esta versión no usaremos un vector booleano como en la sección anterior sino que usaremos un vector de enteros llamado lp (sin embargo al ser de enteros esto trae consigo más uso de memoria, pero nos permitirá hacer cosas más interesantes).

Definimos $lp[M]$ como el mínimo primo p tal que $p \mid M$. A continuación se muestran los primeros valores del arreglo lp:

M	2	3	4	5	6	7	8	9	10	11	12	13	14
$lp[M]$	2	3	2	5	2	7	2	3	2	11	2	13	2

Podemos notar que $lp[M] = M$ sí y solo sí M es primo así que esto nos ayuda a verificar si un número es primo o no en $O(1)$ como en la sección anterior. Sin embargo, ¿cómo podemos calcular el arreglo lp de modo eficiente?

Para ello hay que notar que cada número M se puede escribir del siguiente modo:

$$M = lp[M] \cdot K \tag{1}$$

Por ejemplo para $M = 175$ se tiene que $lp[M] = 5$, entonces podemos escribir M como:

$$M = lp[M] \cdot 35$$

Otra observación importante es que, como $lp[M]$ es el **mínimo** primo que divide a M entonces cualquier primo p tal que $p \mid K$ (y por ende $p \mid M$) cumple que $p \geq lp[M]$, (en el ejemplo anterior los posible primos p son 5 y 7 que son mayores o iguales a 5), es decir:

$$lp[M] \leq lp[K]$$

Basándonos en lo anterior, en vez de iterar por el valor M podemos iterar por los valores K y luego considerar todos los primos $p \leq lp[K]$, donde p representa todos los posibles valores de $lp[M]$ para algún M.

Por ejemplo, sea $K = 35$ y supongamos que $lp[K]$ ya fue calculado ($lp[K] = 5$), entonces iteraremos por todos los primos $p \leq lp[K]$, obteniendo los siguientes M:

$$p = 2 \;\Rightarrow\; M = 2 \cdot K = 70$$
$$p = 3 \;\Rightarrow\; M = 3 \cdot K = 105$$
$$p = 5 \;\Rightarrow\; M = 5 \cdot K = 175$$

Puede verificar que para dichos M, su representación usando (1) utiliza $K = 35$. Además notemos que justamente p representa el lp del M obtenido, por lo que podemos asignar:

$$lp[M] = p$$

en cada iteración. Cabe resaltar que para la realización eficiente de este algoritmo debes iterar únicamente en los primos p por lo que será necesario almacenar todos los primos en un vector aparte.

La implementación de este algoritmo sería el siguiente:

```cpp
vector<int> lp, primes;

void sieve(int N){
    lp.resize(N + 1);

    for(int K = 2; K <= N; K++){
        // Si lp[K] = 0 entonces K no tiene divisor en el intervalo [2, K-1]
        // por ende K es primo
        if(lp[K] == 0){
            lp[K] = K;
            primes.push_back(K); // Insertamos en el vector de primos
        }
        for(int i = 0; i < primes.size(); i++){
            int p = primes[i];
            if(p > lp[K]) break; // Solo consideramos p <= lp[K]

            int M = p * K;
            if(M > N) break; // M esta fuera del intervalo [1, N]

            lp[M] = p;
        }
    }
}
```

Como se mencionó anteriormente cada valor M tiene una **única** representación usando (1), por ende para cada valor $M \in [2, N]$ se estaría asignando su respectivo lp una única vez, haciendo que la complejidad de este algoritmo sea $O(N)$.

§6.1. Uso de LP al factorizar

Con el arreglo lp no solo podemos determinar si un número es primo o no en $O(1)$, sino que también es posible obtener la factorización de un número M en el intervalo $[1, N]$ de modo más eficiente que en la sección 4.

En dicha sección para factorizar un número M se mencionó que se debía buscar el menor primo $p \mid M$ iterando desde 1 a $\sqrt{M}$ y luego dividir sucesivamente M/p hasta eliminar dicho primo de M, sin embargo usando lp es posible obtener dicho primo p en O(1) (justamente $p = lp[M]$).

