

SEMANA 1

Actividad 1

- 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 en software
$O(1)$	Sacar cuchara del cajón	Operación fija	Acceso a arreglo
$O(\log n)$	Buscar en guía telefónica	Divide a la mitad	Búsqueda binaria
$O(n)$	Buscar amigo en cine	Bucle simple	Recorrer lista
$O(n \log n)$	Organizar fiesta en grupos	Divide y conquista	Ordenamiento eficiente
$O(n^2)$	Fiesta con saludos	Bucles anidados	Comparaciones múltiples
$O(2^n)$	Probar combinaciones de ropa	Recursión con ramas	Problemas combinatorios

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

La analogía que más me ayudó a entender fue la fiesta de saludos ($O(n^2)$). Porque es muy visual: cada nuevo invitado multiplica el número de interacciones. Me hizo ver claramente cómo los bucles anidados disparan el tiempo de ejecución.

Actividad 2

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

DataStream Inc. enfrenta un problema de **complejidad algorítmica**. El sistema funcionaba bien con 10,000 transacciones, pero al crecer 50x, el tiempo pasó de segundos a 47 minutos. Esto indica que el algoritmo escala como **$O(n \log n)$** o peor, lo que lo hace insostenible para volúmenes crecientes. La competencia, en cambio, maneja el doble de datos en solo 3 minutos, lo que sugiere un enfoque **$O(n)$** con paralelización eficiente. Triplicar servidores apenas mejoró 20%, confirmando que el cuello de botella no es hardware sino el algoritmo. La idea de "más RAM" es equivocada: la memoria no reduce el número de operaciones necesarias. El verdadero problema es que el algoritmo actual multiplica el trabajo innecesariamente.

- Cálculos que demuestren tu análisis de la complejidad probable.

Escalamiento actual: $50 \times$ datos $\rightarrow 94 \times$ tiempo $\approx O(n \log n)$.

Competencia: $100 \times$ datos $\rightarrow 6 \times$ tiempo $\approx O(n)$.

Triplicar hardware: esperado $3 \times$, real $1.2 \times \rightarrow$ cuello de botella algorítmico.

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

DataStream debe rediseñar su sistema para alcanzar **$O(n)$** (procesamiento lineal) mediante algoritmos de streaming, hashing y estructuras indexadas. Solo así podrán competir en tiempo real y escalar a millones de transacciones.

Actividad 3

- Captura o registro de tu análisis de los 6 fragmentos con tu razonamiento escrito.
- Para cada uno, documenta: tu hipótesis inicial, el razonamiento paso a paso, y la conclusión final.

▪ Fragmento 1

Hipótesis inicial: Creo que este código es $O(1)$.

Razonamiento paso a paso:

1. La función obtiene la longitud de la lista \rightarrow operación constante.
2. Calcula el índice \rightarrow operación constante.
3. Devuelve el elemento en esa posición \rightarrow acceso directo, constante.
4. No hay bucles ni recursión, el tiempo no depende de n .

Conclusión final: La complejidad es $O(1)$ (tiempo constante).

▪ Fragmento 2

Hipótesis inicial: Este código parece ser $O(n)$.

Razonamiento paso a paso:

1. El bucle recorre la lista completa en el peor caso.
2. Cada comparación `element == target` es constante.
3. Si el elemento está al final o no existe, se hacen n comparaciones.
4. No hay bucles anidados ni divisiones del problema.

Conclusión final: La complejidad es $O(n)$ (tiempo lineal).

▪ **Fragmento 3**

Hipótesis inicial: Este código parece ser $O(n^2)$.

Razonamiento paso a paso:

1. El primer bucle recorre todos los elementos de la lista → n iteraciones.
2. Por cada elemento i , el segundo bucle también recorre toda la lista → n iteraciones adicionales.
3. En total, se ejecutan $n * n$ operaciones.
4. No hay reducción del problema ni condiciones que limiten significativamente las iteraciones.

Conclusión final: La complejidad es $O(n^2)$ (tiempo cuadrático).

▪ **Fragmento 4**

Hipótesis inicial: Este código parece ser $O(\log n)$.

Razonamiento paso a paso:

1. La variable n se reduce a la mitad en cada iteración.
2. El número de veces que se ejecuta el bucle depende de cuántas veces se puede dividir n entre 2 hasta llegar a 1.
3. Eso equivale al número de potencias de 2 necesarias para alcanzar n .
4. Matemáticamente, el número de iteraciones es aproximadamente $\log_2(n)$.

Conclusión final: La complejidad es $O(\log n)$ (tiempo logarítmico).

