Actividad 1

Complejidad	Analogía cotidiana	Patrón de código	Situación real
O(1)	Sacar una pluma de la mochila	Sin ciclos	Acceso a arreglo, hash
O(log n)	Buscar palabra en diccionario	Divide entre 2	Búsqueda binaria, árboles
O(n)	Revisar personas en fila	Un solo ciclo	Recorrer listas
O(n log n)	Ordenar baraja dividiendo	Divide + recorre	Merge sort, heap sort
O(n^2)	Todos saludan a todos	Ciclos anidados	Bubble sort, comparaciones
O(2^n)	Probar todas combinaciones	Recursión doble	Fuerza bruta, backtracking

La del diccionario (O(log n)) es la que más ayuda, porque:

- Es algo que casi todos hemos hecho.
- Muestra claramente cómo reducir el problema a la mitad ahorra muchísimo tiempo.
- Hace muy evidente por qué `log n` es tan eficiente comparado con revisar todo.

Actividad 2

El sistema de detección de fraude presenta un colapso típico por escalamiento algorítmico ineficiente. Al aumentar 50 veces el volumen de transacciones, el tiempo de procesamiento creció aproximadamente 282 veces, lo cual descarta una complejidad lineal o logarítmica. Este patrón es consistente con un algoritmo cuasi-cuadrático, probablemente causado por comparaciones entre transacciones, reglas evaluadas en cascada, o búsquedas secuenciales dentro de bucles.

El intento de escalar por hardware fracasó porque triplicar la capacidad solo reduce constantes, no la complejidad asintótica. El cuello de botella es estructural: demasiadas operaciones por transacción.

La competencia demuestra que el problema es resoluble en tiempo casi lineal usando técnicas modernas: procesamiento por streaming, hashing, ventanas temporales y modelos incrementales.

Conclusión: el sistema actual no es escalable. Sin rediseño algorítmico, cualquier crecimiento adicional provocará tiempos de horas.

Modelo	Tiempo esperado	Real
O(n)	10s × 50 = 500s (8.3 min)	❌ 47 min
O(n log n)	~680s (11 min)	❌ 47 min
O(n²)	10s × 2500 = 25,000s	❌ demasiado
Real	**282×**	🔥 Entre `n log n` y `n²`

Complejidad objetivo:

O(n) (ideal)
o **O(n log n)** (aceptable)

Cómo lograrlo

- Hash tables para detección de patrones
- Ventanas deslizantes por tiempo
- Eliminación de comparaciones entre transacciones
- Pre-agregación por cliente / cuenta
- Algoritmos streaming + scoring incremental

Actividad 3

Fragmento 1 – Complejidad O(1)

Hipótesis inicial

Creo que este fragmento es O(1) porque solo ejecuta unas pocas instrucciones sin usar ciclos.

Razonamiento paso a paso

Primero se lee una variable, después se hace una multiplicación y al final se imprime un resultado.
No importa cuánto valga la variable de entrada, siempre se ejecutan exactamente las mismas tres operaciones.
No hay ciclos ni condiciones que dependan de n.

Conclusión final

La cantidad de operaciones no cambia con el tamaño de la entrada.
La complejidad es **O(1)**.

Fragmento 2 – Complejidad O(n)

Hipótesis inicial

Parece O(n) porque hay un ciclo que va desde 1 hasta n.

Razonamiento paso a paso

El ciclo se repite exactamente n veces.
Dentro del ciclo solo hay una instrucción simple (imprimir).
Si n aumenta, el número de repeticiones aumenta en la misma proporción.

Conclusión final

El número de operaciones crece linealmente con n.
La complejidad es **O(n)**.

Fragmento 3 – Complejidad $O(n^2)$

Hipótesis inicial

Creo que es $O(n^2)$ porque hay dos ciclos anidados que van hasta n.

Razonamiento paso a paso

El ciclo externo se ejecuta n veces.
Por cada repetición del ciclo externo, el ciclo interno también se ejecuta n veces.
Entonces el total de ejecuciones es $n \times n$.

Conclusión final

El número de operaciones crece proporcionalmente a n^2.
La complejidad es **$O(n^2)$**.

