
Universidad de Buenos Aires, Facultad de Ingeniería

Materia: Teoría de Algoritmos (75.29/95.06/TB024)

Curso: Echevarría – 2do Cuatrimestre 2025 Nombre:

Martín Guerrero

Padrón: 107774

# Índice

1. Introducción

2. Supuestos / Limitaciones / Condiciones

3. Diseño y pseudocódigo

4. Análisis de Complejidad Temporal

5. Seguimiento (215 a 225)

6. Tiempo de Ejecución

7. Informe de Resultados

8. Desarrollos Alternativos

9. Conclusión

# Introducción

En este trabajo se nos pidió analizar y mejorar un programa que busca **números amigos** dentro de un rango. El código original funcionaba bien pero tardaba demasiado porque revisaba todos los divisores uno por uno. El objetivo fue optimizarlo para que mantenga la misma función amigos(MAX) y la misma salida, pero con un tiempo de ejecución mucho menor. Además, se nos pidió realizar un breve seguimiento, como calcular la complejidad temporal, utilizar gráficos para medir los nuevos tiempos y realizar un seguimiento entre cierto intervalo, entre otras cosas.

# Supuestos / Limitaciones / Condiciones

- **Dominio**: se buscan pares de números amigos en el intervalo **[1, MAX]** ●

**Lenguaje y versión**: Python **3.10**, sin dependencias externas.

- **Interfaz**: se **mantiene exactamente** el prototipo amigos(MAX) y la **semántica de salida por stdout** (imprime pares y, al final, el tiempo).

- **Definiciones**:
    1. Suma de divisores propios s(n)s(n)s(n): suma de todos los divisores positivos de nnn **estrictamente menores** que nnn.
    2. Amistad: a y b son amigos si s(a)=b, s(b)=a a≠b

- **Limitaciones**:

    1. **Espacio O(MAX)** por el arreglo de sumas de divisores propios.

    2. Para MAX muy grande, el consumo de memoria puede ser el factor limitante (aunque el tiempo escale casi lineal con MAX(log MAX) ).



- **Supuestos:** El algoritmo optimizado, no imprime pares de números que ya se imprimieron de manera inversa. Por ejemplo, si ya se imprimió el (220,284), no se va a imprimir el par (284,220). Después el comportamiento se mantuvo de manera normal, al presentado en el enunciado, mejorando el tiempo de ejecución.

# Diseño y pseudocódigo

En lugar de calcular los divisores de cada número uno por uno (fuerza bruta), usamos una lista que va acumulando la suma de divisores propios. Para cada posible divisor d, lo sumamos en todas las posiciones de la lista que corresponden a sus múltiplos. De esa forma, en una sola pasada obtenemos s(n) para todos los números hasta MAX.

```
func amigos(MAX):
    iniciar cronómetro
    si MAX < 2:
        imprimir 0.0 y salir
    suma_divisores_propios = arreglo de tamaño MAX+1 con ceros

    para d desde 1 hasta MAX//2:
        para m desde 2*d hasta MAX paso d:
            suma_divisores_propios [m] += d
```

```
para i desde 2 hasta MAX:
    s = suma_divisores_propios [i]
    si s > i y s <= MAX y suma_divisores_propios [s] == i:
        imprimir (i, s)

imprimir tiempo de ejecución
```

# Análisis de Complejidad Temporal

El algoritmo optimizado se divide en dos etapas principales:



- **Cálculo de la suma de divisores propios (con la lista suma_divisores_propios):**
    1. Se recorre cada posible divisor d desde 1 hasta MAX//2.
    2. Para cada d, se actualizan todos sus múltiplos m = 2d, 3d, … ≤ MAX.
    3. Esto significa que d se suma aproximadamente MAX/d veces.
    4. El costo total de esta etapa es O(MAX* log (MAX))

- Chequeo de pares amigos:
    1. Se recorre la lista desde i = 2 hasta MAX.
    2. En cada paso se hacen un par de comparaciones y una asignación.
    3. Esto cuesta O(MAX).

**Complejidad Total** = O(MAX* log (MAX)) + O(MAX) = **O(MAX* log (MAX))**

# Seguimiento (215 a 225)

Para cada número n en el rango [215, 225], se muestra s(n) (la suma de sus divisores propios) y si forma un par amigo.

| n | s(n) | Forma Par |
|---|---|---|
| 215 | 49 | No (s(49)=8) |
| 216 | 384 | No (s(384)=636) |
| 217 | 39 | No (s(39)=17) |
| 218 | 112 | No (s(112)=136) |
| 219 | 77 | No (s(77)=19) |
| 220 | 284 | Sí, con 284 (fuera del rango 215–225) porque s(284)=220 |



| | | |
|---|---|---|
| 221 | 31 | No (s(31)=1) |
| 222 | 234 | No (s(234)=312) |
| 223 | 1 | No (s(1)=0) |
| 224 | 280 | No (s(280)=440) |
| 225 | 178 | No (s(178)=92) |

