

## 1 Explorador de Complejidades

Complejidad	Analogia personal	Patron de codigo que lo genera	Situacion real en desarrollo
$O(1)$	Abrir un cajon y tomar el primer cubierto	Acceso directo a un indice de un arreglo: return array[0]	Consultar un valor en cache o acceder a un registro por ID
$O(\log n)$	Buscar un apellido en una guia telefonica partiendo por la mitad	Busqueda binaria: reducir el rango en cada iteración	Consultas en bases de datos con indices, busqueda en arboles balanceados
$O(n)$	Revisar cada asiento en un cine hasta encontrar a tu amigo	Recorrer un arreglo con un bucle simple	Validar todos los registros de un archivo o recorrer una lista de usuarios
$O(n \log n)$	Ordenar una biblioteca dividiendo en secciones y luego ordenando cada sección	MergeSort o QuickSort: dividir y conquistar	Ordenar grandes volumenes de datos, ranking en motores de busqueda
$O(n^2)$	Comparar cada estudiante con todos los demás para ver quién es más alto	Bucles anidados: comparar cada elemento con todos los demás	Algoritmos ingenuos de ordenamiento (BubbleSort), detección de duplicados en matrices
$O(2^n)$	Probar todas las combinaciones posibles de ropa antes de salir	Generar subconjuntos con recursion: cada elemento tiene dos opciones	Problemas combinatorios, fuerza bruta en optimizacion, arboles de decision en IA

La analogia que mas me sirvio fue la segunda de  $O(\log n)$  con la guia telefonica, porque muestra muy claramente cómo reducir el problema a la mitad en cada paso hace que el crecimiento sea mucho más lento que recorrer todo



## 2 El Caso de DataStream Inc.

El sistema de detección de fraude en DataStream funcionaba bien a 10,000 transacciones/día pero al escalar a 500'000 el tiempo pasó de segundos a 47 minutos esto indica que el algoritmo tiene escalamiento lineal cada transacción requiere procesamiento independiente, no hay estructuras enconomies para reducir la cantidad de trabajo por proceso escalar la capacidad del servidor en el 20% no ayudó mucho a mejorar el tiempo lo que indica que la mayor parte del trabajo no puede escalar o paralelizar otros procesan 1,000,000 en 3 minutos lo que indica que sus algoritmos son más eficientes (presumiblemente O(log n) o O(n log n)) combinadas con escalamiento o paralelización el problema no es en hardware ni en el diseño algoritmo el cuello de botella está en cómo recorren y procesan la transacción

$$T_n = T_v \cdot \frac{500000}{10000} = 1 \text{ min} \cdot 50 = 50 \text{ min}$$

$$\frac{1000000}{3 \text{ min}} = \frac{333333 \text{ transacciones}}{\text{min}}$$

$$\text{DataStream: } \frac{500000}{47 \text{ min}} = \frac{10638 \text{ transacciones}}{\text{min}}$$

Diferencia  $\approx 31 \times$  → implica O(log n) o O(n log n) con paralelización

Aunque DataStream todavía necesita mejorar su velocidad y hacer que su algoritmo sea O(log n) o O(n log n) a través de técnicas como árboles平衡ados, hashing distribuido, índices, su procesamiento será capaz de crecer mucho más lentamente que sus datos y esto le dará competitividad en el mercado

### 3 Analizador de Código

Fragmento 1:

**Fragmento 1 (nivel básico)**

Copy

```
function getFirstTransaction(transactions):
    return transactions[0]
```

👉 Pregunta: ¿Cuál crees que es la complejidad de este código?

Hipotesis Inicial: Mi hipotesis inicial fue que era un  $O(n)$

Razonamiento: Vi que estaba anidado y pense que era con estructura repetitiva como for, while, do-while, etc.

