Fase 1: comprende

1.- Explorador de complejidades

Tabla comparativa de las 6 complejidades con: analogía personal que te haga sentido, patrón de código que la genera, y una situación donde la has visto o podrías verla.

Complejidad	Analogía	Patrón de código	situación donde la has visto o podrías verla.
O(1) — Tiempo constante	Sacar una llave de un llavero donde cada llave tiene su gancho etiquetado. No importa cuántas llaves haya: si conoces la etiqueta y el gancho, la operación toma el mismo tiempo.	func obtener_elemento(arr, i): return arr[i]	Acceso por índice a arrays, lectura de claves en hash map promedio (lookup si asumimos buen hash), push/pop en stack (implementación basada en arreglo), comprobaciones de condiciones simples, lectura de configuración.

O(log n) — Logarítmico	Buscar una palabra en un diccionario físico: abres por la mitad, decides izquierda/derecha, vuelves a la mitad del subgrupo, etc. Cada paso reduce enormemente lo que queda por revisar.	func binary_search(arr, target):
		  lo = 0; hi = len(arr)-1
		  while lo <= hi:
		  mid = (lo+hi)//2
		  if arr[mid] == target: return mid
		  if arr[mid] < target: lo = mid+1 else hi = mid-1
		  return -1
			Búsqueda binaria en arrays ordenados, operaciones en árboles balanceados (AVL, Red-Black) como insertar/buscar, operaciones en heaps (altura ≈ log n), búsqueda en índices B-tree/B+tree en DBs

O(n) — Lineal	Recorrer una fila de personas para preguntarles si tienen puesto un distintivo; necesitas hablar con cada persona una vez en el peor caso.	func buscar_linear(arr, target):
		  for i in 0..len(arr)-1:
		  if arr[i] == target: return i
		  return -1
			Filtrar/contar elementos, copiar arrays, construir histogramas, validaciones que revisan todos los elementos, operaciones de I/O que leen líneas una a una, mapas que aplican transformación constante por elemento.

O(n log n) — n log n (ej. mergesort)	Organizar una pila grande de cartas dividiéndola repetidamente en mitades hasta cartas individuales (nivel log n) y luego reacomodando/mezclando	func merge_sort(arr):
		  if len(arr) <= 1: return arr
		  left = merge_sort(arr[:len/2])
		  right = merge_sort(arr[len/2:])
		  return merge(left, right)
			• Algoritmos de ordenamiento eficientes (merge sort, heapsort, quicksort promedio).
			• Transformadas/oper

	cada nivel (trabajo proporcional a n por nivel).		aciones que combinan niveles logarítmicos con trabajo lineal (p. ej. algunas implementaciones de FFT tienen n log n). ● Algoritmos de dividir y vencer con coste lineal de combinación.

O(n²) — Cuadrático	Comprobar quién conoce a quién en una fiesta revisando todas las parejas posibles: para cada persona preguntas por cada otra persona.	func todas_parejas(arr):
		  for i in 0..len(arr)-1:
		  for j in 0..len(arr)-1:
		  procesar(arr[i], arr[j])
			Algoritmos ingenuos: comparar todas las parejas (detección de duplicados por comparación pairwise), matrices de distancia (computar distancia entre todos los pares), algoritmos de fuerza bruta en pequeños n, implementaciones simples de sorts (bubble, selection) y algunos pasos en algoritmos de aprendizaje cuando no se optimiza.

O(2^n) — Exponencial	Encender/apagar n interruptores; el número de configuraciones posibles es 2^n (cada interruptor tiene 2 estados), y si examinas todas las configuraciones el trabajo crece exponencialmente.	func generar_subconjuntos(arr, i=0, cur=[]):
		  if i == len(arr): output(cur); return
		  generar_subconjuntos(arr, i+1, cur) // no incluir arr[i]
		  generar_subconjuntos(arr, i+1, cur + [arr[i]]) // incluir arr[i]
			● Búsqueda exhaustiva / enumeración de subconjuntos (subset sum, power set).
			● Algoritmos de fuerza bruta para problemas NP-hard (travelling salesman fuerza bruta, algunas soluciones de backtracking en el peor caso).
			● Recursiones ingenuas (p. ej. Fibonacci recursivo sin memoización).
			● En la práctica, se tolera solo para n pequeños; se usan heurísticas, podas (branch-and-bound) o programación dinámica para reducirlo.

2.- El Caso de DataStream Inc.

- Diagnóstico escrito del problema de DataStream (máximo 200 palabras).