De este modo lo único que se necesita saber son los exponentes de estos primos haciendo las divisiones sucesivas. Un ejemplo de implementación sería la siguiente.

```cpp
vector<int> lp, primes; // Ya calculados

vector<pair<int, int>> fact(int M){
    vector<pair<int, int>> f;
    if(M == 1){
        f.push_back(make_pair(1, 1));
        return f;
    }
    while(M > 1){
        int p = lp[M];
        int exp = 0;
        while(M % p == 0){
            exp++;
            M /= p;
        }
        f.push_back(make_pair(p, exp));
    }
    return f;
}
```

Se puede demostrar que la complejidad de la función `fact` es $O(\log M)$ (Intente demostrar esta complejidad).

Esto puede ser muy útil en problemas donde se realicen gran cantidad de consultas pidiendo la factorización de diversos números.

§7. Máximo Común Divisor

Dados dos números enteros no negativos a y b con $a \leq b$ definimos su máximo común divisor (gcd) como el mayor divisor de ambos números y lo denotaremos como $\gcd(a, b)$.

Básicamente lo que se busca es hallar el máximo entero positivo d tal que:

$$d \mid a \quad \& \quad d \mid b$$

Un caso especial es cuando $a = 0$. Como cualquier entero positivo $d \mid 0$ entonces el $gcd(a, b) = b$, por ejemplo, el $gcd(0, 9) = 9$.

Pero, ¿qué sucede cuando ambos valores no son 0? Para este caso vamos a utilizar una de las propiedades mencionadas en la sección de preliminares, la cual es:

$$d \mid a \ \ \& \ \ d \mid b \ \Rightarrow \ d \mid b - a$$

Además podemos restar a repetidamente obteniendo $d \mid b - ak$, para algún k. Notemos que nos conviene minimizar el valor de $b - ak$ pero que no llegue a ser negativo, entonces, ¿cuál sería el menor valor que puede tomar esa expresión? Consideremos el siguiente ejemplo para $a = 7, b = 32$:

$$d \mid b = 32$$
$$d \mid b - a = 25$$
$$d \mid b - 2a = 18$$
$$d \mid b - 3a = 11$$
$$d \mid b - 4a = 4$$

Notemos que 4 es el mínimo valor que puede tomar la expresión $b - ak$ antes de volverse negativo, sin embargo hacer estas iteraciones no es eficiente (por ejemplo cuando $a = 1$). Sin embargo ¿qué pasaría si evaluamos los valores de $b - ak$ en módulo a?

Como $a \equiv 0 \mod a$, entonces se cumple que:

$$b \equiv b - a \equiv b - 2a \equiv \ldots \equiv b - ak \mod a$$

En el ejemplo se puede verificar que todos los números X obtenidos cumplen que $X \equiv 4 \mod 7$, entonces:

$$b - ak \equiv b \mod a$$

Esto quiere decir que el valor mínimo no negativo que puede tomar $b - ak$ es simplemente $(b \mod a)$, además como $(b \mod a)$ es un residuo solo puede tomar los valores $\{0, 1, 2, \ldots, a - 1\}$, por lo que $(b \mod a) < a$, aprovechando esto podemos establecer lo siguiente:

Sea $d_{max} = \gcd(a, b)$, entonces:

$$d_{max} \mid a$$
$$d_{max} \mid b$$
$$d_{max} \mid (b \mod a)$$

Como d_{max} es el máximo que divide a a y $(b \mod a)$, entonces $d_{max} = gcd(b \mod a, a)$. Por lo tanto:

$$\gcd(a, b) = \gcd(b \mod a, a)$$

Nota que el mayor elemento siempre disminuye en cada iteración, y eventualmente se llegará a lo siguiente:

$$\gcd(a, b) = \gcd(b \mod a, a) = \ldots = \gcd(0, d_{max}) = d_{max}$$

Una vez que se llegue al último estado se obtiene el valor del d_{max}. Por ejemplo calcularemos $\gcd(36, 208)$ Realizando lo anterior, obtenemos:

$$\gcd(36, 208) = \gcd(28, 36) = \gcd(8, 28) = \gcd(4, 8) = \gcd(0, 4) = 4$$

Entonces $\gcd(36, 208) = 4$. Este proceso se conoce como **Algoritmo de Euclides** y se puede demostrar que la complejidad del algoritmo es $O(\log \max(a, b))$. A continuación se muestra una posible implementación:

```
int gcd(int a, int b){
    if(a == 0) return b;
    return gcd(b % a, a);
}
```

