

1. Explorador de Complejidades

Complejidad	Analogía	Código	Situación real
O(1) Constante	Presionar el botón de un elevador para ir a un piso en específico	obtenerElemento(lista, índice): return lista[índice]	Acceder directamente a un elemento de un arreglo mediante un índice conocido
O(log n) Logarítmico	Buscar una palabra en un diccionario abriéndolo por la mitad cada vez	contarDivisiones(n): contador = 0 mientras n > 1: n = n / 2 contador = contador + 1 return contador	Búsqueda binaria
O(n) Lineal	Revisar uno por uno los asientos de un cine para encontrar a una persona	buscar(lista, objetivo): para cada elemento en lista: si elemento == objetivo: return encontrado	Buscar un elemento en un arreglo, comprobando cada uno de sus elementos
O(n log n) Linealítmica	Para ordenar un mazo de cartas lo divides en partes, ordenas cada parte y luego las combinas	mergeSort(lista): si tamaño ≤ 1: return dividir lista mergeSort(izq) mergeSort(der) combinar	Ordenamiento de elementos eficiente
O(n ²) Cuadrática	En una reunión, cada persona saluda a todas las demás	compararTodos(lista): para i en lista: para j en lista: comparar(i, j)	Comparaciones de pares y algoritmos mal diseñados
O(2 ⁿ) Exponencial	Estoy organizando una reunión, y por cada persona que conozco tengo la posibilidad de invitarla o de no invitarla	invitar(personas, i): si i == tamaño(personas): mostrar invitación return invitar(personas, i + 1) // invitar invitar(personas, i + 1) // no invitar	No es deseado, pero se puede encontrar en algoritmos de generación de subconjuntos

¿Cuál fue la analogía que más te ayudó a entender y por qué?

La analogía de la complejidad constante (presionar un botón determinado de un elevador), porque es muy sencilla de entender y se relaciona perfectamente con mis experiencias de la vida real. Además, puede adaptarse a otras complejidades fácilmente.

2. El caso de DataStream Inc.

Diagnóstico

Claramente se presenta un problema en el algoritmo creado por el desarrollador, no en el hardware. La cantidad de transacciones creció 50 veces, pero el tiempo que toma procesarlas creció 564 veces, lo que nos permite observar que el aumento no fue proporcional. Podemos descartar que el algoritmo sea lineal, pues para que sea así el crecimiento del tiempo debería ser similar al incremento de las transacciones.

Cálculos

$$\frac{500,000}{10,000} = 50 \text{ /// } 47m * 60s = 2820s \text{ /// } \frac{2820s}{5s} = 564s$$

¿Qué complejidad deberían buscar y por qué?

$O(n)$, porque permite escalar linealmente el volumen

Checkpoint metacognitivo - Fase COMPRENDE

¿Puedo explicar con mis propias palabras qué mide Big O sin mencionar "tiempo de ejecución en segundos"?

Predice los resultados de un algoritmo en el peor caso posible según va creciendo o escalando.

Si alguien me dice "mi algoritmo es $O(n^2)$ ", ¿qué imagen mental se forma?

¿Puedo predecir su comportamiento?

Un tiempo de ejecución que crece exponencial y desproporcionadamente. Sí es posible predecir su comportamiento, tan solo debes calcular el cuadrado de la cantidad de operaciones correspondiente.

¿Qué preguntas formulé a la IA que no había considerado antes de comenzar?

Que me explicara a detalle algunos conceptos y que me dijera como se leían ciertas notaciones matemáticas

¿En qué momento sentí que "esto tiene sentido" y qué lo provocó?

Las analogías me ayudaron bastante a diferenciar las distintas complejidades.

3. Analizador de código

Fragmento 1

Fragmento 1

pseudocode

Copiar código

```
funcion obtenerPrimero(arreglo):  
  si tamaño(arreglo) == 0:  
    retornar null  
  retornar arreglo[0]
```

👉 Pregunta:

¿Cuál crees que es la complejidad temporal de este fragmento?
Explicame *por qué*, no solo el Big-O.

📄

👍

🗨

📤

🔄

⋮

Es lineal porque solo utiliza un bucle simple que recorre todos los elementos de un arreglo una sola vez

Bien que intentes justificarlo 👍, pero revisemos el código **con lupa** paso a paso.