El sistema que había desarrollado la empresa DataStream Inc. paso de tener 10k a 500k transacciones de un momento para otro; cuando eran aproximadamente 10k transacciones la información se procesaba en segundos y ahora que son 500k tarda casi una hora. esto se debe a la complejidad que el equipo de desarrollo utilizo (complejidad cuadrática) ya que esta mientras mas grande sea la entrada de datos que tenga, mas tardada sera la salida

- Cálculos que demuestren tu análisis de la complejidad probable.

n2/n1=500000/10000=50

(n2/n1)^2=50^2=2500 (lo cual la cuadratica es la que mas se acerca a los segundos en lo que tardaba en procesarse 500k de datos (osease 47 minutos o 2820 segundos)

- Una recomendación concreta: ¿Qué complejidad deberían buscar y por qué?

deberian de usar la complejidad lineal, ya que si por ejemplo digamos que tienen 1000000 transacciones en un dia con la lineal solo se harian ese numero de operaciones en cambio si usas otras complejidades las operaciones pueden crecer demasiado debido al numero tan alto de transacciones en el dia

Checkpoint Metacognitivo - Fase COMPRENDE

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

si, Big O mide la eficiencia de algun programa dependiendo de las entradas que este tenga

Si alguien me dice "mi algoritmo es O(n²)", ¿qué imagen mental se forma? ¿Puedo predecir su comportamiento?

que mi algoritmo es cuadratico, osease las operaciones totales que haga seran multiplicadas por si mismas. puedo predecir su comportamiento debido a que si tu algoritmo es O(n^2) mientras mas datos de entradas tengas, mas tardada sera la respuesta de tu algoritmo

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

de que se encarga la estructura de datos

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

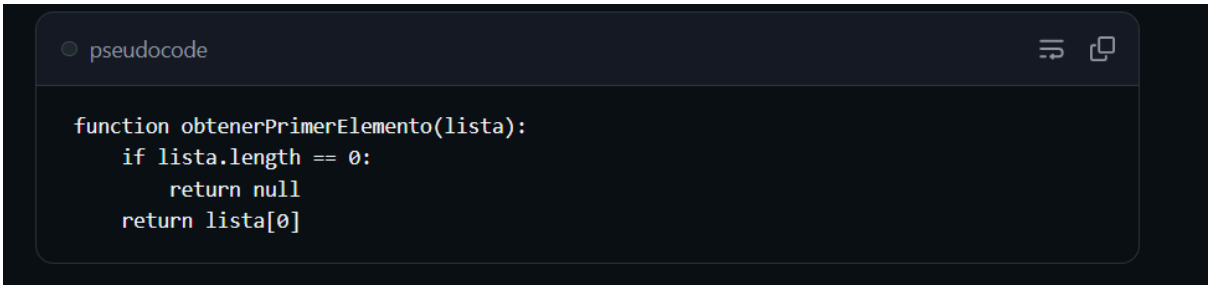
cuando note que dependiendo de como estructures tu algoritmo este puede tener distintas notaciones lo cual lo puede hacer que sea mas o menos eficiente

Fase 2: aplica

3.- Analizador de Código

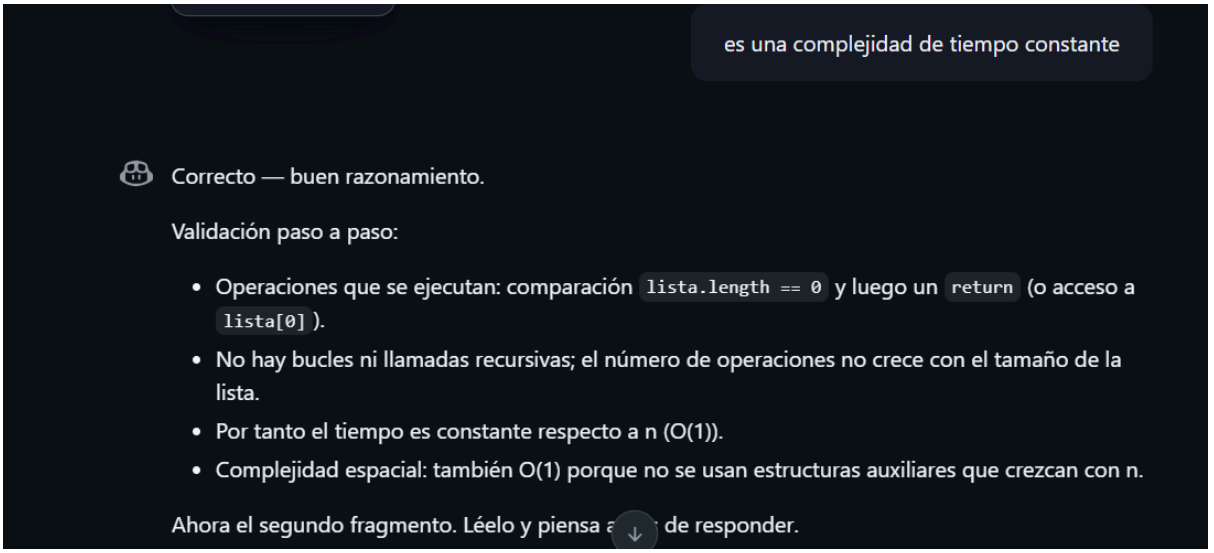
- Captura o registro de tu análisis de los 6 fragmentos con tu razonamiento escrito.