Conclusion: Vi que no tenia una estructura repetitiva y el transactions era solo de la posicion 0 por lo que es un  $O(1)$

Fragmento 2:

### Fragmento 2 (ejemplo con recorrido lineal)

 Copy

```
function findTransaction(transactions, target):
    for i in range(0, length(transactions)):
        if transactions[i] == target:
            return i
    return -1
```

Hipotesis Inicial: Mi hipotesis inicial fue un O(n)

Razonamiento: Vi que el bucle recorre todo el arreglo y comparandolo con la tabla me di cuenta de que si era un O(n)

Conclusion: El fragmento es de codigo es un O(n)

Fragmento 3:

### Fragmento 3 (ejemplo con bucles anidados)

 Copy

```
function compareTransactions(transactions):
    for i in range(0, length(transactions)):
        for j in range(0, length(transactions)):
            if transactions[i] == transactions[j]:
                markDuplicate(i, j)
```

Hipotesis Inicial: O( $n^2$ )

Razonamiento: El anidamiento de los for recorre n veces el arreglo lo que hace que sea  $n \times n$

Conclusion: el fragmento es un O( $n^2$ )

Fragmento 4:

#### Fragmento 4 (ejemplo con división)

 Copy

```
function halveUntilOne(n):
    while n > 1:
        n = n / 2
```

Hipotesis Inicial:  $O(\log n)$

Razonamiento: por que el procedimiento es proporcional a  $\log_2(n)$ ,

Conclusion:es un  $O(\log n)$

Fragmento 5:

#### Fragmento 5 (parece $O(n^2)$ , pero no lo es)

 Copy

```
function checkPairs(transactions):
    for i in range(0, length(transactions)):
        for j in range(i+1, i+2):
            process(transactions[i], transactions[j])
```

Hipotesis Inicial:  $O(n^2)$

Razonamiento: por los dos bucles que hay, pero el primer bucle si se ejecuta  $n$  veces y el otro solo una iteracion por lo que es un  $O(n)$

Conclusion:  $O(n)$

Fragmento 6:

## Fragmento 6 (combinado)

 Copy

```
function complexProcess(transactions):
    for i in range(0, length(transactions)):
        halveUntilOne(length(transactions))           // Loop A
        for j in range(0, length(transactions)):       // Loop B (divide entre 2
            if transactions[i] == transactions[j]:
                markDuplicate(i, j)
```

Hipotesis Inicial:  $O(\log n)$  y  $O(n^2)$

Razonamiento: Los tres bucles tienen partes de esas complejidades, pero dando esta respuesta a la ia vi que era otra combinacion

Conclusion: es un  $n \log n + n^2$

#### 4 Generador de Casos Límite

Escenario	Funcion A ( $O(n)$ )	Funcion B ( $O(n^2)$ )
n = 100, objetivo en posición 50	50 operaciones	5,000 operaciones
n = 10,000, objetivo en posición 5,000	5,000 operaciones	50,000,000 operaciones
n = 1,000,000, objetivo en última posición	1,000,000 operaciones	1,000,000,000,000 operaciones

Dos códigos pueden dar el mismo resultado pero tener un rendimiento muy diferente debido a su complejidad algorítmica la complejidad define el crecimiento del número de operaciones cuando se incrementa la magnitud de los datos un algoritmo  $O(n)$  tiene una escala lineal: si los datos se duplican también se duplica el tiempo, en cambio un  $O(n^2)$  tiene una escala cuadrática si se duplican los datos el tiempo se cuadriplica en listas pequeñas la diferencia es inobservable sin embargo en millones de elementos se transforma en un desastre por eso a pesar de que ambos devuelven el mismo índice uno es efectivo y el otro se vuelve poco práctico en términos productivos

- Revisar los bucles anidados, estructura de datos usadas
- Hacer pruebas de rendimiento ejecutando el mismo código con valores crecientes

