Números primos

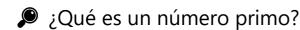
Los números primos han fascinado a la humanidad desde hace más de **2.000 años**. Ya los pitagóricos los llamaban los "ladrillos fundamentales" ## de la aritmética*, porque todos los demás números se pueden construir a partir de ellos multiplicándolos.

En el siglo III a.C., el matemático griego Euclides demostró algo increíble:

• **los números primos son infinitos ∞.** No importa cuántos encontremos, ¡siempre habrá más escondidos en la recta numérica!

Hoy en día, los números primos son la base de la **criptografía moderna** $\widehat{\mathfrak{N}}$. Gracias a ellos podemos **proteger contraseñas, compras online y mensajes privados** $\textcircled{\blacksquare}$.

En programación competitiva aprenderemos a **detectar primos rápidamente** y a usarlos en problemas que combinan lógica, rapidez y creatividad.



Un **número primo** es un número **mayor que 1** que **solo se puede dividir entre 1 y él mismo**. En otras palabras, **no tiene más divisores**.

- Ejemplos de números primos:
 - 2 → solo se puede dividir entre 1 y 2.
 - 3 → solo se puede dividir entre 1 y 3.
 - 5 → solo se puede dividir entre 1 y 5.
 - 7 → solo se puede dividir entre 1 y 7.
- **X** Ejemplos de números que **no son primos** (llamados **compuestos**):
 - 4 → se divide entre 1, 2 y 4.
 - 6 → se divide entre 1, 2, 3 y 6.
 - 9 → se divide entre 1, 3 y 9.

⊗ Nota:

- El **número 1** no se considera primo (porque solo tiene un divisor).
- El 2 es el único primo par. Todos los demás primos son impares.

Primalidad utilizando fuerza bruta

Idea del algoritmo utilizando fuerza bruta

Un número primo es aquel que solo se puede dividir entre 1 y él mismo.

La forma más sencilla de comprobar si un número n es primo es:

- 1. Verificar que n > 1.
- 2. Probar todos los divisores posibles desde 2 hasta n-1.
- 3. Si encontramos un divisor exacto (n % i == 0), entonces n no es primo.

4. Si no encontramos ninguno, entonces n sí es primo.

El Código en Python utilizando fuerza bruta

```
def es_primo(n: int) -> bool:
    Comprueba si un número es primo usando fuerza bruta.
    Argumentos:
        n (int): Número a comprobar.
    Returns:
        bool: True si es primo, False en caso contrario.
    if n <= 1: # Los primos son mayores que 1
       return False
    # Probar todos los divisores desde 2 hasta n-1
    for i in range(2, n):
        if n % i == 0: # Si encontramos un divisor exacto
            return False # No es primo
    return True # Si no encontramos divisores, es primo
# Ejemplos
print(es primo(2)) # True
print(es_primo(15)) # False
print(es_primo(17)) # True
```

👸 Complejidad utilizando fuerza bruta

- En el peor caso, debemos probar todos los números desde 2 hasta n-1.
- Esto implica hacer hasta **O(n)** operaciones.

♣ Primalidad optimizada con raíz cuadrada (√n)

Un número n es primo si no tiene divisores aparte de 1 y él mismo.

En lugar de probar **todos los números hasta** n-1, basta con probar hasta \sqrt{n} , porque:

- Si n tiene un divisor mayor que \sqrt{n} , necesariamente tendrá otro menor que \sqrt{n} .
- Ejemplo: Para n = 36, sus divisores son:
 - Menores o iguales a $\sqrt{36} = 6 \rightarrow \{2, 3, 6\}$
 - Mayores que √36 → {12, 18}

Si encontramos uno pequeño, automáticamente sabemos que existe el grande.

■ Código en Python usando raíz cuadrada (√n)

```
def es_primo(n: int) -> bool:
    """
    Comprueba si un número es primo usando optimización con raíz cuadrada.
    Argumentos:
```

```
n (int): Número a comprobar.
    Returns:
        bool: True si es primo, False en caso contrario.
    if n <= 1: # Los primos son mayores que 1
       return False
    # Solo probamos hasta √n
    limite = int(n**(1/2)) + 1
    for i in range(2, limite):
        if n % i == 0: # Si encontramos un divisor exacto
            return False # No es primo
    return True # Si no encontramos divisores, es primo
# Ejemplos
print(es_primo(2))
                     # True
print(es_primo(15)) # False
print(es_primo(17)) # True
```

Complejidad usando raíz cuadrada (√n)

- Ahora solo probamos divisores hasta √n.
- Esto reduce la complejidad a O(√n).
- Mucho más eficiente que la fuerza bruta O(n), sobre todo para números grandes.

Criba de Eratóstenes

Idea del algoritmo Criba de Eratóstenes

La Criba de Eratóstenes es un método antiguo (inventado por el matemático griego **Eratóstenes**, hace más de 2000 años) para encontrar **todos los números primos menores o iguales que un número N**.

La idea es:

- 1. Escribir todos los números desde 2 hasta N.
- 2. Empezar con el primer número (2) y marcar todos sus múltiplos como compuestos (no primos).
- 3. Pasar al siguiente número no marcado (3) y marcar todos sus múltiplos.
- 4. Repetir el proceso con 5, 7, 11, ... hasta √N.
- 5. Los números que queden sin marcar son **primos**.

Ejemplo con N = 30:

- Quitamos múltiplos de 2 → quedan tachados 4, 6, 8, ...
- Quitamos múltiplos de 3 → tachamos 9, 12, 15, ...
- Quitamos múltiplos de 5 → tachamos 25, 30, ...
- Los que quedan sin tachar → 2, 3, 5, 7, 11, 13, 17, 19, 23, 29.

Complejidad Criba de Eratostenes

- La criba realiza operaciones de tachado sobre múltiplos de primos.
- Su complejidad es aproximadamente O(N log log N), mucho más rápida que comprobar primalidad número por número.

• Además, permite calcular muchos primos a la vez.

Código en Python Criba de Eratostenes

```
def criba_eratostenes(n: int) -> list[int]:
    Encuentra todos los números primos hasta n usando la Criba de Eratóstenes.
    Argumentos:
        n (int): Límite superior.
    Returns:
        list[int]: Lista de primos hasta n.
    # Inicialmente todos son considerados primos
    es_primo = [True] * (n + 1)
    es_primo[0] = es_primo[1] = False # 0 y 1 no son primos
    p = 2
    while p * p <= n:
        if es_primo[p]:
            # Marcar múltiplos de p como no primos
            for i in range(p * p, n + 1, p):
                es primo[i] = False
        p += 1
    # Devolver todos los números que quedaron como primos
    return [i for i, primo in enumerate(es_primo) if primo]
# Ejemplo de uso
print(criba_eratostenes(30))
\# \rightarrow [2, 3, 5, 7, 11, 13, 17, 19, 23, 29]
```

Tabla: estimación de operaciones para distintos N

N (oper., estim.)	Criba ≈ N·log log N	Optimizado ≈ √N (oper.)	Fuerza bruta ≈ N (oper.)	N (límite)
153		10	100	100
22 203		100	10 000	10 000
2 625 792	2 6	1 000	1 000 000	1 000 000
257 023	3 031 2	31 622	1 000 000 000	1 000 000

Problemas: Clase 2 – Números primos