Algoritmo 1:



pseudocode

```
function obtenerPrimerElemento(lista):
    if lista.length == 0:
        return null
    return lista[0]
```



es una complejidad de tiempo constante

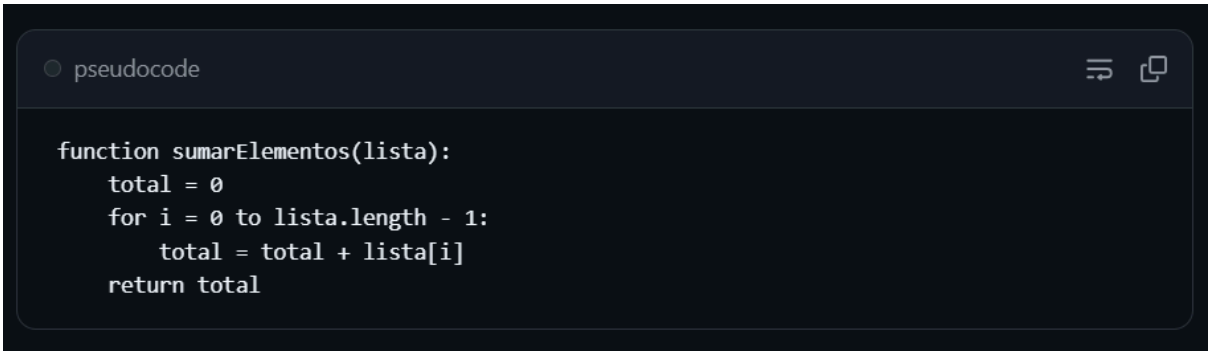
Correcto — buen razonamiento.

Validación paso a paso:

- Operaciones que se ejecutan: comparación `lista.length == 0` y luego un `return` (o acceso a `lista[0]`).
- No hay bucles ni llamadas recursivas; el número de operaciones no crece con el tamaño de la lista.
- Por tanto el tiempo es constante respecto a n (O(1)).
- Complejidad espacial: también O(1) porque no se usan estructuras auxiliares que crezcan con n.

Ahora el segundo fragmento. Léelo y piensa a de responder.

Algoritmo 2:



pseudocode

```
function sumarElementos(lista):
    total = 0
    for i = 0 to lista.length - 1:
        total = total + lista[i]
    return total
```

perdon me confundi, quise decir que era O(n)

Correcto — O(n). Buen trabajo.

Validación paso a paso:

- El bucle `for` recorre cada elemento de la lista exactamente una vez, ejecutando una suma por iteración.
- Si la lista tiene n elementos, la operación principal se ejecuta n veces, así que el tiempo crece linealmente con n.
- No hay bucles anidados ni recursión, y las operaciones dentro del bucle son de coste constante.
- Complejidad espacial: O(1), porque solo se usan unas pocas variables (`total` , índice `i`) que no crecen con n.

Algoritmo 3:

pseudocode

```
function imprimirTodosLosPares(lista):
    for i = 0 to lista.length - 1:
        for j = 0 to lista.length - 1:
            print(lista[i], lista[j])
```

O(n^2)

Correcto — O(n^2). Buen razonamiento.

Validación paso a paso:

- Hay dos bucles for anidados: el externo recorre i desde 0 hasta n-1 y por cada i el interno recorre j desde 0 hasta n-1.
- Para cada par (i, j) se ejecuta una impresión, así que el número total de operaciones principales es n * n = n^2.
- Por tanto el tiempo crece proporcionalmente a n^2, es decir O(n^2).
- Complejidad espacial: O(1), porque solo se usan las variables de índice `i` y `j` además del acceso a la lista; no se crea ninguna estructura adicional dependiente de n.

Algoritmo 4:

pseudocode

```
function buscarElemento(lista, objetivo):
    inicio = 0
    fin = lista.length - 1
    while inicio <= fin:
        medio = floor((inicio + fin) / 2)
        if lista[medio] == objetivo:
            return medio
        else if lista[medio] < objetivo:
            inicio = medio + 1
        else:
            fin = medio - 1
    return -1
```

es O (log n)

Correcto — O(log n). Buen razonamiento.

Validación paso a paso:

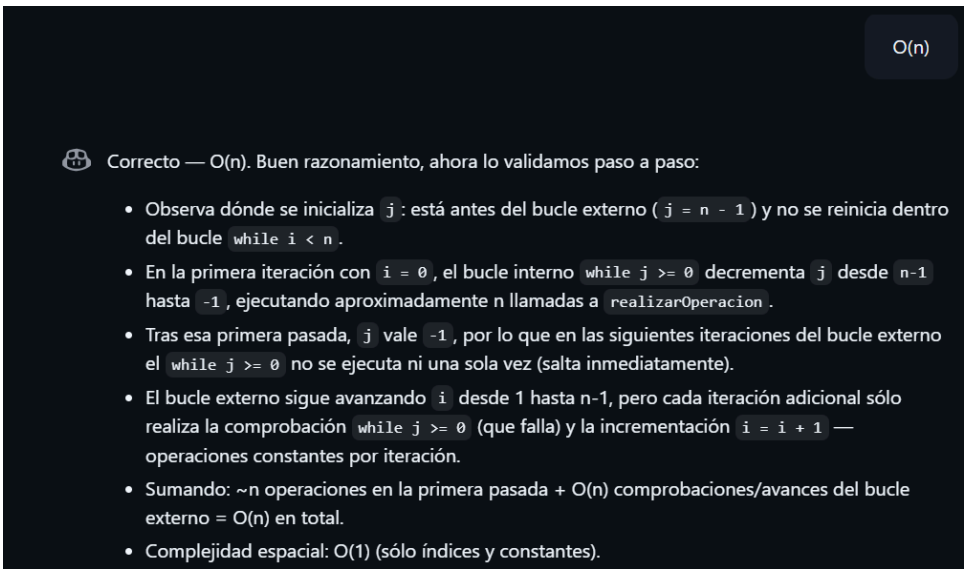
- Cada iteración del while calcula el medio y compara; luego reduce el intervalo al eliminar la mitad izquierda o derecha.
- Si el intervalo tiene k elementos, tras una iteración queda aproximadamente k/2.
- Empezando con n elementos, el número de iteraciones necesarias para llegar a tamaño 1 es aproximadamente log2(n).
- El trabajo dentro de cada iteración (acceso y comparaciones) es constante, así que el tiempo total es proporcional al número de iteraciones: O(log n).
- Complejidad espacial: O(1), porque solo se usan unas pocas variables (inicio, fin, medio).

Ahora, el quinto fragmento (parece O(n^2) pero no lo es). Léelo y piensa antes de responder.

Algoritmo 5:

pseudocode

```
function procesoEnganoso(lista):
    n = lista.length
    i = 0
    j = n - 1
    while i < n:
        while j >= 0:
            realizarOperacion(lista[i], lista[j])
            j = j - 1
        i = i + 1
```



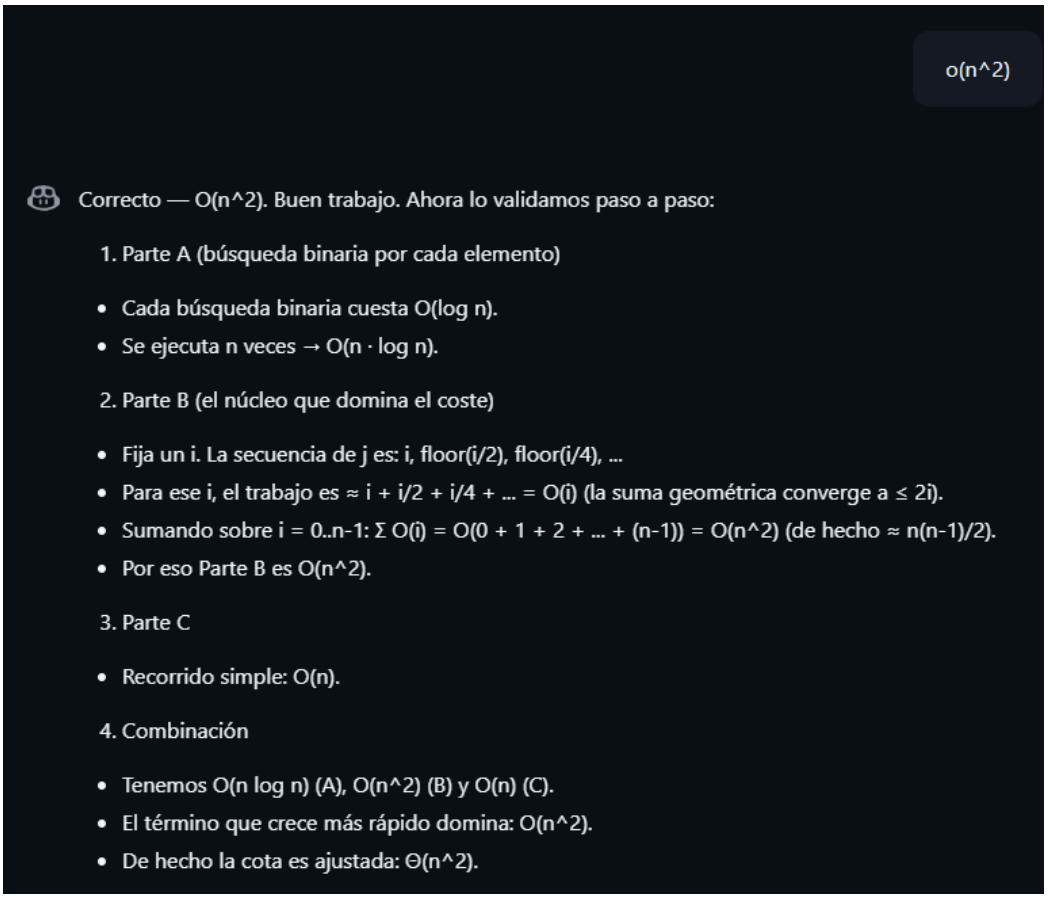
Algoritmo 6:

```
function mezclaCompleja(lista):
    n = lista.length

    // Parte A: para cada elemento, haces una búsqueda binaria en la lista
    for i = 0 to n - 1:
        objetivo = lista[i]
        inicio = 0
        fin = n - 1
        while inicio <= fin:
            medio = floor((inicio + fin) / 2)
            if lista[medio] == objetivo:
                break
            else if lista[medio] < objetivo:
                inicio = medio + 1
            else:
                fin = medio - 1

    // Parte B: para cada i, haces un bucle decreciente por mitades,
    // y en cada paso ejecutas un bucle lineal hasta j
    for i = 0 to n - 1:
        j = i
        while j > 0:
            k = 0
            while k < j:
                realizarOperacion(lista[k])   // coste O(1)
                k = k + 1
            j = floor(j / 2)
```

```
// Parte C: un simple recorrido final
for x = 0 to n - 1:
    realizarOperacionConstante()
```