Te hago algunas preguntas para que tú mismo lo confirmes:

1. 🔍 ¿Ves algún bucle (`for`, `while`) en el código?

Léelo línea por línea.

2. 📏 ¿Alguna instrucción depende del tamaño del arreglo recorriendo sus elementos?

Por ejemplo: acceder a *todos* los elementos vs acceder a *uno solo*.

3. 🧠 ¿Qué significa `arreglo[0]` en términos de operaciones?

- ¿Accede a un elemento específico?
- ¿O necesita mirar todo el arreglo antes?

4. 🧐 Imagina dos casos:

- Un arreglo con 1 elemento
- Un arreglo con 1 millón de elementos

👉 ¿El número de operaciones cambia de forma proporcional al tamaño?

Respóndeme estas preguntas y, con eso, dime de nuevo cuál crees que es la complejidad y por qué.

📄

👍

🗨

📤

🔄

⋮

Es constante, porque no utiliza bucles y accede directamente a un solo elemento del arreglo

↓

¡Exacto! 🎯 Muv bien corregido.

¡Exacto! 🎯 Muy bien corregido.

Validemos tu razonamiento paso a paso:

1. ✅ **No hay bucles**

El código no usa `for`, `while` ni llamadas recursivas.

2. ✅ **Acceso directo**

`arreglo[0]` es una operación de tiempo constante en arreglos (acceso por índice).

3. ✅ **Independiente de `n`**

Da igual si el arreglo tiene 1, 10 o 1 millón de elementos:
el número de operaciones **no cambia**.

🔴 **Conclusión:**

La complejidad temporal es **$O(1)$** porque el tiempo de ejecución no depende del tamaño del input.

Muy bien 🍌 — eso es justo el tipo de razonamiento que buscan en exámenes.

Fragmento 2

Genial 😊 seguimos.

Fragmento 2

pseudocode

📋 Copiar código

```
funcion imprimirTodos(arreglo):  
  para i desde 0 hasta tamaño(arreglo) - 1:  
    imprimir(arreglo[i])
```

👉 **Pregunta:**

¿Cuál crees que es la complejidad temporal de este fragmento?

Explicámelo como si se lo tuvieras que justificar a un profe exigente 😊

📄 🍌 🗨️ ⬆️ ↺ ⋮

Es lineal, solo utiliza un bucle que imprime todos los elementos de un arreglo una sola vez

¡Muy bien! ✅ Esa explicación está sólida, pero vamos a **afinarla como en un examen**.

Validemos tu razonamiento paso a paso:

1. 🔄 Número de iteraciones

El bucle va desde `0` hasta `tamaño(arreglo) - 1`, es decir, se ejecuta **n veces** si el arreglo tiene **n** elementos.

2. 📖 Trabajo por iteración

Dentro del bucle solo hay una operación constante:

`imprimir(arreglo[i])` → **O(1)**.

3. ✅ Costo total

- n iteraciones
- cada una cuesta O(1)

👉 $n \times O(1) = O(n)$

🔴 Conclusión correcta:

La complejidad temporal es **O(n)**.

Perfecto. No hay nada engañoso aquí y lo explicaste bien 👍

Fragmento 3

Fragmento 3

pseudocode

📋 Copiar código

```
funcion contarPares(arreglo):  
  contador = 0  
  para i desde 0 hasta tamaño(arreglo) - 1:  
    para j desde 0 hasta tamaño(arreglo) - 1:  
      si (arreglo[i] + arreglo[j]) es par:  
        contador = contador + 1  
  retornar contador
```

👉 Pregunta:

¿Cuál crees que es la complejidad temporal de este fragmento?

No te apresures: dime cómo cuentas las iteraciones y por qué.

📄 📌 🗨️ ⬆️ ↺️ ⋮

Es cuadrática, porque hay bucles anidados. En el bucle interior se compara cada uno de los elementos con todos los demás

Paso a paso

1. 🔄 **Bucle externo (i)**

Se ejecuta `n` veces, donde `n = tamaño(arreglo)`.

2. 🔄 **Bucle interno (j)**

Para *cada* valor de `i`, el bucle interno también se ejecuta `n` veces.

3. 🚩 **Trabajo interno**

La operación:

pseudocode

📋 Copiar código

```
si (arreglo[i] + arreglo[j]) es par
```

es $O(1)$.