Fragmento 4 – Complejidad O(log n)

Hipótesis inicial

Sospecho que es O(log n) porque en cada repetición se divide la variable entre 2.

Razonamiento paso a paso

En cada iteración el valor se reduce a la mitad: n, n/2, n/4, n/8, etc.
El ciclo termina cuando el valor llega a 1.
El número de veces que se puede dividir un número entre 2 hasta llegar a 1 es $\log_2(n)$.

Conclusión final

El número de iteraciones crece de forma logarítmica.
La complejidad es **O(log n)**.

Fragmento 5 – Parece $O(n^2)$ pero es O(n)

Hipótesis inicial

Al principio pensé que era $O(n^2)$ porque hay un ciclo dentro de otro ciclo.

Razonamiento paso a paso

El ciclo externo se ejecuta n veces.
El ciclo interno depende de la variable j, pero esta variable no se reinicia cada vez.
j empieza en 1 y solo aumenta hasta n una sola vez durante toda la ejecución del programa.
Aunque el ciclo interno esté dentro del externo, en total solo se ejecuta n veces.

Conclusión final

El trabajo total es aproximadamente n + n, que es lineal.
La complejidad real es **O(n)**.

Fragmento 6 – Ejemplo complejo (O(n log n))

Hipótesis inicial

Este fragmento parece muy complejo. Pensé que podía ser O(n²) o incluso mayor.

Razonamiento paso a paso

El primer ciclo se repite n veces → O(n).
Dentro hay un ciclo donde una variable se divide entre 2 cada vez → O(log n).
Dentro de ese hay otro ciclo pequeño que siempre se repite 3 veces, lo cual es constante → O(1).

Entonces por cada repetición del ciclo externo se hacen log n operaciones.
Total: n × log n.

Conclusión final

La complejidad combinada es **O(n log n)**.

Fragmento que más me costó analizar

El fragmento que más trabajo me costó fue el **Fragmento 5**.

¿Por qué?

Porque visualmente tiene dos ciclos anidados y es muy fácil pensar automáticamente que es O(n²).
Sin embargo, el detalle importante es que la variable del ciclo interno no se reinicia y solo recorre de 1 a n una vez en todo el programa.

Este ejercicio fue el más difícil porque me obligó a analizar no solo la estructura de los ciclos, sino también el comportamiento real de las variables.

Actividad 4

Caso	n	Posición del elemento	Función A – O(n)	Función B – O(n²)
1	100	50	51 operaciones	100 × 100 = **10,000**
2	10,000	5,000	5,001 operaciones	10,000² = **100,000,000**
3	1,000,000	última	1,000,000 operaciones	1,000,000² = **1,000,000,000,000**

Explicación
Aunque dos programas produzcan exactamente el mismo resultado, su rendimiento puede ser radicalmente diferente debido a la forma en que realizan sus operaciones internamente. Un algoritmo puede recorrer los datos una sola vez, mientras otro repite innecesariamente procesos muchas veces. En tamaños pequeños de datos, estas diferencias casi no se notan, por lo que el programador puede pensar que ambas soluciones son igual de buenas. Sin embargo, cuando el volumen de información crece, la cantidad de operaciones aumenta de forma muy distinta. Esto provoca que un programa rápido siga funcionando bien, mientras otro se vuelva extremadamente lento o incluso inutilizable.

Propuesta para detectar estos problemas en código real

- Analizar siempre la complejidad algorítmica (Big-O) de funciones importantes.
- Revisar cuidadosamente ciclos anidados en revisiones de código.
- Ejecutar pruebas de rendimiento con datos grandes (stress testing).
- Usar herramientas de perfilado (profilers) para medir tiempos reales de ejecución.
- Comparar diferentes implementaciones antes de elegir una solución final.