- Para cada uno, documenta: tu hipótesis inicial, el razonamiento paso a paso, y la conclusión final.

Algoritmo 1:

Mi hipótesis inicial era 0(1), ya que para empezar esa es la complejidad más simple de todas debido a que este es independiente a la entrada, por lo que da igual cuantos datos de entradas tengas, siempre hará la misma cantidad de operaciones.

Algoritmo 2:

En este algoritmo mi hipótesis inicial era O(1), pero solo fue por una confusión que tuve. termine concluyendo que era un O(n) debido a que el algoritmo nada más tiene un loop simple.

Algoritmo 3:

Desde el primer momento supe que era O(n^2) al ver dos loops anidados.

Algoritmo 4:
Mi primera hipótesis fue O(n log n), pero luego de analizar bien el código vi que no se rearmaba el código después de ser dividido, por lo que di por sentado que era O (log n).

Algoritmo 5:
Mi primera hipótesis fue O(n^2) pero el ejercicio decía que era una pregunta engañosa, así que descarte esa posibilidad, luego llegué a pensar brevemente que podría ser un O(n log n) pero lo descarté rápidamente debido a que si bien el ejercicio tenía un while para que sea linealitmica tiene que ver una división y una reestructuración de los datos. Estuve un rato analizando el algoritmo y concluí que era O(n) ya que si bien tiene dos while anidados este funciona de forma secuencial como lo sería una complejidad lineal.

Algoritmo 6:
Mi primera hipótesis fue que era un O(n log n) ya que fue la primera complejidad que vi, pero debido a que el código estaba desglosado en partes y cada parte tenía varias complejidades, trate de fijarme cual era la más “importante” por lo que concluí que esta era la complejidad O(n^2), ya que en cada parte del código se iniciaba un for, por lo que eran tres for anidados dentro de las tres partes del código.

- Identifica cuál te costó más trabajo y por qué.

Yo creo que la que más me costó trabajo fue el ejercicio 5 debido a que me dejó pensando que complejidad podría ser si no era cuadrática, hasta que en un punto note que dependiendo de la entrada que pusiera era la cantidad de procesos que se ejecutan.

4.- Generador de Casos Límite

- Tabla comparativa con los cálculos de operaciones para cada caso.

Caso	n	Posición (1-based)	Función A operaciones	Función B operaciones
Caso 1	100	50	50	4950
Caso 2	10000	5000	5000	49995000
Caso 3	1000000	1000000	1000000	1000000000000

- Explicación de 100 palabras sobre por qué códigos "que funcionan igual" pueden tener rendimientos radicalmente diferentes.

Dos programas pueden tener distinta estructura pero pueden hacer lo mismo. En este caso si bien la función A (lineal) y la función B (cuadrática) hacen lo mismo que es buscar cierto elemento en una lista, estas funciones tienen distinto rendimiento. La función A se desplaza de forma lineal, por lo que no tiene que hacer desplazamientos “innecesarios”, mientras que la función B, al ser cuadrática y tener dos loops anidados, tiene que terminar primero un loop para empezar el otro, por lo que básicamente esta multiplicando al cuadrado todas las iteraciones cometidas, por lo que al final termina haciendo más procesos y eso hará que el algoritmo sea menos óptimo.

- Propuesta de cómo detectarías este tipo de problemas en código real.
  la propuesta que doy para detectar este tipo de problemas es si hay un problema respecto al tiempo en los procesos de los datos, revisar el código y si vemos que hay dos o más loops anidados cambiarlos por la complejidad O(n), así es probable que reduzcamos el tiempo de carga de los datos y sea más eficiente nuestro programa.

Checkpoint Metacognitivo - Fase APLICA

- ¿Qué patrón de código es mi "señal" inmediata de O(n²)? ¿Y de O(log n)?

Si veo que un algoritmo tiene dos o más for anidados es que es O(n^2), y si veo que un algoritmo tiene en una parte donde se usa una división para comparar valores sabré que es O (log n).

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

¿Puedo ahora analizar código nuevo sin ayuda? Si no, ¿qué me falta?
Podría intentarlo pero no estoy seguro de poder por ejemplo con algun O(n) engañoso como el de una pregunta. Supongo que me hace falta estudiar mas el tema.

