# Trabajo Práctico N°0

**T**eoría de Algoritmos [TB024]

**Catedra:**

- CURSO: 03 - Echevarria

**Alumno:**

- 108091, Martín Morilla

**Fecha de entrega:** 31 de Agosto de 2025

# Índice

# 1. Introducción

Los *números amigos* son pares de enteros positivos $a$ y $b$ tales que la suma de los divisores propios de $a$ (todos sus divisores excepto $a$ mismo) es igual a $b$, y recíprocamente la suma de los divisores propios de $b$ es igual a $a$. Un ejemplo clásico es el par 220 y 284: los divisores propios de 220 suman 284 y los divisores propios de 284 suman 220.

El trabajo parte de una implementación base provista por la cátedra que expone la función `amigos(MAX)`. Dicha implementación, aunque correcta, presenta un desempeño inadecuado para valores grandes de `MAX` (por ejemplo, tardanzas del orden de varios minutos para `MAX = 100000`). El objetivo de este trabajo práctico es mejorar la eficiencia del programa manteniendo el prototipo de `amigos(MAX)`.

A lo largo del informe se describen los supuestos y limitaciones, el diseño del algoritmo optimizado y las estructuras de datos utilizadas, junto con un análisis de complejidad temporal. Se incluye un seguimiento de ejecución detallado para los números del 215 al 225, mediciones temporales y gráficos para los rangos solicitados (1–50 000; 1–100 000; 1–150 000; 1–200 000; 1–250 000), y una discusión comparativa entre los resultados empíricos y las predicciones teóricas. Finalmente, se exponen conclusiones y alternativas de optimización adicionales.

# 2. Metodología

## 2.1. Supuestos, limitaciones y condiciones

El algoritmo implementado se basa en los siguientes supuestos y restricciones, con algunas diferencias con el comportamiento del código original provisto por la cátedra:

- **Dominio de entrada:** La función `amigos(MAX)` recibe un número entero $MAX > 0$. No se contemplan valores negativos de $MAX$.
- **Rango de búsqueda:** Se evalúan candidatos $i$ en el rango $[2, MAX)$, es decir, hasta $MAX - 1$. El par especial $(0, 0)$ se incluye explícitamente.
- **Números perfectos:** Se consideran válidos como pares amigos del tipo $(n, n)$, dado que la suma de sus divisores propios coincide con $n$.
- **Suma de divisores superior a $MAX$:** Si un número $i < MAX$ tiene como suma de divisores un valor $s \geq MAX$, nunca se llegaria a calcular los divisores de $s$ si $s \geq MAX$, para respetar la restricción de no procesar números superiores a $MAX$.
- **Duplicados:** No se reportan pares duplicados. Por convención, se imprime el par $(i, s)$ únicamente cuando $i < s$, evitando repeticiones como $(220, 284)$ y $(284, 220)$.

## 2.2. Pseudocódigo y estructuras de datos utilizadas

El algoritmo utiliza principalmente:

- **Diccionario 'cache'**: almacena la suma de divisores de cada número [2,MAX) para acceso rápido.
- **Lista 'prints'**: guarda los pares de números amigos encontrados.

**Algorithm 1** Búsqueda de números amigos (impresión directa)

```
1: procedure AMIGOS(MAX)
2:     cache ← precalcular sumas de divisores de 2 a MAX-1
3:     for i = 2 to MAX-1 do
4:         posible_amigo ← cache[i]
5:         if posible_amigo = i then
6:             print (i, i)
7:         else if cache[posible_amigo] == i then
8:             if i < posible_amigo then                ▷ evitar duplicados
9:                 print (i, posible_amigo)
10:            end if
11:        end if
12:    end for
13: end procedure
```

**SumaDeDivisores(n)** Se calcula iterando hasta $\sqrt{n}$, sumando divisores y sus complementarios.

## 2.3. Análisis de complejidad

### 2.3.1. Algoritmo base

Implementa una búsqueda de *números amigos* mediante un enfoque de fuerza bruta. Para cada número $i$ entre 0 y $MAX - 1$, realiza los siguientes pasos:

1. Calcula la suma de los divisores propios de $i$, denotada como $s$.
2. Calcula la suma de los divisores propios de $s$, denotada como $s2$.
3. Verifica si $i = s2$; en caso afirmativo, imprime el par $(i, s)$ como números amigos.

**Análisis temporal.**

- El primer bucle interno itera desde 1 hasta $i - 1$, realizando operaciones constantes en cada paso. Por lo tanto, su complejidad es $O(i)$.
- El segundo bucle interno itera desde 1 hasta $s - 1$. Considerando que $s$ puede crecer hasta $O(i)$ en el peor caso, este bucle también tiene complejidad $O(i)$.
- Por lo tanto, para un número $i$ dado, el tiempo total es $O(i) + O(i) = O(i)$.
- Sumando para todos los números hasta $MAX - 1$, la complejidad temporal total es:

$$\sum_{i=1}^{MAX-1} O(i) = O(MAX^2)$$

### 2.3.2. Implementación final

Este algoritmo mejora la eficiencia mediante el almacenamiento previo de la suma de divisores de cada número en un diccionario. La verificación de pares de números amigos se realiza utilizando estas sumas precalculadas, evitando cálculos repetidos.