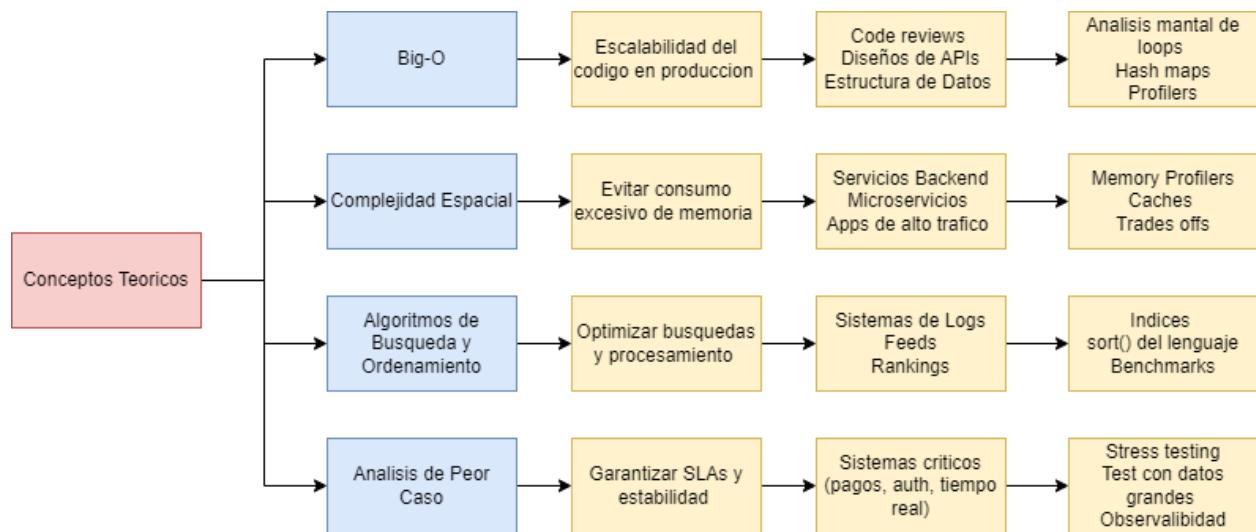
## 5 Protocolo de Pensamiento en Voz Alta

1. ¿Cuál fue lo PRIMERO que miraste cuando viste el código? ¿Por qué empezaste ahí? Por ejemplo en los fragmentos me basaba en los bucles y estructuras comparandolas con los ejemplos dados y de ahí partia comparando y analizando
2. ¿En qué momento sentiste certeza sobre tu respuesta? ¿Qué te dio esa certeza? Analizando los fragmentos y comparandolos
3. Si te hubieras equivocado (o si te equivocaste), ¿cuál fue el supuesto que no verificaste? En un fragmento donde era  $O(n)$  en vez de  $O(\log n)$ , y es que me precipite y no lo analice bien
4. ¿Qué regla o patrón aplicaste casi automáticamente? ¿De dónde viene ese automatismo? De ejemplos anteriores
5. Si tuvieras que enseñarle a un compañero cómo analizar este código, ¿cuáles serían tus 3 pasos? Analizar, comparar y reflexionar

Mi algoritmo personal como ya lo dije en la pregunta 5 es analizar, comparar y reflexionar a si me aseguro que con los dos primeros pasos ir en camino hacia la respuesta y el ultimo paso es reflexionar sobre la respuesta

Una debilidad identificada es que a veces me precipito en dar respuestas y un plan para abordarla seria seguir haciendo ejercicios para ir desarrollando mas paciencia y no adelantarme a dar respuestas erroneas

## 6 Conexión Teoría-Práctica



Situaciones:

- Code reviews en un equipo de desarrollo: detectar bucles anidados que podrían escalar mal con grandes volúmenes de datos
- Diseño de un sistema de búsqueda interna: decidir si usar un algoritmo de búsqueda lineal ( $O(n)$ ) o estructuras más eficientes
- Optimización de un servicio en producción: identificar consultas SQL con joins innecesarios  $O(n^2)$  y reemplazarlas por índices o particiones que reduzcan la carga

Herramientas:

- cProfile (Python): mide tiempo de ejecución por función
- JMH (Java Microbenchmark Harness): estándar en la industria para microbenchmarks en Java
- VisualVM (Java): profiler gráfico para analizar CPU, memoria y threads en aplicaciones Java

## 7 El Auditor Implacable

Codigo 1:

### CÓDIGO 1:

text

 Copy code

```
function processArray(arr):  
    n = length(arr)  
  
    for i from 1 to n:  
        j = i  
        while j > 0:  
            j = j / 2  
            print(arr[j])
```

Errores Identificados: Confusion entre la suma y multiplicacion de variables, uso incorrecto de j

Propuesta: El for corre n veces, para cada i el while corre  $O(\log i)$ ,

complejidad total:  $\sum_{i=1}^n \log i = O(n \log n)$

Complejidad Correcta:  $O(n \log n)$

¿Se encontraron todos los errores?

Se encontraron todos los errores

Código 2:

## CÓDIGO 2:

```
text Copy code

function findDuplicates(arr):
    n = length(arr)

    for i from 1 to n:
        for j from i+1 to n:
            if arr[i] == arr[j]:
                return true

    return false
```

Errores: confusión entre mejor y peor caso, conclusión incorrecta sobre la complejidad real

Propuesta: el peor caso ejecuta ambos loops completamente

Complejidad:  $O(n^2)$

¿Se encontraron todos los errores?

Si, el problema era puramente conceptual

Codigo 3:

### CÓDIGO 3:

text

 Copy code

```
function mergeAndSort(a, b):
    result = empty list

    for x in a:
        result.append(x)

    for y in b:
        result.append(y)

    sort(result)

    return result
```

Errores: calculo incorrecto del costo de sort, eleccion incorrecta del termino dominante

Correccion: copiar  $O(n + m)$ , ordenar  $O(n+m) \log(n+m)$

Complejidad:  $O((n+m)\log(n+m))$

¿Se encontraron todos los errores?

Si ambos errores estan relacionados pero son distintos

Codigo 4:

#### CÓDIGO 4:

text

 Copy code

```
function searchMatrix(matrix, target):  
    for each row in matrix:  
        if binarySearch(row, target):  
            return true  
    return false
```

Errores: error grave de composicion de complejidades, perdida del factor multiplicativo

Correccion: loop externo  $O(n)$ , binary search interno  $O(\log n)$

Complejidad:  $O(n \log n)$

¿Se encontraron todos los errores?

Si no hay problemas adicionales

Porcentaje de errores detectados=100%

Categorizacion: Los errores del caso 1 y 4 al ser errores conceptuales

## 8 Simulación Empírica

n	Algoritmo A	Razon A	Algortimo B	Razon B	Algoritmo C	Razon C
100	0.9	-	1.2	-	1	-
500	4.8	5.3	29.5	24.6	5.9	5.9
1000	9.7	2.0	118	4	13.2	2.2
2000	19.5	2.0	470	4	28	2.1
5000	49	2.5	2950	6.3	82	2.9
10000	98.5	2.0	11850	4	180	2.2

- Algoritmo A: Las razones se mantienen cercanas a 2 cuando n se duplica → confirma  $O(n)$
- Algoritmo B: Las razones rondan 4 cuando n se duplica → confirma  $O(n^2)$
- Algoritmo C: Las razones están entre 2 y 3 → confirma  $O(n \log n)$ , porque el crecimiento es mayor que lineal pero menor que cuadrático

Patrón Esperado:

- $O(n)$ : Al duplicar n, el tiempo debería multiplicarse por ~2.
- $O(n^2)$ : Al duplicar n, el tiempo debería multiplicarse por ~4.
- $O(n \log n)$ : Al duplicar n, el tiempo debería multiplicarse por algo entre 2 y 3 (dependiendo del log).