¿Cómo cambiaron mis preguntas a la IA entre el Reto 3 y el Reto 4?
asumieron cierto rol acordado al reto

FASE 3: REFLEXIONA

5.- Protocolo de Pensamiento en Voz Alta

- Documento de reflexión de tus respuestas a las 5 preguntas.

Fortalezas metacognitivas (qué haces bien)

Reconoces y nombras tus automatismos: identificaste que tu reacción inicial fue aplicar reglas aprendidas (p. ej. “dos while → cuadrática”) en vez de analizar el comportamiento real; eso es clave para poder controlarlo.
Revisas y corriges hipótesis: no te quedaste con la primera intuición; volviste a mirar el propósito del algoritmo y cambiaste la conclusión al ver que los while eran secuenciales.
Te fijas en el propósito del código: usar la finalidad (buscar un valor secuencialmente) como evidencia es una buena práctica para orientar el análisis.
Eres honesto sobre tu incertidumbre y la fuente del automatismo (estudio previo), lo cual facilita planear prácticas dirigidas.

Sugerencia específica y práctica (una sola mejora accionable)

Adopta un checklist sistemático para decidir si bucles “anidados” son realmente dependientes (cuadráticos) o secuenciales (lineales). Aplica este procedimiento cada vez que analices loops:

Identifica los bucles y la relación entre ellos: ¿el while/for interno se reinicia completamente para cada iteración del externo o continúa avanzando sin retroceder?
Cómo comprobarlo: busca si el índice del inner se inicializa en función del outer o si avanza desde donde quedó.
Expresa el número de iteraciones en función de n (o de los índices externos): escribe T(n) ≈ sumatoria de (iteraciones del inner) sobre iteraciones del outer.
Si la sumatoria es O(n) → lineal; si es O(n^2) → cuadrática.
Verifica el coste del cuerpo del bucle: ¿hay operaciones que no son O(1) (copias de arrays, reordenamientos, llamadas costosas)? Inclúyelas en T(n).
Confirma con un trazado pequeño: elige n pequeño (p. ej. 5 o 6) y cuenta explícitamente cuántas veces se ejecuta la línea clave (o el inner). Si el conteo crece aproximadamente como n vs n^2, ya tienes evidencia empírica.

- Tu "algoritmo personal" de 3-5 pasos para analizar complejidad.

algoritmo personal

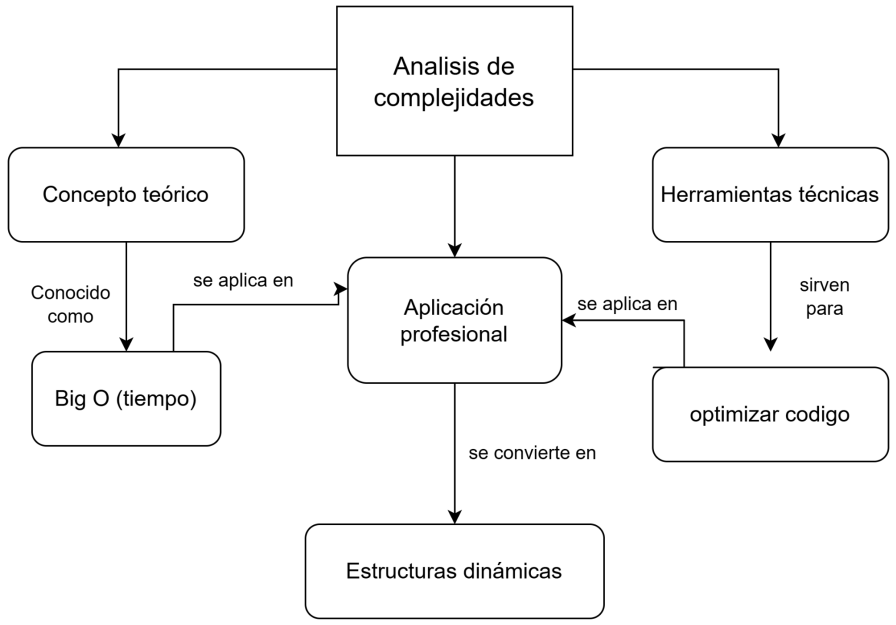
paso 1: revisar que operadores y que estructuras hay en el codigo
paso 2: analizar la relación que hay entre las estructuras
paso 3:revisar el proposito del codigo
paso 4: sustituir n por valores pequeños para poder hacer mas simple el entendimiento del codigo

- Una debilidad identificada y un plan concreto para abordarla.

la debilidad mas notoria que tengo es mi poca capacidad para comprender codigo, puedo determinar que complejidad se esta usando mas sin embargo a la hora de analizar como funciona el codigo batallo un poco, por lo que la forma en la que puedo abordar esta dificultad es estudiando mas.