**Análisis temporal.**

- Cálculo de la suma de divisores para todos los números hasta $MAX - 1$: cada número requiere $O(\sqrt{n})$, por lo que el tiempo total es

$$\sum_{n=2}^{MAX-1} O(\sqrt{n}) \approx O(MAX^{3/2})$$

- Verificación de números amigos utilizando el diccionario: $O(MAX)$
- Complejidad total: $O(MAX^{3/2}) + O(MAX) = O(MAX^{3/2})$

# 3. Resultados

## 3.1. Seguimiento del 215 al 225

Para ilustrar el funcionamiento del algoritmo, se realiza un seguimiento paso a paso para los números del 215 al 225 (se supone que en este caso **MAX = 226**). Se calcula primero la suma de divisores de cada número y luego se verifica si forman pares de números amigos.

**Cálculo de la suma de divisores:**

| Número $n$ | Divisores propios | Suma de divisores (cache[$n$]) |
|---|---|---|
| 215 | 1, 5, 43 | 49 |
| 216 | 1, 2, 3, 4, 6, 8, 9, 12, 18, 24, 27, 36, 54, 72, 108 | 304 |
| 217 | 1, 7, 31 | 39 |
| 218 | 1, 2, 109 | 112 |
| 219 | 1, 3, 73 | 77 |
| 220 | 1, 2, 4, 5, 10, 11, 20, 22, 44, 55, 110 | 284 |
| 221 | 1, 13, 17 | 31 |
| 222 | 1, 2, 3, 6, 37, 74, 111 | 234 |
| 223 | 1 | 1 |
| 224 | 1, 2, 4, 7, 8, 14, 16, 28, 32, 56, 112 | 280 |
| 225 | 1, 3, 5, 9, 15, 25, 45, 75 | 178 |

**Verificación de números amigos:** Se revisa cada número $i$ hasta [2, MAX) y se obtiene su posible amigo $cache[i]$. Si $cache[posible\_amigo] == i$, se considera un par de números amigos:

| Número $i$ | posible_amigo | cache[posible_amigo] | ¿Par amigo? |
|---|---|---|---|
| 215 | 49 | 8 | No |
| 216 | 304 | ? | No |
| 217 | 39 | 17 | No |
| 218 | 112 | 136 | No |
| 219 | 77 | 19 | No |
| 220 | 284 | ? | No |
| 221 | 31 | 1 | No |
| 222 | 234 | ? | No |
| 223 | 1 | 0 | No |
| 224 | 266 | ? | No |
| 225 | 178 | 92 | No |

**Resultado:** En este rango entonces no hay par de números amigos, ya que como se puede ver, nuestro algoritmo no calcula la suma de los divisores de números $\geq MAX$. Es por eso que la pareja (220,284) sí se tiene como MAX = 226, no se imprimirá como números amigos. Para que esa pareja se considerase amiga, MAX debería ser igual a ***285***.

Este seguimiento permite observar cómo el algoritmo utiliza el diccionario para evitar recalcular sumas de divisores y cómo se identifican los pares de números amigos.