## 9 Explorador de Fronteras: Análisis Amortizado

El análisis amortizado es una técnica para evaluar el costo real de una operación a lo largo de una secuencia, en lugar de juzgarla solo por su peor caso individual. A diferencia del análisis clásico de Big-O (peor caso), el análisis amortizado reconoce que algunas operaciones costosas ocurren muy raramente y que su costo puede “repartirse” entre muchas operaciones baratas

La motivación principal surge porque el peor caso puede ser engañoso. Por ejemplo, en un arreglo dinámico, insertar un elemento puede costar  $O(n)$  cuando el arreglo se redimensiona y copia todos los elementos. Sin embargo, ese evento ocurre solo cuando la capacidad se llena. La mayoría de las inserciones cuestan  $O(1)$ . Analizar solo el peor caso da una visión demasiado pesimista del comportamiento real

El análisis amortizado responde a la pregunta:

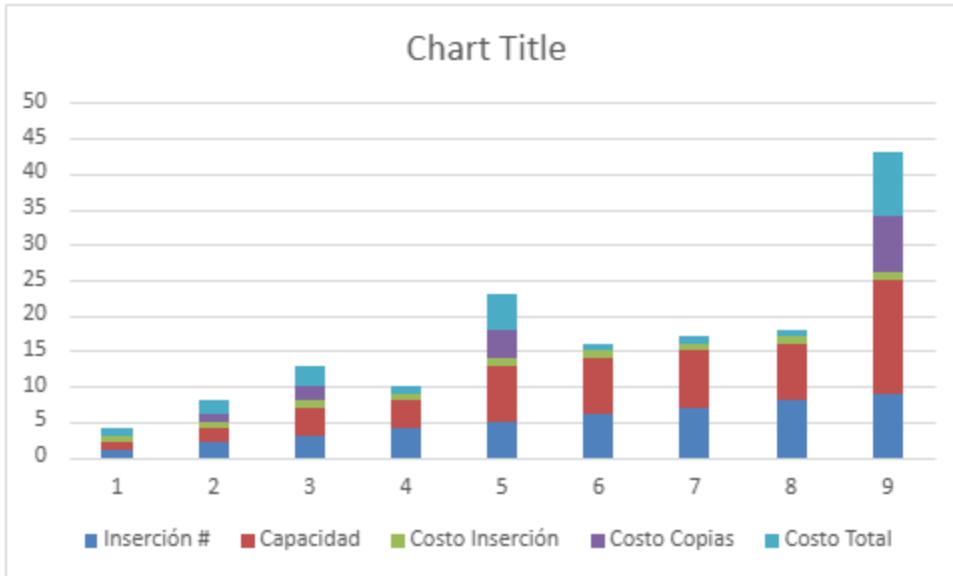
“Si ejecuto muchas operaciones, ¿cuál es el costo promedio garantizado por operación?”

Importante: no usa probabilidad. No es “promedio esperado”, sino una garantía matemática sobre una secuencia completa

Un ejemplo clásico es el arreglo dinámico (como `ArrayList` en Java o `list` en Python). Cuando el arreglo se llena, se crea uno nuevo (normalmente del doble de tamaño) y se copian todos los elementos. Aunque una inserción específica puede costar  $O(n)$ , el costo total de copiar elementos a lo largo de  $n$  inserciones es menor que  $2n$ . Sumando también las inserciones normales, el costo total es proporcional a  $n$ , lo que implica que el costo amortizado por inserción es  $O(1)$

Otro ejemplo es el contador binario. Incrementar un número como 0111 requiere cambiar muchos bits (peor caso  $O(n)$ ), pero la mayoría de los incrementos solo cambian un bit. A lo largo de muchas operaciones, cada bit cambia un número limitado de veces, y el costo total crece linealmente. De nuevo, el costo amortizado por operación es  $O(1)$

En la práctica, el análisis amortizado permite confiar en estructuras de datos que serían descartadas si solo se mirara el peor caso. Nos ayuda a entender el costo real y sostenido de un sistema, que es lo que importa en software profesional



Antes de entender el análisis amortizado, es natural pensar:

“Si una operación puede costar  $O(n)$ , entonces es peligrosa.”