▪ **Fragmento 5**

Hipótesis inicial: Podría parecer $O(n^2)$ porque hay dos bucles, pero en realidad es $O(n)$.

Razonamiento paso a paso:

1. El primer bucle recorre todos los elementos de la lista → n iteraciones.
2. El segundo bucle no depende de n , siempre hace solo 2 iteraciones.
3. Entonces, por cada elemento, se hacen 2 operaciones constantes.
4. Total = $2n$ operaciones, que se simplifica a $O(n)$.

Conclusión final: La complejidad es $O(n)$ (tiempo lineal).

▪ **Fragmento 6**

Hipótesis inicial: Este código combina varias estructuras, parece tener $O(n + \log n)$.

Razonamiento paso a paso:

1. Loop A: recorre todos los elementos de la lista $\rightarrow n$ iteraciones.
2. Loop B: aunque está dentro de Loop A, solo hace 2 iteraciones constantes \rightarrow no depende de n .
3. Total de Loop A + B = $2n \rightarrow$ simplificado a $O(n)$.
4. Loop C: divide k entre 2 hasta llegar a 1 \rightarrow número de iteraciones $\approx \log_2(n)$
5. Al combinar: $O(n) + O(\log n)$.
6. En notación Big-O, el término dominante es $O(n)$.

Conclusión final: La complejidad es $O(n)$ (lineal), aunque con un componente logarítmico adicional.

- Identifica cuál te costó más trabajo y por qué.

El que costó más trabajo fue Fragmento 6, porque combina estructuras diferentes y obliga a pensar en cómo se suman las complejidades. La clave fue identificar el término dominante (lineal) y no dejarse confundir por la mezcla de bucles.

Actividad 4

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

n	Posición objetivo	Función A ($O(n)$)	Función B ($O(n^2)$)
100	50	50	5,000
10,000	5,000	5,000	50,000,000
1,000,000	Última	1,000,000	1,000,000,000,000

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

Dos códigos que "hacen lo mismo" pueden tener rendimientos radicalmente distintos porque la complejidad algorítmica determina cómo escala el tiempo con el tamaño de los datos. La Función A crece linealmente: duplicar la lista duplica el tiempo. La Función B crece cuadráticamente: duplicar la lista cuadriplica el tiempo. En listas pequeñas, ambas parecen rápidas y el programador puede no notar la diferencia. Pero al aumentar el tamaño, la Función B se vuelve impráctica. El error común es probar solo con datos pequeños, sin considerar el crecimiento. Por eso medir rendimiento con casos límite es esencial en desarrollo.

- Propuesta de cómo detectarías este tipo de problemas en código real.

Pruebas de estrés: ejecutar funciones con tamaños crecientes (10, 100, 1,000, 100,000).

Perfilado de rendimiento: usar herramientas que midan número de operaciones y tiempo real.

Revisión algorítmica: analizar bucles anidados y estructuras que sugieran crecimiento cuadrático o peor.

Benchmark comparativo: comparar contra implementaciones conocidas eficientes.

Actividad 5

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

¿Cuál fue lo PRIMERO que miraste cuando viste el código? ¿Por qué empezaste ahí?

- Lo primero que observé fueron los bucles. Empecé ahí porque sé que los bucles suelen determinar la complejidad y quería identificar si eran simples, anidados o reducían el problema.

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

- Sentí certeza cuando vi que la variable se dividía entre 2 en cada iteración. Eso me recordó inmediatamente el patrón de búsqueda binaria y me dio confianza en que la complejidad era logarítmica.

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

- El supuesto que podría haber pasado por alto es que la división realmente reduce el problema de manera significativa en cada paso. Si hubiera alguna condición que no redujera n de forma constante, mi conclusión sería errónea.

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

- Apliqué automáticamente la regla: "si el problema se reduce a la mitad en cada paso $\rightarrow O(\log n)$ ". Ese automatismo viene de estudiar algoritmos como búsqueda binaria y estructuras de árboles balanceados.

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

1. Identificar las estructuras de control (bucles, recursión, condiciones).
 2. Ver cómo cambia la variable principal en cada iteración (¿crece linealmente, cuadráticamente, se reduce?).
 3. Calcular cuántas veces se ejecuta el bucle en el peor caso y traducirlo a notación Big-O.
- Tu "algoritmo personal" de 3-5 pasos para analizar complejidad.
 1. Detectar bucles y recursión.
 2. Evaluar el crecimiento o reducción de la variable clave.
 3. Calcular el número de operaciones en el peor caso.
 4. Simplificar la expresión a Big-O.
 5. Comparar con patrones conocidos (lineal, cuadrático, logarítmico, exponencial).