4. 📋 **Costo total**

- Bucle externo: `n`
- Bucle interno: `n`
- Operación constante dentro

👉 $n \times n \times O(1) = O(n^2)$

📌 **Conclusión:**

La complejidad temporal es $O(n^2)$.

Ojo a algo importante (bien visto por ti):

no es solo “porque hay bucles anidados”, sino **porque ambos dependen de `n`**. Eso es clave.

Fragmento 4

Fragmento 4

pseudocode

📋 Copiar código

```
funcion buscarBinario(n):  
  contador = 0  
  mientras n > 1:  
    n = n / 2  
    contador = contador + 1  
  retornar contador
```

👉 **Pregunta:**

¿Cuál crees que es la complejidad temporal de este fragmento?

Intenta explicarlo en términos de **cuántas veces se ejecuta el `while`** y qué le pasa a `n` en cada iteración.

📋 👍 🗨️ 🔗 🔄 ...

Es logarítmica, esto porque con cada operación se divide el tamaño de `n` a la mitad

¡Exacto! 🎯 Esa es la **idea clave** y la expresaste muy bien.

Vamos a validarlo paso a paso para que quede blindado:

¡Exacto! 🎯 Esa es la **idea clave** y la expresaste muy bien.


Vamos a validarlo paso a paso para que quede blindado:

1. 🤖 **Condición del `while`**

El bucle continúa mientras `n > 1`.

2. ✂️ **Qué pasa con `n` en cada iteración**

pseudocode

 Copiar código

```
n = n / 2
```

En cada vuelta, `n` se reduce a la mitad.

3. 🧠 **Pregunta clave**

¿Cuántas veces puedes dividir `n` entre 2 hasta que llegue a 1?

👉 Eso es exactamente $\log_2(n)$ veces.

4. 📝 **Trabajo por iteración**

Todo lo que ocurre dentro del bucle es $O(1)$.

🔴 **Conclusión correcta:**

La complejidad temporal es $O(\log n)$.

Muy bien. Este patrón aparece muchísimo (búsqueda binaria, heaps, divide & conquer).

Fragmento 5

Fragmento 5

ps

 Copiar código

```
funcion imprimirPares(arreglo):  
  para i desde 0 hasta tamaño(arreglo) - 1:  
    para j desde 0 hasta i:  
      imprimir(arreglo[i])
```

👉 **Pregunta:**

A primera vista parece costoso...

¿Cuál crees que es la complejidad temporal *real* de este fragmento?

Explicámelo contando **cuántas veces se ejecuta `imprimir` en total**, no solo mirando los bucles.


Es lineal, porque el segundo bucle no utiliza a `n` como límite, sino a `i`. La operación imprimir solo se ejecuta hasta llegar al valor actual de `i`

Muy buena intuición 🧠 y vas por el camino correcto, pero vamos a **apretarla un poco más** para que sea totalmente rigurosa.

Te guío con preguntas (modo socrático 🧠):

1. 🖨️ Para un arreglo de tamaño `n`:
 - Cuando `i = 0`, ¿cuántas veces se ejecuta el `imprimir`?
 - Cuando `i = 1`?
 - Cuando `i = 2`?
 - ...
 - Cuando `i = n - 1`?

2. 🇲🇳 Si sumas todas esas ejecuciones:

 Copiar código

```
1 + 2 + 3 + ... + n
```





👉 ¿A qué expresión matemática equivale esa suma?

3. 🗒️ Ahora la clave:
 - ¿Esa expresión crece de forma proporcional a `n`?
 - ¿O crece proporcional a `n²`?

4. ⚠️ Pregunta crítica de examen:

¿El hecho de que el límite del bucle interno sea `i` automáticamente lo hace $O(n)$?

Respóndeme estas preguntas y dime si mantienes que es $O(n)$ o si quieres ajustar tu respuesta.

     ...

siempre y cuando el segundo bucle no tenga su condición de límite en `n`, la complejidad no será cuadrática. Como no hay divisiones, no es logarítmica. Por descarte tiene que ser lineal

Buen intento 👍, y me gusta que estés razonando... pero aquí hay **una trampa clásica** que vale oro detectar ahora (muchos caen en ella en exámenes).

No te corrijo directo; vamos paso a paso.