§7.1. Mínimo Común Múltiplo

Dados dos enteros no negativos a y b, definimos su mínimo común múltiplo (lcm) como el mínimo entero no negativo que es múltiplo de ambos números. Podemos usar el siguiente lema para su cálculo:

Lema 3.

$$ab = \text{lcm}(a, b) \cdot \gcd(a, b)$$

Por ende la implementación sería simplemente lo siguiente:

```
int lcm(int a, int b){
    return a * b / gcd(a, b);
}
```

§7.2. Identidad de Bézout

Un teorema conocido donde se emplea el gcd es el siguiente:

Teorema 1: Identidad de Bézout

Dados dos enteros a y b, existen enteros x, y tal que:

$$ax + by = \gcd(a, b)$$

El objetivo es determinar aquellos enteros x, y que satisfacen la identidad de Bézout para a y b.

Sea $d = \gcd(a, b)$, por el algoritmo de euclides sabemos que:

$$\gcd(a, b) = \gcd(b \mod a, a) = \ldots = \gcd(0, d) = d$$

Entonces, por la identidad de bézout, existen enteros x_i, y_i tal que se cumplen todas las siguientes ecuaciones:

$$\begin{aligned} ax_1 + by_1 &= d \\ (b \mod a)x_2 + ay_2 &= d \\ &\ldots \\ 0x_k + dy_k &= d \end{aligned}$$

Notamos que el último par puede ser $(x_k, y_k) = (0, 1)$. Asumamos que ya tenemos calculado algún (x_i, y_i), veremos cómo calcular (x_{i-1}, y_{i-1}), para facilitar esto calcularemos (x_1, y_1), asumiendo que conocemos el valor de (x_2, y_2):

$$(b \mod a)x_2 + ay_2 = d \tag{2}$$

Notamos que en la ecuación anterior ya utilizamos a, sin embargo falta el término b por lo que debemos reacomodar $(b \mod a)$ de algún modo. Esto se puede escribir como:

$$b \mod a = b - \left\lfloor \frac{b}{a} \right\rfloor \cdot a \tag{3}$$

Donde $\lfloor x \rfloor$ representa el máximo entero menor o igual a x (básicamente x sin decimales). Ejemplo: $\lfloor 3{,}2 \rfloor = 3$, $\lfloor 5{,}9 \rfloor = 5$, $\lfloor 2 \rfloor = 2$.

Nota que la ecuación (3) es básicamente el algoritmo de la división. Reemplazando en (2):

$$\left(b - \left\lfloor \frac{b}{a} \right\rfloor \cdot a\right) x_2 + ay_2 = d$$

$$bx_2 - a \cdot \left(\left\lfloor \frac{b}{a} \right\rfloor x_2\right) + ay_2 = d$$

$$a\left(y_2 - \left\lfloor \frac{b}{a} \right\rfloor x_2\right) + bx_2 = d$$

Una vez que tenemos los valores a y b, comparando con $ax_1 + by_1 = d$, obtenemos:

$$\begin{aligned} x_1 &= y_2 - \left\lfloor \frac{b}{a} \right\rfloor x_2 \\ y_1 &= x_2 \end{aligned}$$

de modo general, teniendo el caso base $(x_k, y_k) = (0, 1)$, podemos establecer la recurrencia en los valores x, y del siguiente modo:

$$\begin{aligned} x_i &= y_{i+1} - \left\lfloor \frac{b_i}{a_i} \right\rfloor x_{i+1} \\ y_i &= x_{i+1} \end{aligned}$$

donde $a_i x_i + b_i y_i = d$. Una posible implementación sería la siguiente:

```cpp
int gcd(int a, int b, int& x, int& y){
    if(a == 0){
        x = 0;
        y = 1;
        return b;
    }
    int x1, y1;
    int d = gcd(b % a, a, x1, y1);
    x = y1 - (b / a) * x1;
    y = x1;
    return d;
}
```