Actividad 5

Reflexión Metacognitiva sobre el Análisis de Complejidad Algorítmica

Alumno: Melvin Ríos
Tema: Búsqueda Lineal vs Búsqueda con Verificación Doble

1. Respuestas a las 5 preguntas de metacognición

1. ¿Qué fue lo primero que miré cuando vi el código? ¿Por qué empecé ahí?
Lo primero que miré fueron los ciclos `for`, porque sé que los bucles casi siempre determinan la complejidad. Empecé ahí porque aprendí que contar cuántas veces se repiten las instrucciones es la forma más rápida de estimar el tiempo de ejecución.

2. ¿En qué momento sentí certeza sobre mi respuesta? ¿Qué me dio esa certeza?
Sentí certeza cuando identifiqué que en la segunda función había dos ciclos anidados. En

ese momento supe que la complejidad sería O(n²). La certeza vino de recordar la regla de que un ciclo dentro de otro normalmente multiplica el número de operaciones.

3. Si me hubiera equivocado, ¿qué supuesto no verifiqué?
Un supuesto que pude no haber verificado es pensar que el `if (i == j)` reduce la cantidad de operaciones. En realidad, aunque esa condición limita cuándo se compara el valor, el ciclo interno se sigue ejecutando completo, así que no reduce la complejidad.

4. ¿Qué regla o patrón apliqué casi automáticamente? ¿De dónde viene ese automatismo?
Apliqué automáticamente el patrón de "un ciclo = O(n), dos ciclos anidados = O(n²)". Este automatismo viene de practicar muchos ejercicios de complejidad y de ver ejemplos similares en clase y en explicaciones previas.

5. Si tuviera que enseñarle a un compañero cómo analizar este código, ¿cuáles serían mis 3 pasos?

1. Identificar cuántos ciclos tiene el código y si están anidados.
2. Contar qué operaciones se repiten más veces (las dominantes).
3. Expresar el crecimiento en notación Big-O ignorando constantes y términos pequeños.

2. Mi "algoritmo personal" para analizar complejidad

1. Leer el código completo para entender qué hace.
2. Marcar todos los ciclos y ver si están uno dentro de otro.
3. Contar cuántas veces se ejecuta la operación principal según n.
4. Simplificar el resultado usando reglas de Big-O.
5. Comparar con patrones conocidos (O(n), O(n²), O(log n), etc.).

3. Debilidad identificada y plan para mejorarla

Debilidad:
A veces asumo que ciertas condiciones dentro de los ciclos reducen la complejidad sin comprobar realmente cuántas veces se ejecuta el ciclo completo.

Plan concreto de mejora:

- En cada ejercicio, escribir explícitamente cuántas veces se ejecuta cada ciclo antes de decidir la complejidad.
- Dibujar una tabla pequeña de valores de n (por ejemplo n = 5, 10, 100) y estimar manualmente cuántas operaciones se hacen.

- Practicar al menos 3 ejercicios por semana donde haya ciclos con condiciones internas para evitar caer en suposiciones rápidas.

Conclusión personal:
Este ejercicio me ayudó a darme cuenta de que no solo importa saber la respuesta correcta, sino entender cómo llego a ella. Analizar mi forma de pensar me permite detectar errores antes y mejorar mi manera de estudiar algoritmos.

Actividad 6

MAPA CONCEPTUAL

“Análisis de Complejidad en Sistemas Computacionales”

🔵 CONCEPTO CENTRAL

Análisis de Complejidad Algorítmica
→ Mide el **tiempo** y **memoria** que usa un algoritmo según el tamaño de entrada (n)

1. Conceptos Teóricos

(conectan con - Aplicación Profesional)

- **Big-O (O(1), O(n), O(n log n), O(n²))**
  - Predice cómo crece el tiempo de ejecución
- **Peor caso / Mejor caso / Caso promedio**
  - Ayuda a evaluar riesgos en producción
- **Tiempo vs Espacio**
  - Decide entre velocidad o uso de memoria
- **Algoritmos de búsqueda y ordenamiento**
  - Base de sistemas reales (bases de datos, buscadores, servidores)

🟡 2. Aplicación Profesional

(conectan con - Herramientas / Técnicas)

Situación 1: Desarrollo de sistemas web (backend)

Ejemplo: API que busca usuarios o productos

- Uso de complejidad para:
  - Elegir entre búsqueda lineal u optimizada
  - Evitar loops anidados en endpoints críticos