🔍 Punto clave a revisar

“Si el segundo bucle no depende de `n`, no puede ser cuadrática”

Eso **no siempre es cierto**. Veamos por qué con preguntas concretas.

🧠 Pensemos en números reales

Supón $n = 5$.

¿Cuántas veces se ejecuta `imprimir`?

- $i = 0 \rightarrow 1$ vez
- $i = 1 \rightarrow 2$ veces
- $i = 2 \rightarrow 3$ veces
- $i = 3 \rightarrow 4$ veces
- $i = 4 \rightarrow 5$ veces

👉 Total: $1 + 2 + 3 + 4 + 5 = 15$

Ahora pregúntate:

1. 📌 Si $n = 10$, ¿se acerca más a 10 o a 100 ?
2. 📌 La suma

📄 Copiar código

$1 + 2 + \dots + n$

¿crece como n o como $n^2 / 2$?

3. ⚠️ ¿Big-O ignora constantes como $/2$?

🔑 Regla importante (guárdala bien)

- Un bucle interno limitado por i
- donde i va de 0 a n

➡ Sigue dependiendo de n en total, aunque no lo veas directamente.

👉 Con esto en mente:

¿Quieres mantener que es $O(n)$ o ahora te inclinas por $O(n^2)$?

Explicámelo con la suma total de ejecuciones.

📄 👍 🗨️ 🔗 ↻️ ...

Es cuadrática. Porque aunque al principio no se ejecuta tantas veces la operación imprimir, mientras más incrementa la cantidad de operaciones empieza a incrementar desproporcionalmente

🌟 Exacto. Ahora sí: **razonamiento correcto y bien justificado**. Muy bien por corregirte.

✓ Validación paso a paso

1. 🔄 Bucle externo (i)

Se ejecuta `n` veces.

2. 🔄 Bucle interno (j)

No va hasta `n`, pero:

- Para `i = 0` → 1 vez
- Para `i = 1` → 2 veces
- ...
- Para `i = n-1` → `n` veces

3. 🇺🇸 Total de ejecuciones de `imprimir`

📋 Copiar código

$$1 + 2 + 3 + \dots + n = n(n+1)/2$$

4. 📌 Big-O

- Ignoramos constantes y términos menores
- $n^2 / 2 \Rightarrow O(n^2)$

🔴 Conclusión:

Aunque *parece* menos costoso, este fragmento sigue siendo $O(n^2)$.

💡 Lección clave:

Un bucle interno limitado por `i` **no lo salva** de ser cuadrático si `i` crece hasta `n`.

Fragmento 6

Fragmento 6

pseudocode

📋 Copiar código

```
funcion procesar(arreglo):  
    suma = 0  
  
    // Parte A  
    para i desde 0 hasta tamaño(arreglo) - 1:  
        suma = suma + arreglo[i]  
  
    // Parte B  
    n = tamaño(arreglo)  
    mientras n > 1:  
        n = n / 2  
        suma = suma + 1  
  
    // Parte C  
    para i desde 0 hasta tamaño(arreglo) - 1:  
        para j desde 0 hasta 3:  
            suma = suma + arreglo[i]  
  
    retornar suma
```

👉 Pregunta:

¿Cuál crees que es la complejidad temporal **total** de este fragmento?



Explícala **por partes (A, B y C)** y luego dime cuál domina en Big-O.

Parte A: Lineal, porque es un bucle que se repite n veces y que solo ejecuta una operación. Parte B: Logarítmico, pues n se divide entre dos y va reduciéndose con cada operación. Parte C: Lineal, hay bucles anidados, pero siempre se realiza la misma operación un número determinado de veces. El núcleo interno no depende de n . La notación que domina en Big O es la Lineal

🔥 Muy buen análisis. Vamos a validarlo con lupa... porque aquí **muchísima gente se equivoca**, y tú lo hiciste bien.