6.- Conexión Teoría-Práctica

- Mapa conceptual conectando: Concepto teórico → Aplicación profesional → Herramientas/Técnicas



- 3 situaciones de tu futura carrera donde aplicarás análisis de complejidad.

  situacion 1: incrementacion de tiempo de espera de carga de datos cuando trabajamos con demasiados datos de entrada

  situacion 2: calcular la similitud entre usuarios por comportamiento para recomendaciones

  situacion 3: detección de fraude con límites de latencia

- Lista de 2-3 herramientas de profiling/benchmarking que investigarás.

1: async-profiler
2: py-spy
3: k6

Checkpoint Metacognitivo - Fase REFLEXIONA

¿Qué descubrí sobre mi propio proceso de pensamiento que no sabía antes?
siempre trato de analizar antes de actuar

¿Cómo describiría mi "estilo" de análisis? ¿Soy más visual, más matemático, más intuitivo?
mi estilo de analisis lo considero mas visual, ya que viendo ciertas estructuras en ciertos algoritmos puedo determinar donde esta la mala optimizacion del codigo

¿Qué conexión inesperada hice entre este tema y algo que ya sabía?
en que los algoritmos en la programacion son muy similares a los de la vida cotidiana

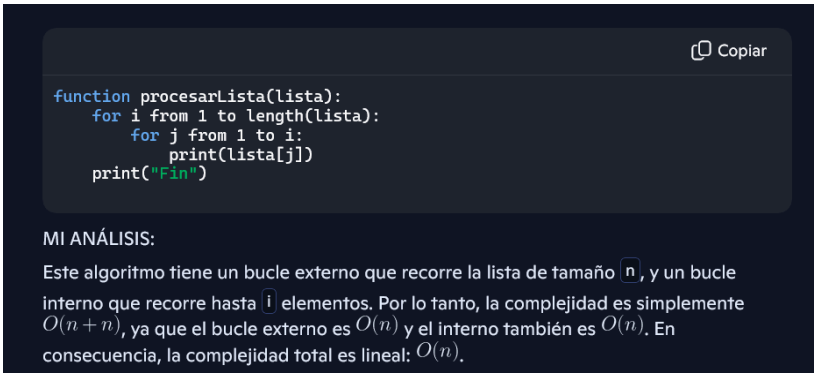
Si empezara la semana de nuevo, ¿qué haría diferente en mi proceso de aprendizaje?
investigar en otros lados respecto al tema antes de empezar, para asi comprender mejor el tema y mas aparte de eso usar un prompt similar al ejercicio 6 para reafirmar mis conocimientos

FASE 4: VALIDA

7.- El Auditor Implacable

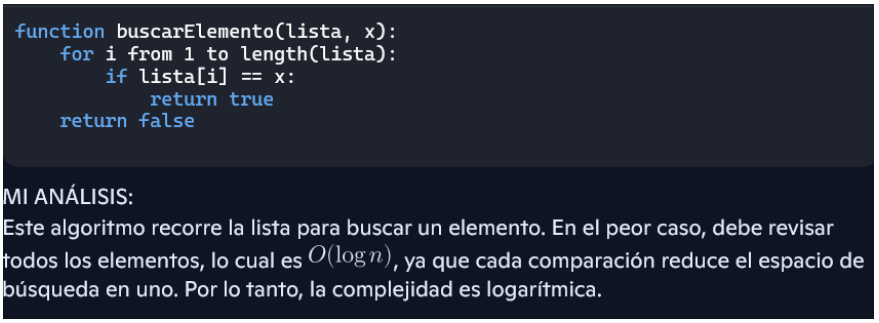
- Registro de los 4 casos con: tu identificación de errores, la corrección que propondrías, y verificación de si encontraste todos.

Caso 1:



Res
Respuesta: la complejidad no es lineal sino cuadrática, ya que al estar anidados no se suman sino se multiplican

Caso 2:

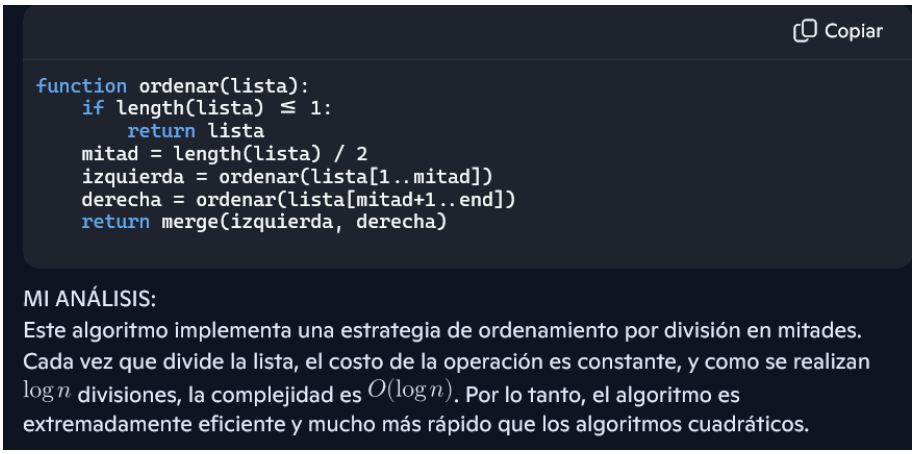


```
function buscarElemento(lista, x):
    for i from 1 to length(lista):
        if lista[i] == x:
            return true
    return false
```

MI ANÁLISIS:
Este algoritmo recorre la lista para buscar un elemento. En el peor caso, debe revisar todos los elementos, lo cual es $O(\log n)$, ya que cada comparación reduce el espacio de búsqueda en uno. Por lo tanto, la complejidad es logarítmica.