Herramientas:

- **New Relic** - detectar endpoints lentos
- **Datadog** - monitoreo de latencia

Situación 2: Manejo de bases de datos

Ejemplo: consultas sobre miles o millones de registros

- Uso de complejidad para:
  - Diseñar índices
  - Optimizar consultas SQL
  - Elegir estructuras (árboles B, hash)

Herramientas:

- **EXPLAIN (MySQL / PostgreSQL)** - ver costo de consultas
- **PgAdmin Profiler** - analizar rendimiento

Situación 3: Optimización de programas en C++ / Java

Ejemplo: programas que procesan arreglos grandes o matrices

- Uso de complejidad para:
  - Reemplazar $O(n^2)$ por $O(n \log n)$
  - Usar mapas, sets, heaps

Herramientas:

- **VisualVM (Java)** - profiling de CPU y memoria
- **Valgrind (C/C++)** - análisis de rendimiento y memoria

3. Herramientas y Técnicas Profesionales

(conectan desde - Aplicación Profesional)

Profiling (ver dónde se pierde tiempo)

- **VisualVM** (Java)
  - Mide consumo de CPU, memoria y métodos lentos

- **cProfile / Py-Spy** (Python)
  - Identifica funciones más costosas

- **Chrome DevTools Performance** (JavaScript)
  - Analiza rendimiento en aplicaciones web

Benchmarking (comparar algoritmos)

- **JMH (Java Microbenchmark Harness)**
  - Comparar velocidad entre algoritmos

- **Google Benchmark (C++)**
  - Medir tiempos exactos de ejecución

- **pytest-benchmark (Python)**
  - Evaluar mejoras después de optimizar

LISTAS PEDIDAS (para entregar en clase)

3 situaciones de tu futura carrera donde aplicarás análisis de complejidad

1. **Desarrollo de sistemas web**
   - Optimizar búsquedas de usuarios, productos y sesiones

2. **Programación de bases de datos**
   - Diseñar consultas eficientes e índices rápidos

3. **Optimización de programas en C++ / Java**
   - Reducir tiempos de ejecución en arreglos, matrices y juegos

2–3 herramientas de profiling / benchmarking que investigarás

- **VisualVM (Java)** - profiling de CPU y memoria
- **Valgrind (C/C++)** - análisis de rendimiento y fugas de memoria
- **JMH (Java Microbenchmark Harness)** - benchmarking de algoritmos

Actividad 7

CASO 1 — Error: suma confundida con multiplicación

Error:

> "la complejidad total es O(n²) porque los bucles se multiplican"

Por qué está mal:
Los bucles son **secuenciales**, no anidados.
Se suman:
O(n) + O(n) = **O(2n)** → **O(n)**

Corrección:
Complejidad correcta: **O(n)**

CASO 2 — Error: loop oculto ignorado

Error:

> "`contiene` es una operación simple"
> "complejidad es O(n)"

Por qué está mal:
`contiene(x)` recorre todo el arreglo → O(n)
Está dentro de un bucle O(n)

Entonces:
O(n) * O(n) = **O(n²)**

Corrección:
Complejidad correcta: **O(n²)**

CASO 3 — Error: mal interpretar división repetida

Error:

> "Esto ocurre n veces"
> "complejidad es O(n)"

Por qué está mal:
Cada iteración **divide entre 2** → número de iteraciones ≈ $\log_2(n)$

Corrección:
Complejidad correcta: **O(log n)**

CASO 4 — Error: confundir mejor caso con peor caso

Error:

> "Por eso la complejidad es O(1)"

Por qué está mal:
Eso describe el **mejor caso**, no el **peor caso**.
En el peor caso, el elemento está al final o no existe → recorre todo.