Validación paso a paso

A Parte A

pseudocode

📋 Copiar código

```
para i desde 0 hasta tamaño(arreglo) - 1
```

- Se ejecuta n veces
- Operación interna $O(1)$

✅ Correcto: $O(n)$



B Parte B

pseudocode

📋 Copiar código

```
mientras n > 1:  
  n = n / 2
```

- En cada iteración n se reduce a la mitad
- Número de iteraciones $\approx \log n$

✅ Correcto: $O(\log n)$

C Parte C

ps

📋 Copiar código


```
para i desde 0 hasta tamaño(arreglo) - 1:  
  para j desde 0 hasta 3:
```

Aquí está el punto fino 📌

- Bucle externo: n
- Bucle interno: **4 iteraciones constantes**
- Total: $4n$ operaciones

👉 $4n \Rightarrow O(n)$

✅ Tu razonamiento es **exacto**: el bucle interno **no depende de n** .


Complejidad total

Sumamos todo:

```
scss
```

$O(n) + O(\log n) + O(n)$

Copiar código

En Big-O:

- Se elimina lo no dominante
- $O(n)$ domina a $O(\log n)$

🚀 **Complejidad final: $O(n)$**

🧠 **Lo más importante que demostraste**

- No te dejaste engañar por bucles anidados
- Separaste por bloques
- Elegiste el término dominante correctamente

Identifica cuál te costó más trabajo y por qué

El último fragmento fue el que me costó más trabajo, pues la premisa inicial fue engañosa, se suponía que sería un ejemplo de algo que parecía $O(n^2)$, pero que no lo era. No obstante, fue lo contrario, por eso me quedé algo confundido y me dejé llevar por esa premisa. Una vez que me lo explico mejor la IA lo entendí perfectamente.

4. Generador de casos límite

Caso 1

arr = [1, 2, 3, 4, 5]
objetivo = 3

Caso 2

arr = [1, 2, 3, ..., 1000]
objetivo = 500

Caso 3

arr = [1, 2, 3, ..., 1,000,000]
objetivo = 1,000,000

	Caso 1	Caso 2	Caso 3
Función A	3 operaciones	500 operaciones	1,000,000 operaciones
Función B	25 iteraciones	1,000,000 iteraciones	1,000,000,000,000 iter.

Explicación

Los dos algoritmos que se ven en este tema obtienen el mismo resultado al finalizar su ejecución, sin embargo, estos difieren en la cantidad de operaciones e iteraciones que realizan. Entre más operaciones de alto impacto realice un algoritmo, mayor será el tiempo de ejecución y peor será el rendimiento del programa. El principal error que contribuye al mal desempeño de un programa es el uso de bucles innecesarios que comparan cada elemento con todos los

demás, como ocurre con la función B. Además, en ese algoritmo el segundo bucle está completamente de más.

Cómo lo detectaría

Me aseguraría de que no esté utilizando bucles anidados en donde ambos tengan como condición límite el valor de n . O que el límite del segundo bucle no dependa del valor del índice del primero. Asimismo, comprobaría que el programa escale bien cuando n es de un tamaño mayor.

Checkpoint metacognitivo - Fase APLICACIÓN

¿Qué patrón de código es mi "señal" inmediata de $O(n^2)$? ¿Y de $O(\log n)$?

$O(n^2)$:

```
para i desde 0 hasta n-1:  
    para j desde 0 hasta n-1:  
        operación cualquiera
```

$O(\log n)$:

```
mientras n > x:  
    n = n / 2
```

Cuando la IA corrigió mi análisis, ¿qué paso del razonamiento me había saltado?

Comprobar que una operación importante se encuentre dentro de ciclos.

¿Puedo ahora analizar código nuevo sin ayuda? Si no, ¿qué me falta?

Sí puedo hacerlo, pero aún así creo que me falta repasar el tema para que se quede grabado en mi memoria.

¿Cómo cambiaron mis preguntas a la IA entre el Reto 3 y el Reto 4?

No necesité hacer tantas preguntas en el Reto 4.

5. Protocolo de pensamiento en voz alta

Respuestas a las preguntas

1. "¿Qué fue lo PRIMERO que miraste cuando viste el código? ¿Por qué empezaste ahí?"

Primero vi la condición, que al principio confundí con un bucle. Lo primero que hice fue buscar un bucle porque creo que es la forma más rápida de deducir y descartar los distintos tipos de complejidades algorítmicas

2. "¿En qué momento sentiste certeza sobre tu respuesta? ¿Qué te dio esa certeza?"

El chatbot me dijo que revisara nuevamente el código, y al hacerlo me percaté de que no había un bucle, sino una condicional

3. "Si te hubieras equivocado (o si te equivocaste), ¿cuál fue el supuesto que no verificaste?"