§8. Función φ de euler

Dado un entero positivo N la función $\varphi(N)$ indica la cantidad de enteros entre 1 y N que son **coprimos** con N. Dos números a y b se considera coprimos si se cumple que $\gcd(a,b) = 1$, es decir la función $\varphi(N)$ cuenta cuántos enteros $K, 1 \leq K \leq N$ satisfacen:

$$\gcd(N, K) = 1$$

Por ejemplo, analizando $\varphi(10)$ verificamos que los únicos números entre 1 y 10 que son coprimos con 10 son $\{1, 3, 7, 9\}$, por ende $\varphi(10) = 4$. A continuación se muestran los primeros valores de $\varphi(n)$:

N	1	2	3	4	5	6	7	8	9	10	11	12	13	14
$\varphi(N)$	1	1	2	2	4	2	6	4	6	4	10	4	12	6

Se puede demostrar que $\varphi(n)$ se puede calcular del siguiente modo:

Lema 4.

Sea $N = p_1^{\alpha_1} p_2^{\alpha_2} \dots p_k^{\alpha_k}$ la descomposición en factores primos de dicho entero positivo, entonces:

$$\varphi(N) = N\left(1 - \frac{1}{p_1}\right)\left(1 - \frac{1}{p_2}\right)\cdots\left(1 - \frac{1}{p_k}\right) = N \cdot \prod_{i=1}^{k}\left(1 - \frac{1}{p_i}\right)$$

Para programar el cálculo de esta función vamos a iniciar asumiendo que $\varphi(K)$ son todos los números del 1 al K es decir:

$$\varphi(K) = K$$

Ahora, dado un primo p nota que cada entero N tal que $p \mid N$ considera a p en el cálculo de $\varphi(n)$, en particurlar $\left(1 - \frac{1}{p}\right)$ es un factor de $\varphi(n)$.

Entonces podemos realizar lo siguiente:

```cpp
vector<int> primes; // Precalculado con sieve por ejemplo
vector<int> phi;
void compute_phi(int N){
    phi.resize(N + 1);
    for(int i = 1; i <= N; i++)
        phi[i] = i;

    for(int p : primes){
        for(int mult = p; mult <= N; mult += p){
            phi[mult] = phi[mult] - phi[mult] / p;
        }
    }
}
```

Sin embargo esto trae consigo un precálculo de los primos, lo cual realmente no es necesario. Hay que observar que durante el cálculo todo primo p va a mantener que $\varphi(p) = p$ puesto que ningún otro primo anterior modifica a $\varphi(p)$, por dende podemos establecer esa condición para verificar que p es primo.

```cpp
vector<int> phi;
void compute_phi(int N){
    phi.resize(N + 1);
    for(int i = 1; i <= N; i++)
        phi[i] = i;

    for(int p = 2; p <= n; p++){
        // Verificamos que p sea primo
        if(phi[p] == p){
            for(int mult = p; mult <= N; mult += p){
                phi[mult] = phi[mult] - phi[mult] / p;
            }
        }
    }
}
```

Nota que las iteraciones son similares a las realizadas en la Criba de Eratóstenes descrita en la sección 5, por lo que la complejidad de este algoritmo es $O(N \log \log N)$.

Estas son algunas propiedades interesantes de la función $\varphi(n)$:

Lema 5: Suma de divisores

$$\sum_{d|N} \varphi(d) = N$$

Teorema 2: Teorema de Euler

Sean a y m dos enteros tal que $\gcd(a, m) = 1$, entonces se cumple que:

$$a^{\varphi(m)} = 1 \mod m$$