## 3.2. Tiempos de ejecución

### 3.2.1. Algoritmo base


Figura 1: Algoritmo base

### 3.2.2. Implementación final

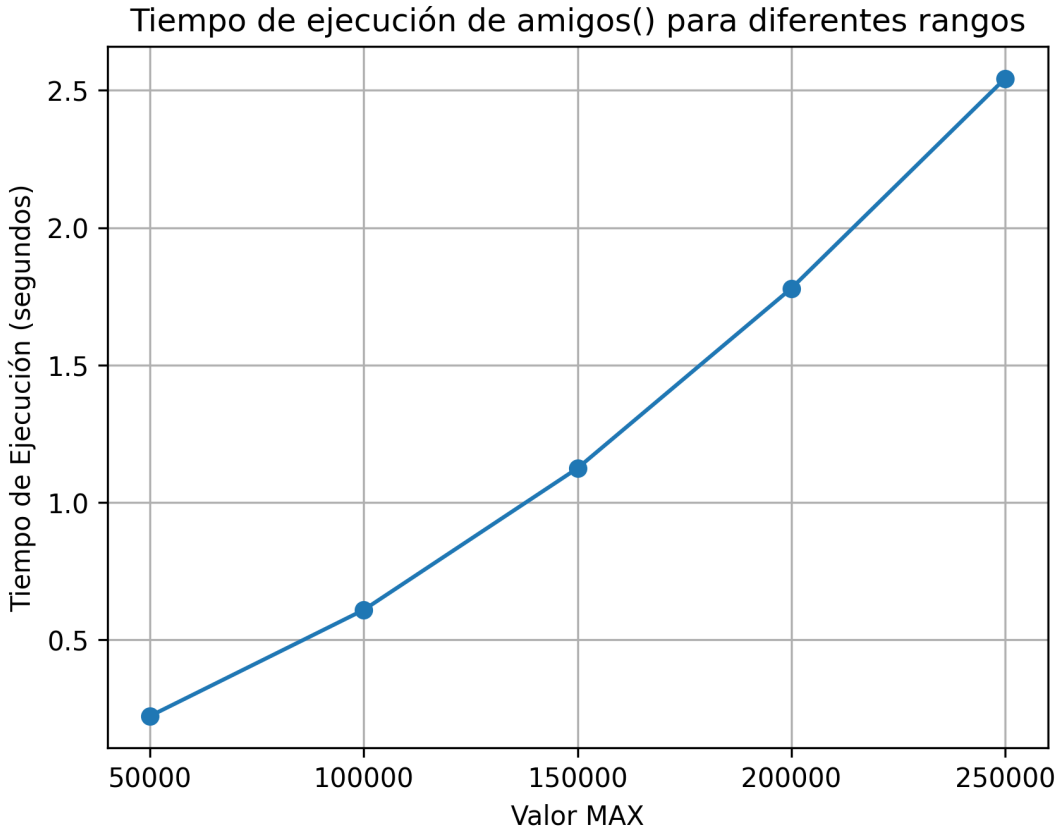


Figura 2: Mejora del algoritmo base

## 3.3. Discusión y alternativas

Durante el desarrollo del algoritmo para encontrar números amigos, se optó por la implementación final presentada anteriormente, basada en el precálculo de sumas de divisores mediante un diccionario. Inicialmente, esta fue la estrategia más intuitiva a mi parecer. Sin embargo, revisando el algoritmo uno comienza a pensar "¿no habrá alguna forma en la que ***no*** se tengan que calcular la suma de los divisores de ***todos*** los números hasta MAX?".
Por lo que, al buscar alternativas para optimizar la eficiencia, se encontró una implementación basada en un enfoque tipo *sieve*, descrita en un blog de Codeforces [1]. Esta alternativa es considerablemente más eficiente que la implementación final debido a que:

- Precalcula las sumas de divisores de todos los números hasta el límite en tiempo lineal respecto al número de actualizaciones, evitando iteraciones repetidas sobre divisores individuales.
- Utiliza un enfoque acumulativo similar al de la criba de Eratóstenes, donde cada divisor contribuye directamente a los múltiplos correspondientes, logrando una complejidad teórica cercana a $O(MAX \log MAX)$, mucho mejor que la complejidad $O(MAX^{3/2})$ de la implementación final.

- Minimiza las operaciones redundantes y reduce significativamente el tiempo de ejecución para valores grandes de $MAX$.

Aunque esta alternativa no fue implementada en el presente trabajo, constituye una referencia útil para futuras optimizaciones del algoritmo.


Figura 3: Algoritmo tipo Sieve.

# 4. Conclusiones

En este trabajo se compararon dos algoritmos para la identificación de números amigos. El primer algoritmo consistía en un enfoque de fuerza bruta, mientras que la implementación final mejoró significativamente la eficiencia mediante el precálculo de sumas de divisores utilizando un diccionario y evitando cálculos repetidos.

**Resumen de mejoras logradas** La comparación de los tiempos de ejecución se presenta en la sección (Tiempos de ejecución).

Como puede observarse claramente en los gráficos, la implementación final logra una reducción considerable en los tiempos de ejecución, demostrando la ventaja práctica de optimizar el algoritmo.

**Validación experimental vs teórica.** Analizando las complejidades teóricas de ambos algoritmos, $O(MAX^2)$ para el algoritmo base y $O(MAX^{3/2})$ para la implementación final, se puede observar una correspondencia aproximada con las curvas de los gráficos. La similitud no es exacta debido a factores como constantes de ejecución, overhead del lenguaje y acceso a memoria, pero la tendencia coincide con lo esperado.

**Importancia del análisis algorítmico.** Destacamos la relevancia de analizar la complejidad de los algoritmos. En áreas donde el tiempo de ejecución es crítico, obtener una mejor complejidad puede marcar una diferencia significativa en la práctica. Por otro lado, en contextos donde los datos son limitados o los tiempos de ejecución son tolerables, un algoritmo con mayor complejidad puede ser suficiente. Comprender las implicancias teóricas y empíricas permite tomar decisiones fundamentadas sobre la implementación más adecuada para cada caso.

# Referencias

[1] Hikari9. (2015, December 22). *Finding amicable numbers using a sieve approach*. Codeforces Blog. https://codeforces.com/blog/entry/22229