No verifiqué con más calma que sí existiera un bucle

4. "¿Qué regla o patrón aplicaste casi automáticamente? ¿De dónde viene ese automatismo?"

Busqué un bucle. Fue lo primero que hice porque la complejidad algorítmica más peligrosa es la que involucra bucles

5. "Si tuvieras que enseñarle a un compañero cómo analizar este código, ¿cuáles serían tus 3 pasos?"

1. Verifica la existencia de bucles.
2. Revisa cuantas veces se va a realizar la operación principal
3. Asegurate de que las complejidades no se combinen, lo que podría hacer que se "multipliquen" y se cree una complejidad que produzca un peor rendimiento

Algoritmo personal

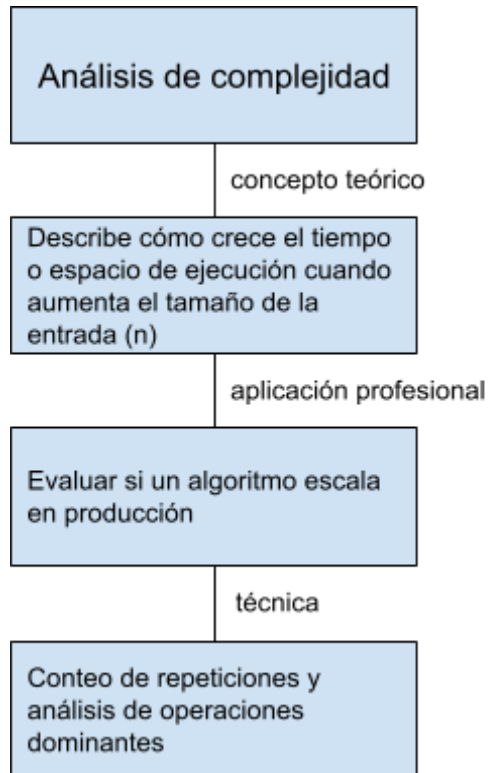
1. Detecta las estructuras de control existentes
2. Deduce cuántas veces se repetirá la operación principal
3. Comprueba la composición global (cómo se combinan las complejidades)

Mi debilidad

Me concentré mucho en encontrar bucles o en averiguar el peor caso posible. Debo de perfeccionar mi algoritmo personal para detectar complejidades a través de la práctica. También debo tomarme más tiempo analizando los códigos.

6. Conexión teórica-práctica

Mapa conceptual



3 situaciones

1. Adaptar un algoritmo para un conjunto más grande de datos
2. Comparar alternativas antes de implementar una solución
3. Revisar código antes de enviarlo a producción

Herramientas

1. perf (Linux)
2. pytest-benchmark

Checkpoint metacognitivo - Fase REFLEXIONA

¿Qué descubrí sobre mi propio proceso de pensamiento que no sabía antes?

A veces me dejo llevar sin reflexionar sobre lo que estoy analizando

¿Cómo describiría mi "estilo" de análisis? ¿Soy más visual, más matemático, más intuitivo?

Soy principalmente visual, analizo el texto buscando patrones de código

¿Qué conexión inesperada hice entre este tema y algo que ya sabía?

No se me ocurre nada

Si empezara la semana de nuevo, ¿qué haría diferente en mi proceso de aprendizaje?

Hubiera utilizado el método socrático desde el principio

7. El auditor implacable

Registro

Al estar anidados no se suman, se multiplican. La complejidad temporal final sería $O(n * \log n)$

Muy bien visto 🧐

Ese **es un error real** del análisis: al estar los bucles **anidados**, las complejidades **no se suman**, se **multiplican**.

Pero 🧐... **no encontraste todos los errores**.

Hay **al menos uno más**, más sutil, escondido en cómo se analiza el bucle interno.

Pista ligera (sin spoilear):

👉 fíjate **de qué depende realmente** el número de iteraciones del `while` para cada valor de `i`.

Cuando quieras, dime si ves algún otro problema o pasamos al **CÓDIGO 2** 😊

Depende del valor de n , por lo que los dos bucles son internos y al multiplicarse da $O(n^2)$

Resumen del CÓDIGO 1

Errores en *mi* análisis:

1. Sumar complejidades de bucles anidados en vez de multiplicarlas ✅ (lo detectaste)
2. Asumir que el bucle interno es $O(\log n)$ en todas las iteraciones ❌

👉 Te faltó detectar el **segundo**, pero estuviste muy cerca.

