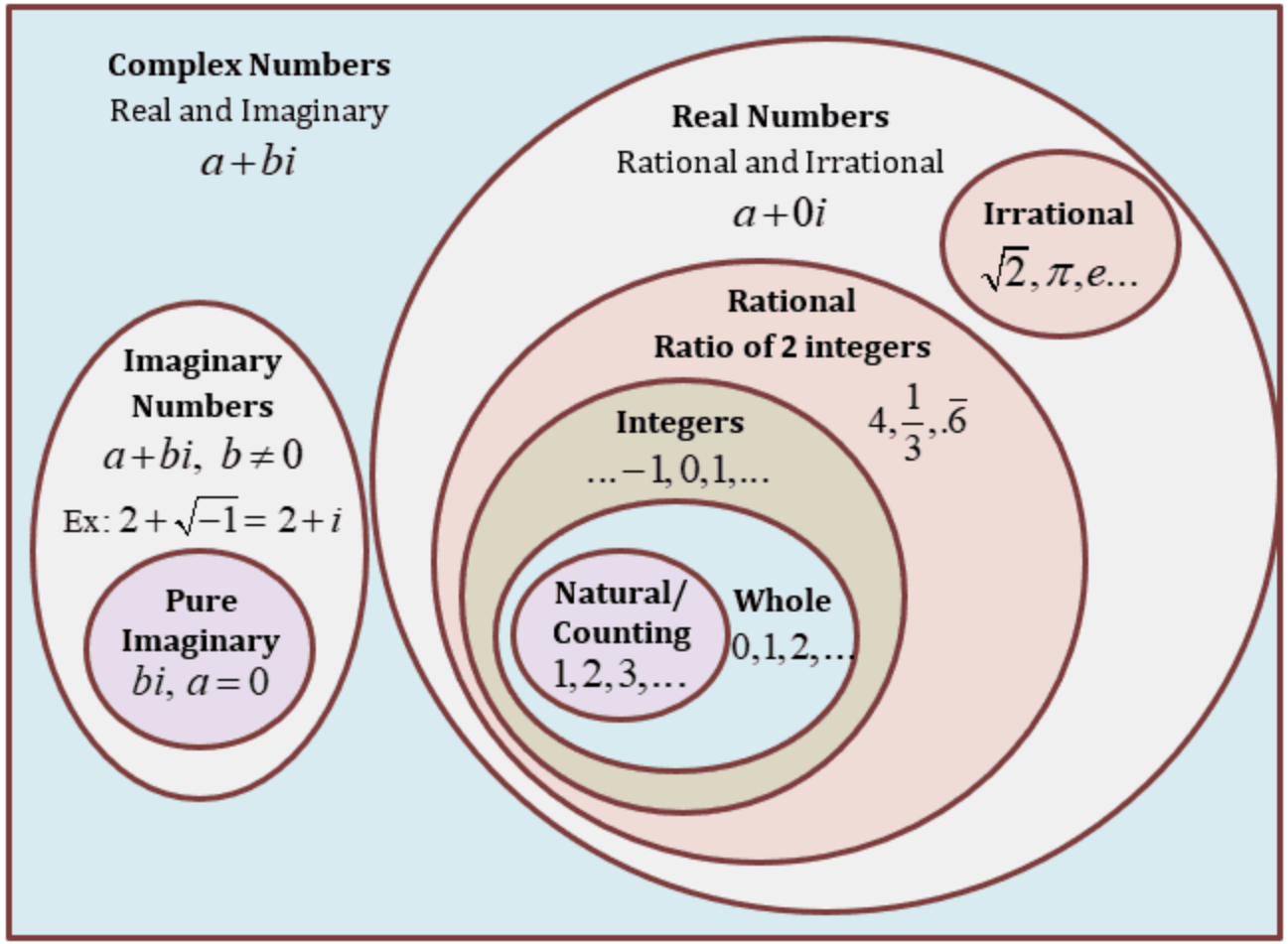
Teoria de numeros

por Ariel Parra

[Γα=Ω5]

¿Qué es la teoría de números?