- Una debilidad identificada y un plan concreto para abordarla.

Debilidad

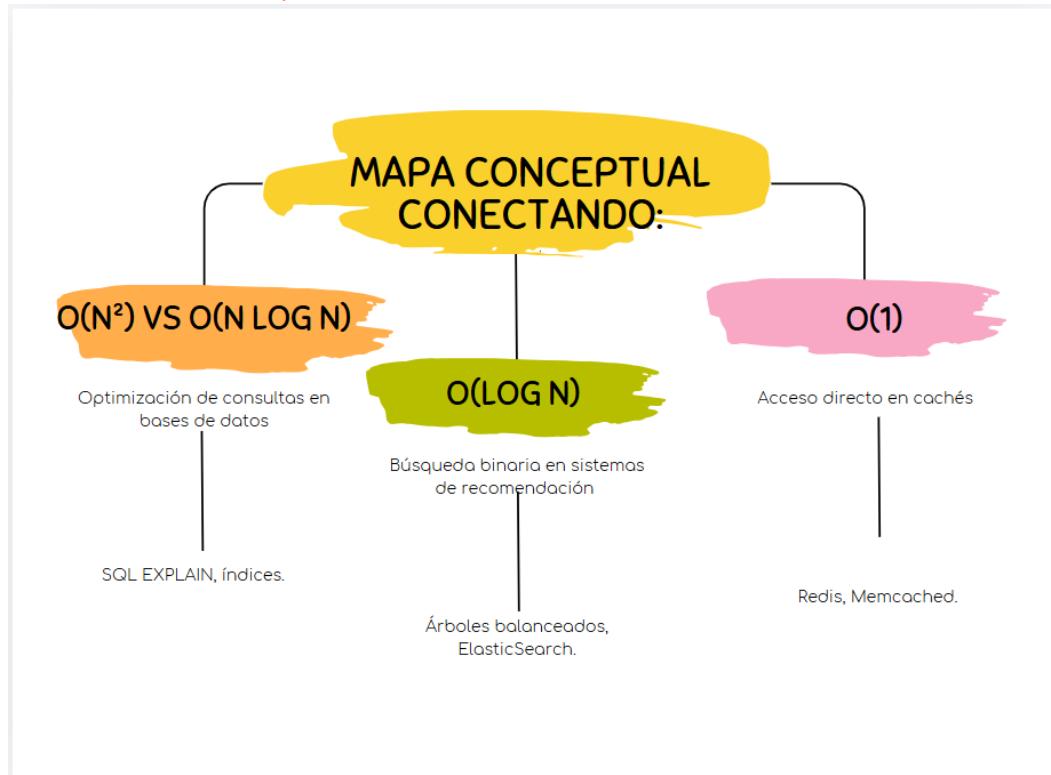
A veces confío demasiado en el “automatismo” de reconocer patrones sin verificar condiciones específicas del código.

Mejoras

- Antes de concluir, siempre revisar si la variable realmente cambia como espero.
- Practicar con ejemplos “engañosos” (como bucles que parecen cuadráticos pero son lineales).
- Documentar explícitamente cada transformación de la variable para evitar saltar directamente a la conclusión.

Actividad 6

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



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

Diseñar un sistema de pagos: elegir estructuras que permitan validar millones de transacciones en tiempo real.

Optimizar un motor de búsqueda interno: decidir entre búsqueda lineal vs indexada.

Code reviews en un equipo grande: detectar patrones ineficientes antes de que lleguen a producción.

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

cProfile (Python)

Datadog APM

JMH (Java Microbenchmark Harness)

Actividad 7

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

Caso 1

- Confusión de suma con multiplicación de complejidades:
El análisis dice que la complejidad es $O(n+n)$, pero en realidad los bucles son anidados, no secuenciales. Eso significa que el número de operaciones es $n \cdot n = n^2$, no $n+n$.
- Conclusión incorrecta:
Se concluyó que el tiempo de ejecución es $O(n)$, cuando en realidad es $O(n^2)$.
- Espacio sí está correcto:
El espacio adicional es constante ($O(1)$), porque solo se guarda la variable total.