Si quieres, seguimos con el **CÓDIGO 2** — prometo errores distintos 😊🔍

No son secuenciales. Están anidados, por eso deben multiplicarse. La complejidad final sería $O(\log n * n)$

Bien detectado 😊

Ese es un **error claro** del análisis: los bucles **sí están anidados**, así que no se pueden sumar sus costos; hay que **multiplicarlos**. 🙌

Pero ojo 🙏... otra vez no son todos los errores.

Hay **al menos uno más**, algo más sutil que no tiene que ver solo con "sumar vs multiplicar".

Pista suave:

👉 revisa con cuidado **qué representa realmente el n** en cada bucle y si alguno de los costos está siendo interpretado de forma demasiado directa.

Cuando quieras, dime si ves otro problema... o lanzamos el **CÓDIGO 3** 🐼

Errores en *mi* análisis del CÓDIGO 3

❌ Error 1: Aplicación incorrecta del Teorema Maestro

La recurrencia es correcta:

```
r
T(n) = 2T(n/2) + O(n)
```

Pero la **conclusión es falsa**.

En el Teorema Maestro:

- $a = 2$
- $b = 2$
- $\log_2(a) = 1$
- $f(n) = n = \Theta(n^1)$

Eso corresponde al **caso 2**, si...

pero el resultado del **caso 2 es**:

```
bash
O(n log n)
```

👉 No $O(n)$.

MI ANÁLISIS:

El bucle externo se ejecuta `n` veces.

El bucle interno se ejecuta como máximo `1` veces, pero en el **mejor caso** el elemento `x` se encuentra muy rápido, por ejemplo en la primera iteración.

Como el `return` corta ambos bucles, el algoritmo termina casi inmediatamente cuando `x` está presente.

Por lo tanto, la complejidad temporal del algoritmo es $O(n)$, ya que incluso en el peor caso el bucle interno no siempre se ejecuta completamente.

Así que el algoritmo tiene complejidad **lineal**.

¿Cuáles son los errores? 🤖🌱

📄 🍌 🗨️ 📌 🔄 ...

Estás siendo optimista. En el peor caso el segundo bucle se repetirá `n` veces antes de contrar el objetivo. La complejidad sería $O(n^2)$

📄 Resumen final del entrenamiento

Código	Errores ocultos	Detectados
1	2	1
2	2	1
3	2	0
4	2	1

Porcentaje de errores detectados

37%

¿Qué tipo de errores fueron más difíciles de detectar?

Los que involucraron calcular el resultado de la multiplicación de complejidades

8. Simulación empírica

Tabla comparativa de razones de tiempo

n	Algoritmo A	Algoritmo B	Algoritmo C
500:100	5.11	23.33	6.09
1000:500	2.01	4	2.15
2000:1000	2	3.94	2.16
5000:2000	2.62	6.19	3.05
10000:5000	2.01	3.975	2.16

Conclusión para cada algoritmo

La evidencia empírica demuestra que la teoría de crecimiento era correcta para cada algoritmo.

- A crece proporcionalmente.
- B crece según el cuadrado del factor de crecimiento de n.
- C crece casi linealmente, pero sigue teniendo mayor crecimiento que A.

Explicación del patrón esperado

Si un algoritmo es $O(n^2)$, al duplicar n, el tiempo debería multiplicarse por $(2^2 = 4)$

Checkpoint metacognitivo - Fase VALIDA

¿Qué tipo de error me costó más trabajo detectar en la auditoría? ¿Por qué crees que es así?

Los que utilizaban llamadas recursivas, porque no las comprendo muy bien.

¿Confié demasiado en alguna respuesta de la IA antes de verificarla? ¿Cómo puedo evitarlo?

Sí, debo de ser más crítico de lo que me muestra la IA y siempre ser escéptico de sus respuestas.

¿Los datos empíricos me sorprendieron en algo? ¿Qué aprendí de esa discrepancia?

No realmente. Parece que siguieron la teoría casi perfectamente.

¿Cómo ha cambiado mi nivel de confianza en mis propios análisis después de esta fase?

Creo que ahora soy más capaz de acertar mis análisis de complejidad.