La teoría de números es una rama de las matemáticas puras que se ocupa de las propiedades y relaciones de los números, particularmente los enteros. Explora la naturaleza fundamental de los números y sus estructuras matemáticas. La teoría de números se ha estudiado durante siglos y tiene profundas conexiones con diversas áreas de las matemáticas, incluidas el álgebra, el análisis y la geometría.



Big Integer

Un BigInteger es una representación de números enteros con precisión **arbitraria**, lo que significa que puede almacenar números tan grandes como lo permita la memoria disponible. A diferencia de los tipos estándar como int o long, que tienen un rango fijo de bits, un BigInteger puede manejar cálculos con números extremadamente grandes sin llegar a desbordarse.

- **En C++:**
  - `int` tiene 32 bits ($\approx \pm 2 \times 10^9$) y `long long` tiene 64 bits ($\approx \pm 9 \times 10^{18}$).
  - Aunque existe `__int128`, no tiene métodos suficientes y es difícil de manejar. Además, las bibliotecas como Boost , que solucionan esto, no se permiten en las competencias ni en plataformas como codeforces.
- **En Java:**
  - `int` es de 32 bits y `long` de 64 bits.
  - Incluye `BigInteger` nativamente dentro de la libreria `java.math.BigInteger`, con multiples métodos para manejarlos.
- **En Python:**
  - Los enteros (`int`) son de precisión arbitraria por defecto.

Big Integer en java

```java
import java.io.*;
import java.math.BigInteger;
public class Main {
    public static void main(String[] args) throws IOException {
        BufferedReader in = new BufferedReader(new InputStreamReader(System.in));
        BufferedOutputStream out = new BufferedOutputStream(System.out);
        String num1 = in.readLine(); String num2 = in.readLine();
        BigInteger bigNum1 = new BigInteger(num1); BigInteger bigNum2 = new BigInteger(num2);
        BigInteger suma = bigNum1.add(bigNum2);
        BigInteger producto = bigNum1.multiply(bigNum2);
        out.write(("Suma: " + suma.toString() + "\n").getBytes());
        out.write(("Producto: " + producto.toString() + "\n").getBytes());
        out.flush();
    }
}
```

Big Integer en python3

```python
import sys
read = sys.stdin.readline
write = sys.stdout.write

def main():
    big_num1 = int(read())
    big_num2 = int(read())

    suma = big_num1 + big_num2
    producto = big_num1 * big_num2

    write(f"Suma: {suma}\n")
    write(f"Producto: {producto}\n")

if __name__ == "__main__":
    main()
```

Criterios de divisibilidad

Un número es divisible por…

- 2: si el último dígito es cero o par.
- 3: si la suma de sus cifras es 3 o múltiplo de 3.
- 4: si sus últimas dos cifras son 00 o múltiplo de 4.
- 5: si el último dígito es 0 o 5.
- 6: si es divisible por 2 y 3.
- 8: si sus últimos 3 dígitos son 000 o múltiplo de 8.
- 9: si sus dígitos son 9 o múltiplo de 9.
- 10: si su último dígito es 0.

Segmented Sieve (criba segmentada)

La **criba segmentada** divide el rango $[0, n-1]$ en segmentos y calcula los números primos en cada segmento, uno por uno. El algoritmo utiliza primero la Criba Simple para encontrar primos menores o iguales a $\sqrt{n}$. Los pasos son:

1. **Criba de eratóstenes regular**: Encuentra todos los primos hasta $\sqrt{n}$ y guárdalos en un vector.
2. **División del rango**: Divide el rango $[0, n-1]$ en segmentos donde el tamaño de cada segmento sea, como máximo, $\sqrt{n}$.
3. **Procesar cada segmento** $[low, high]$:
   - Crea un arreglo `mark[]` de tamaño $(high - low + 1)$, que usa $O(x)$ espacio, donde x es la cantidad de elementos en el segmento.
   - Usa los primos encontrados en el paso 1 para marcar los múltiplos dentro del segmento $[low, high]$.

```cpp
vector<int> segmentedSieve(int n) {
    int limit = floor(sqrt(n)) + 1;
    vector<int> primes; sieve_of_eratosthenes(limit, primes);
    vector<int> all_primes(primes);
    int low = limit, high = 2 * limit;
    while (low < n) {
        if (high >= n) high = n;
        bool mark[limit + 1]; memset(mark, true, sizeof(mark));
        for (int i = 0; i < primes.size(); ++i) {
            int loLim = (low / primes[i]) * primes[i];
            if (loLim < low) loLim += primes[i];
            for (int j = loLim; j < high; j += primes[i]) mark[j - low] = false;
        }
        for (int i = low; i < high; ++i) if (mark[i - low]) all_primes.push_back(i);
        low += limit;
        high += limit;
    }
    return all_primes;
}// Time: O(n * ln(sqrt(n))), Space: O(sqrt(n))
```