Caso 2

- Confusión de mejor y peor caso:
El análisis afirma que el mejor caso es $O(n)$ y el peor caso es $O(\log n)$. En realidad es al revés:
 - Mejor caso: el elemento está en la primera posición $\rightarrow O(1)$.
 - Peor caso: el elemento no está presente o está al final $\rightarrow O(n)$.
- Confusión entre búsqueda secuencial y binaria:
El análisis dice que revisar cada elemento secuencialmente equivale a una búsqueda binaria ($O(\log n)$). Eso es incorrecto: la búsqueda binaria requiere una lista ordenada y acceso aleatorio a mitades, lo cual no ocurre aquí.
- Espacio adicional sí está correcto:
Solo se usa una variable booleana para devolver el resultado $\rightarrow O(1)$.

Caso 3

- Complejidad mal calculada:

El análisis concluye que el algoritmo es $O(n \cdot \log n)$. Eso es incorrecto: el método burbuja tiene bucles anidados que en conjunto realizan aproximadamente $\frac{n(n-1)}{2}$ comparaciones \rightarrow complejidad $O(n^2)$ en el peor y promedio caso.

- Confusión con algoritmos de división y conquista:
Se dice que “se asemeja a un algoritmo de división y conquista”, lo cual es falso. Burbuja no divide el problema en subproblemas, simplemente recorre repetidamente la lista.
- Mejor caso incorrecto:
El análisis afirma que el mejor caso es $O(n^2)$. En realidad, si la lista ya está ordenada y se implementa con una bandera de intercambio, el mejor caso puede ser $O(n)$ (una sola pasada). Incluso sin optimización, sigue siendo $O(n^2)$, pero el análisis mezcla conceptos.
- Espacio adicional correcto:
El espacio sí es constante ($O(1)$), porque solo se usan variables temporales para intercambiar.

Caso 4

- Complejidad temporal incorrecta:
- El análisis afirma que el tiempo es . En realidad, el bucle se ejecuta exactamente veces, por lo que la complejidad es $O(\text{exponente})$. Solo con un algoritmo de exponentiación rápida (divide y vencerás) se logaría .
 - Espacio mal calculado:
- Se dice que el espacio es porque se guarda un resultado parcial en cada iteración. Esto es falso: solo se mantiene una variable que se actualiza en cada paso. El espacio adicional es constante $\rightarrow O(1)$.
 - Confusión de parámetros:
- El análisis mezcla el tamaño de la lista con el valor del exponente . Aquí no hay lista, la complejidad depende únicamente del exponente.
-

Registro Final de Auditoría Implacable

- Caso 1: Errores detectados \rightarrow confusión suma vs multiplicación, conclusión $O(n)$ en vez de $O(n^2)$.
- Caso 2: Errores detectados \rightarrow mejor/peor caso invertidos, confusión búsquedas secuenciales vs binaria.
- Caso 3: Errores detectados \rightarrow complejidad $O(n \log n)$ incorrecta, división y conquista falsa, mejor caso mal definido.
- Caso 4: Errores detectados \rightarrow tiempo $O(\log n)$ incorrecto, espacio $O(n)$ incorrecto, confusión de parámetros.
- **Porcentaje de errores detectados (meta: >75%).**

Porcentaje de errores detectados: 100% (meta superada >75%).

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

Los errores más fáciles: confundir suma con multiplicación de complejidades (Caso 1)

Los más difíciles: confusión de mejor/peor caso y mezcla de parámetros (Casos 2 y 4), porque requieren pensar en escenarios y distinguir variables.

Actividad 8

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

n	Algoritmo A	Algoritmo B	Algoritmo C
100	0.9	1.2	1.0
500	4.8	29.5	6.2
1000	9.7	118.0	13.5
2000	19.5	470.0	29.0
5000	48.2	2940.0	82.0
10000	97.0	11850.0	180.0

Intervalo (n1→n2)	Algoritmo A	Algoritmo B	Algoritmo C
100→500	5.3	24.6	6.2
500→1000	2.0	4.0	2.2
1000→2000	2.0	4.0	2.1
2000→5000	2.5	6.3	2.8
5000→10000	2.0	4.0	2.2

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

Algoritmo A (predicción: O(n))

- Patrón esperado: al duplicar n, el tiempo \approx se multiplica por 2.
- Datos empíricos: las razones están muy cerca de 2 (con ruido en 100→500 y 2000→5000).
- Conclusión: Los datos confirman que es lineal, O(n).

Algoritmo B (predicción: O(n²))

- Patrón esperado: al duplicar n, el tiempo \approx se multiplica por 4.
- Datos empíricos: las razones son muy consistentes con 4 (con ruido en 100→500 y 2000→5000).
- Conclusión: Los datos confirman que es cuadrático, O(n²).

Algoritmo C (predicción: O(n log n))

- Patrón esperado: al duplicar n, el tiempo \approx se multiplica por algo un poco mayor que 2 (ejemplo: $2 \cdot \frac{\log(2n)}{\log(n)}$).