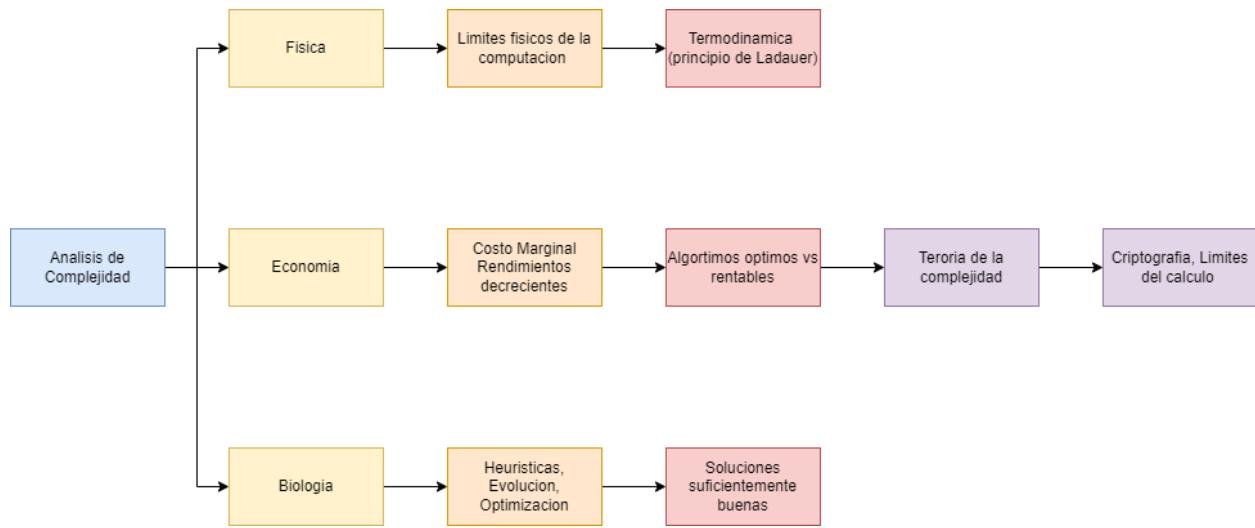
Después de entenderlo, la pregunta cambia a:

“¿Con qué frecuencia ocurre ese costo alto y cómo se distribuye en el tiempo?”

Esto cambia profundamente la forma de evaluar estructuras de datos:

- Dejas de temer operaciones caras **raras**
- Empiezas a pensar en **secuencias**, no en casos aislados
- Entiendes por qué muchas estructuras usadas en la industria son rápidas **en la práctica**, aunque su peor caso sea alto

## 10 Conexiones Interdisciplinarias



Conexion que mas me impacto:

La conexión con la biología es especialmente impactante porque rompe una intuición muy común: que la naturaleza “encuentra la mejor solución” en realidad, los sistemas biológicos operan bajo restricciones extremas de tiempo, energía y recursos, muy similares a las de un sistema computacional la evolución no busca soluciones óptimas en el sentido algorítmico, sino soluciones suficientemente buenas que funcionen en la práctica esto es profundamente análogo al uso de heurísticas en informática cuando un problema es demasiado costoso de resolver exactamente (por ejemplo, problemas NP-difíciles)

Desde el plegamiento de proteínas hasta el comportamiento colectivo de hormigas, la naturaleza evita explosiones combinatorias aceptando aproximaciones. Esto cambia la forma en que vemos la eficiencia: no como perfección, sino como viabilidad. Entender esto ayuda a diseñar algoritmos más realistas y robustos, inspirados no en ideales matemáticos, sino en sistemas que han sobrevivido millones de años.

¿Qué principios de eficiencia usados por sistemas biológicos podrían formalizarse como modelos algorítmicos para resolver problemas NP-difíciles de manera aproximada y energéticamente eficiente?