Respuesta: la complejidad es lineal y no logarítmica, ya que el algoritmo no esta dividiéndose en mitades para la búsqueda de un elemento y lo que hace concuerda con una complejidad lineal

Caso 3:



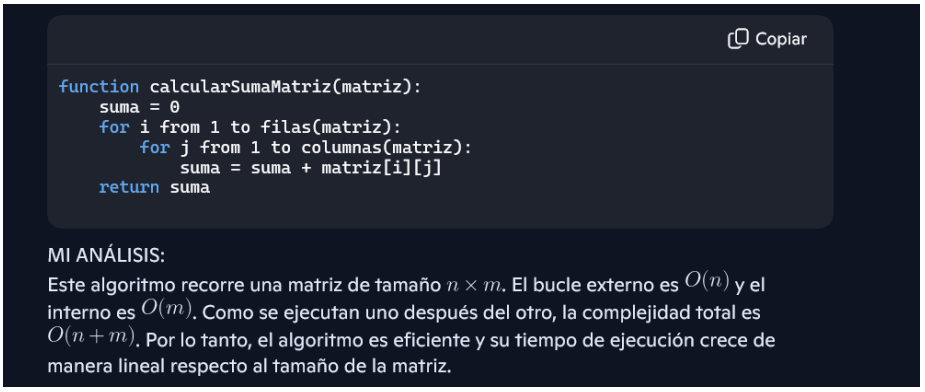
Copiar

```
function ordenar(lista):
    if length(lista) ≤ 1:
        return lista
    mitad = length(lista) / 2
    izquierda = ordenar(lista[1..mitad])
    derecha = ordenar(lista[mitad+1..end])
    return merge(izquierda, derecha)
```

MI ANÁLISIS:
Este algoritmo implementa una estrategia de ordenamiento por división en mitades. Cada vez que divide la lista, el costo de la operación es constante, y como se realizan $\log n$ divisiones, la complejidad es $O(\log n)$. Por lo tanto, el algoritmo es extremadamente eficiente y mucho más rápido que los algoritmos cuadráticos.

Respuesta: el algoritmo es linealitmico ya que si bien hay una división por mitades, estas se vuelven a ordenar

Caso 4:



Respuesta: la complejidad en realidad es cuadrática, mas aparte los dos bucles son n y no n y m. como dije es cuadrática la complejidad ya que al ver 2 for anidados y a que estos dependan de la entrada de datos que tengan al final termina tranformandose en n por n lo que significa que es cuadratica

- Porcentaje de errores detectados (meta: >75%).

100%

- categorización: ¿Qué tipo de errores fueron más difíciles de detectar?

los que tuvieron un logaritmo

8.- Simulación Empírica

- Tabla de datos con tus cálculos de razones entre tiempos consecutivos.

n->n	Razón a	Razón b	Razón c
500/100	2.01 / 0.42= 4.79	25.8 / 1.05 = 24.6	3.91 / 0.58 = 6.74
1000/500	4.08 / 2.01 = 2.03	102.6 / 25.8 = 3.98	8.72 / 3.91 = 2.23
2000/1000	8.36 / 4.08 = 2.05	411.4 / 102.6 = 4.01	18.9 / 8.72 = 2.17
5000/2000	21.2 / 8.36 = 2.54	2620 / 411.4 = 6.37	56.4 / 18.9 = 2.98
10000/5000	42.7 / 21.2 = 2.01	10480 / 2620 = 4.00	123.5 / 56.4 = 2.19

- Conclusión para cada algoritmo: ¿Los datos empíricos confirman la predicción teórica?

  no confirman al 100% pero si puedes suponer que tipo de complejidad es solo analizandola como datos empiricos

- Explicación del patrón esperado: "Si un algoritmo es O(n²), al duplicar n, el tiempo debería multiplicarse por ___".

si mismo

Checkpoint Metacognitivo - Fase VALIDA

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

yo creo que las complejidades que estan anidadas en general, a veces hay algunas que parecen cuadraticas y no lo son y como usualmente me guio por su estructura y no tanto por lo que hacen es donde me confundo

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

en estos momentos me ha pasado pero ahorita no, pieces evitarlo mejorando tus prompts para que la ia tenga mas contexto de que hacer o siempre puedes verificar o pedirle fuentes a la ia para que no haya errores

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

lo que me sorprendio de los datos empiricos es que solo con la experiencia ya puedes estimar que tipo de complejidad es con la que estas trabajando

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

un poco, aun tengo demasiadas cosas en las que mejorar