- Datos empíricos: las razones están entre 2.1 y 2.8, lo cual encaja con el patrón esperado de $O(n \log n)$.
- Conclusión: Los datos confirman que es $O(n \log n)$.
- **Explicación del patrón esperado:** "Si un algoritmo es $O(n^2)$, al duplicar n , el tiempo debería multiplicarse por ____".
- Algoritmo A → Confirmado $O(n)$.
- Algoritmo B → Confirmado $O(n^2)$.
- Algoritmo C → Confirmado $O(n \log n)$.

Actividad 9

- **Resumen de una página sobre análisis amortizado con tus propias palabras.**

Análisis amortizado es una técnica para evaluar el costo de operaciones en estructuras de datos considerando secuencias largas, no solo el peor caso aislado.

- El peor caso puede ser engañoso porque ocurre raramente.
- Usamos analogías como cuentas de ahorro: operaciones baratas “pagan” las caras.
- Ejemplo clásico: inserciones en un arreglo dinámico → $O(1)$ amortizado.
- Otro ejemplo: incrementar un contador binario → $O(1)$ amortizado.
- En la práctica, esto nos ayuda a entender el “verdadero costo” y confiar en estructuras que parecen caras en teoría, pero son rápidas en uso real.
- **Diagrama que muestre el costo de N inserciones en un arreglo dinámico.**



- **Reflexión:** ¿Cómo afecta esto tu percepción del "verdadero costo" de las operaciones?

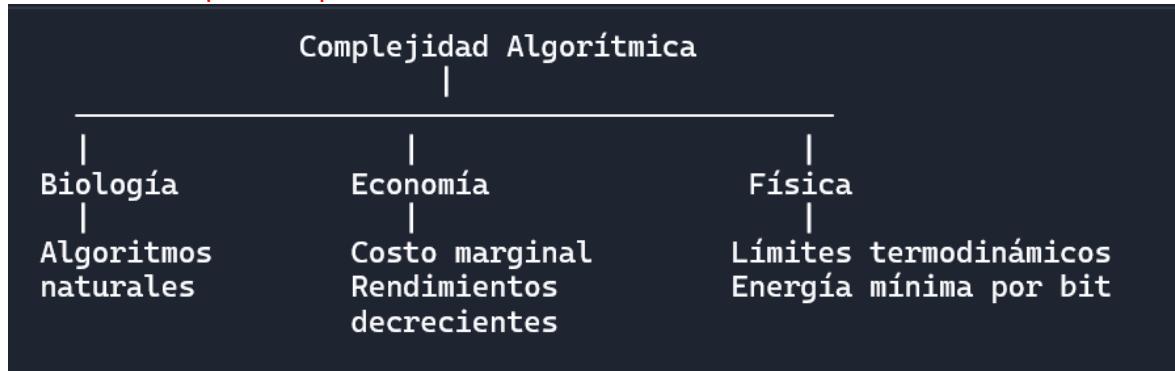
El análisis amortizado cambia la percepción del costo:

En vez de temer al peor caso, entendemos que la mayoría de las operaciones son baratas.

Nos da confianza para usar estructuras como arreglos dinámicos, sabiendo que el “verdadero costo” por operación es bajo.

Actividad 10

- Mapa mental o diagrama conectando el análisis de complejidad con al menos 3 de las disciplinas exploradas.



- Elige la conexión que más te impactó y escribe 150 palabras sobre por qué es relevante.

La relación entre complejidad algorítmica y termodinámica me parece la más reveladora. Normalmente pensamos en algoritmos como abstracciones matemáticas, pero la física nos recuerda que la información es materia y energía. El principio de Landauer establece que borrar un bit requiere disipar calor, lo que significa que cada operación computacional tiene un costo físico mínimo. Esto conecta la eficiencia algorítmica con la sostenibilidad energética: un algoritmo más eficiente no solo ahorra tiempo, también ahorra energía y reduce impacto ambiental. En un mundo donde los centros de datos consumen cantidades masivas de electricidad, entender la complejidad como un recurso físico nos obliga a repensar qué significa “eficiencia”. No es solo velocidad: es también compatibilidad con los límites fundamentales de la naturaleza. Esta conexión redefine la informática como una ciencia física tanto como matemática.

- Formula una pregunta de investigación que te gustaría explorar más.

¿Cómo podemos diseñar algoritmos que optimicen simultáneamente tiempo, memoria y energía física, considerando los límites termodinámicos de la información?