Criba de suma de divisores

```
vector<int> sumaDivisores(int n){
    vector<int> criba(n + 1, 1);
    criba[0] = 0;
    for (int i = 2; i <= n; ++i) {
        for (int j = i * 2; j <= n; j += i) {
            criba[j] += i;
        }
    }
    return criba;
}
```

Factorización en números primos.

```
vector<int> factors(int n) {
  vector<int> f;
  for (int i = 2; i*i <= n; ++i) {
    while (n%i == 0) {
        f.push_back(x);
        n /= i;
      }
    }
  if (n > 1) f.push_back(n);
return f;
}
```

Máximo común divisor (GCD) y Mínimo común múltiplo (LCM)

El **algoritmo de Euclides** es la forma más eficiente de calcular el GCD utilizando divisiones sucesivas:

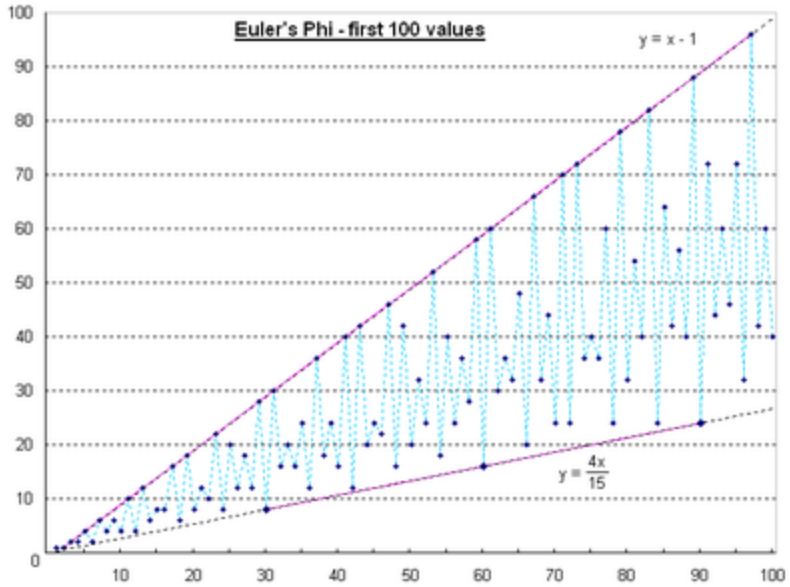
- El GCD de dos números$a$ yb es el mismo que el GCD de b y $a\%b$.
- Este proceso continúa recursivamente hasta que $b = 0$, momento en el cual a es el GCD.

```
int gcd(int a,int b) {
  return b == 0 ? a : gcd(b, a % b);
}//O(log(min(a,b)))
int lcm(int a,int b) {
    return (a / gcd(a, b)) * b; // evitando overflow
}//O(log(min(a,b)))
```

> Si estás utilizando una versión de C++ menor a C++17, puedes usar la función `__gcd` teniendo cuidado con el caso `__gcd(0,0)` y manualmente declarar la funcion de `lcm` si la llegas a necesitar.

Teorema de Euler (Euler's totient function)

La función totiente de Euler $\Phi(n)$ para un número$n$ es la cantidad de números en $1, 2, 3, \ldots, n-1$ que son coprimos con$n$, es decir, los números cuyo MCD (Máximo Común Divisor) con$n$ es 1.