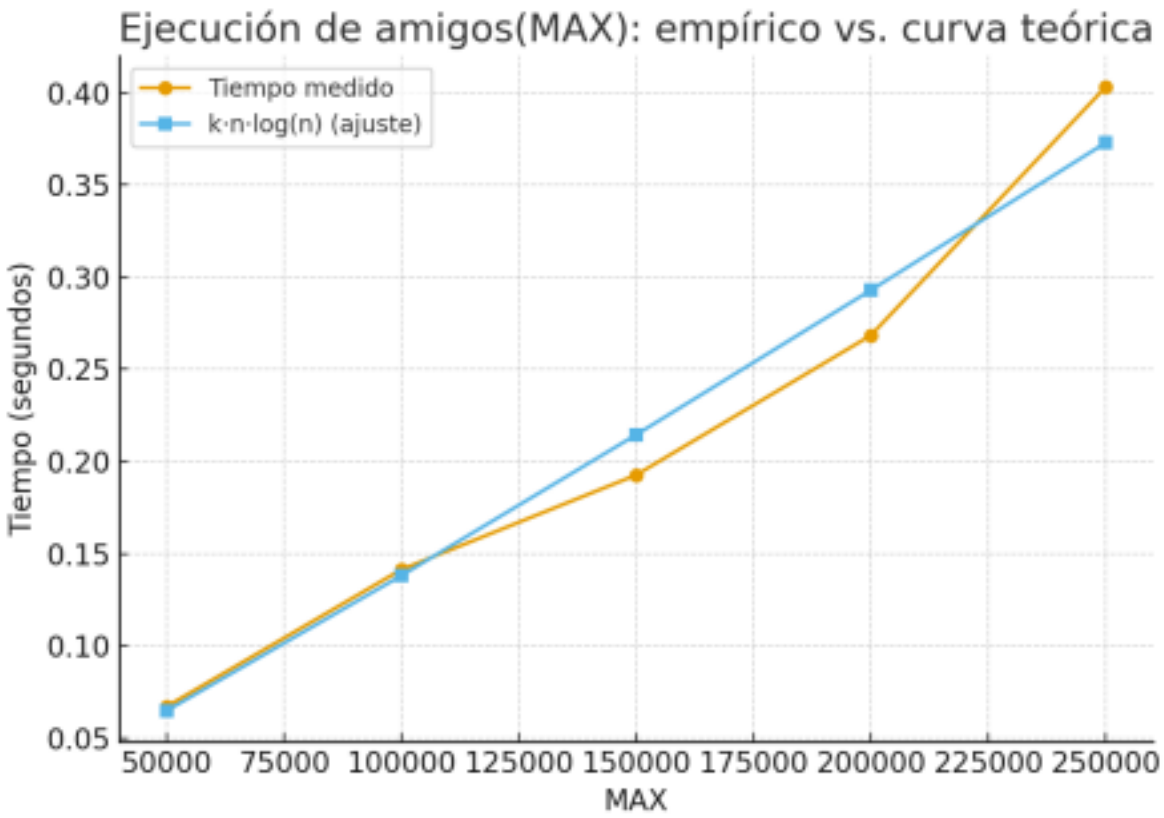
# Tiempo de Ejecución


Los tiempos de ejecución muestran un crecimiento acorde a la complejidad O(MAX·log(MAX)). Incluso para MAX = 250.000 el cálculo tarda menos de medio



segundo, lo que confirma la eficiencia del algoritmo optimizado en comparación con la versión original que era cuadrática.

| MAX | Tiempo (s) |
|---|---|
| 50.000 | 0.0672 |
| 100.000 | 0.1414 |
| 150.000 | 0.1972 |
| 200.000 | 0.2684 |
| 250.000 | 0.4030 |






El ajuste k.n.log(n) (línea teórica) sigue de cerca los tiempos medidos, confirmando la complejidad O(MAX.log(MAX)) del algoritmo.

# Informe de Resultados

Los tiempos obtenidos confirman lo esperado teóricamente: el crecimiento es acorde a O(MAX log MAX), muy por debajo del algoritmo original O(MAX²). Con la versión optimizada, calcular hasta 250.000 se resuelve en menos de medio segundo, mientras que el código inicial tardaba varios minutos en llegar a 100.000.

# Desarrollos Alternativos

**Segmentación:** dividir el rango en bloques más pequeños para mejorar el uso de la memoria caché o reducir el consumo de memoria.

**Paralelización:** dividir el rango en bloques y procesarlos en paralelo. (Modelo Fork-Join)

# Conclusión

Este trabajo permitió ver cómo un mismo problema puede resolverse de distintas maneras, y cómo una mejora en el enfoque cambia por completo los tiempos de ejecución. Con un pequeño cambio en la forma de calcular los divisores, pasamos de un programa que tardaba minutos en obtener resultados a otro que lo hace en menos de un segundo para el mismo rango. Además, al medir y



comparar con la teoría, confirmamos que la complejidad O(MAX·log(MAX)) describe muy bien el comportamiento del algoritmo.