1. Considera cada número 'p' (donde$p$ varía de 2 a $\sqrt{n}$).
   Si p divide n , realiza lo siguiente:
   a) Resta todos los múltiplos de$p$ de 1 an (todos los múltiplos de p tendrán un MCD mayor que 1 con n).
   b) Actualiza n dividiéndolo repetidamente por p.
2. Si el valor reducido de$n$ es mayor que 1, elimina todos los múltiplos de$n$ del resultado.

Código en C++:

```cpp
int phi(int n) {
  int result = n;
  for(int p = 2; p * p <= n; ++p) {
    if (n % p == 0) {
      while (n % p == 0) n /= p;
      result -= result / p;
    }
  }
  if (n > 1) result -= result / n;
  return result;
} // O(Φ n log n)
```

Algebra Modular

El Teorema Pequeño de Fermat (no confundir con el Último Teorema de Fermat) establece que todos los enteros a no divisibles por p cumplen que:

$$a^{p-1} \equiv 1 \pmod{p}$$

En consecuencia,

$$a^{p-2} \cdot a \equiv 1 \pmod{p}$$

Por lo tanto, a^{p-2} es un inverso modular de a módulo p.

Formulas

$$(a+b) \mod m = (a \mod m + b \mod m) \mod m$$

$$(a-b) \mod m = (a \mod m - b \mod m) \mod m$$

$$(a \cdot b) \mod m = ((a \mod m) \cdot (b \mod m)) \mod m$$

$$(a^b) \mod m = (a \mod m)^b \mod m$$

$$\frac{a}{b} \mod m = (a \cdot b^{-1}) \mod m$$

```cpp
long long mod_exp(long long base, long long exp, long long MOD) {
    long long result = 1;
    while (exp > 0) {
        if (exp % 2 == 1) {
            result = (result * base) % MOD;
        }
        base = (base * base) % MOD;
        exp /= 2;
    }
    return result;
}

// Función para calcular (a / b) % MOD
long long divide_mod(long long a, long long b, long long MOD) {
    long long b_inv = mod_exp(b, MOD - 2, MOD); // b^(MOD-2) % MOD
    return (a * b_inv) % MOD;
}
```

Problemas

- **1514C** Product 1 Modulo N
- **1279D** Santa's Bot

Referencias

- Club de Algoritmia CUCEI. (2023). *Number Theory*. Recuperado de https://docs.google.com/presentation/d/1tgwKKvHj8JYzddZDdAPECPIIbE-0GHkmGJcukadLQEk/edit
- dylan007. (2017). *Mathematics in Competitive programming*. Recuperado de https://github.com/dylan007/winter-resources/wiki/Mathematics-in-Competitive-programming
- GeekforGeeks. (2024). *Euler's Totient Function*. Recuperado de https://www.geeksforgeeks.org/eulers-totient-function/
- GeekforGeeks. (2024). *Segmented Sieve*. Recuperado de https://www.geeksforgeeks.org/segmented-sieve/
- kumar, H. (2024). *Number Theory for DSA & Competitive Programming*. Recuperado de https://www.geeksforgeeks.org/number-theory-competitive-programming/
- Laaksonen, A. (2020). *Guide to Competitive Programming*. Recuperado de https://edisciplinas.usp.br/pluginfile.php/7933913/course/section/6549987/Antti Laaksonen - Guide to Competitive Programming_ Learning_Version2.pdf

- Programación Competitiva UFPS. (s.f.). *Matemáticas II*. Recuperado de http://programacioncompetitivaufps.github.io/slides/3-MatematicasII/index.html
- Programación Competitiva UFPS. (s.f.). *Matemáticas*. Recuperado de http://programacioncompetitivaufps.github.io/slides/3-Matematicas/index.html
- Steven, S. & Miguel, A. (2002). *Programming Challengues*. Recuperado de https://i.cs.hku.hk/~provinci/files/b2-programming_challenges.pdf
- strangerr. (2024). *Must do Math for Competitive Programming*. Recuperado de https://www.geeksforgeeks.org/math-in-competitive-programming/
- Yao, D. et al (s.f.). *Divisibility*. Recuperado de https://usaco.guide/gold/divisibility?lang=cpp
- Yao, D. et al (s.f.). *Modular Arithmetic*. Recuperado de https://usaco.guide/gold/modular?lang=cpp