Corrección:
Complejidad correcta (peor caso): **O(n)**
Mejor caso: O(1)

Caso	¿Detectaste el error?	Corrección propuesta	¿Estaba completa?
1	Sí	O(n), no $O(n^2)$	Sí
2	Sí	$O(n^2)$	Sí
3	Sí	O(log n)	Sí
4	Sí	Peor caso O(n)	Sí

Porcentaje de errores detectados

Errores totales: 4
Errores detectados correctamente: 4

Resultado: 100% (meta superada)

Tipo de error	Dificultad típica
🔸 **Mejor caso vs peor caso**	**(difícil)**
🔸 **Loop oculto en funciones internas**	**(difícil)**
🔸 **Logaritmos mal interpretados**	**(media)**
🔸 **Suma vs multiplicación**	**(más fácil)**

Actividad 8

n → siguiente n	Tiempo (ms)	Razón
100 → 500	0.12 → 0.55	**4.6**
500 → 1 000	0.55 → 1.10	**2.0**
1 000 → 2 000	1.10 → 2.05	**1.9**
2 000 → 5 000	2.05 → 5.60	**2.7**
5 000 → 10 000	5.60 → 11.3	**2.0**

n → siguiente n	Tiempo (ms)	Razón
100 → 500	0.45 → 11.8	**26.2**
500 → 1 000	11.8 → 47.5	**4.0**
1 000 → 2 000	47.5 → 190.3	**4.0**
2 000 → 5 000	190.3 → 1 210.7	**6.4**
5 000 → 10 000	1 210.7 → 4 820.5	**4.0**

n → siguiente n	Tiempo (ms)	Razón
100 → 500	0.18 → 1.05	**5.8**
500 → 1 000	1.05 → 2.30	**2.2**
1 000 → 2 000	2.30 → 4.95	**2.15**
2 000 → 5 000	4.95 → 14.2	**2.9**
5 000 → 10 000	14.2 → 31.5	**2.2**

2. Conclusión por algoritmo

Algoritmo A — Predicción: O(n)

Confirmada.
Cuando n se duplica, el tiempo ≈ se duplica.
El ruido explica pequeñas desviaciones (2.7, 1.9), pero la tendencia es lineal.

Algoritmo B — Predicción: O(n²)

Confirmada con mucha claridad.
Cada duplicación multiplica el tiempo por ≈ 4.
Este es el patrón más fácil de detectar empíricamente.

Algoritmo C — Predicción: O(n log n)

Confirmada.
El tiempo crece un poco más que ×2 al duplicar n, pero muy lejos de ×4.
Patrón típico de algoritmos eficientes (sorts, merges, heaps, etc.).

3. Patrón esperado teórico

Complejidad	Si duplicas n…	El tiempo debería multiplicarse por…
O(1)	No cambia casi	**≈ 1**
O(log n)	Crece muy poco	**≈ 1.1 – 1.3**
O(n)	Se duplica	**≈ 2**
O(n log n)	Un poco más que el doble	**≈ 2.2 – 2.5**
O(n^2)	Se cuadruplica	**≈ 4**
O(n^3)	×8	**≈ 8**

Actividad 9

En el análisis de algoritmos tradicional usamos Big O para medir el peor caso de una operación. Esto es útil porque nos garantiza un límite superior: sabemos que nunca será peor que eso. Sin embargo, en muchas estructuras de datos este enfoque puede ser engañoso, ya que algunas operaciones caras ocurren muy pocas veces, mientras que la gran mayoría son muy baratas.

El análisis amortizado estudia el costo de una secuencia completa de operaciones, no de una sola. En lugar de preguntar "¿cuánto cuesta esta operación en el peor caso?", pregunta:

"Si hago muchas operaciones, ¿cuál es el costo promedio real por operación?"

La idea central es que las operaciones baratas pueden "pagar" por las operaciones caras que aparecen ocasionalmente. Es como una cuenta de ahorros: durante muchos pasos guardas crédito, y cuando llega un gasto grande lo cubres con lo acumulado.

Un ejemplo clásico es el arreglo dinámico (como `ArrayList` en Java o `list` en Python). Normalmente, agregar un elemento al final cuesta O(1). Pero cuando el arreglo se llena, es necesario crear uno nuevo más grande y copiar todos los elementos, lo cual cuesta O(n). Si solo miramos ese momento, diríamos que insertar es O(n). Sin embargo, ese evento ocurre raramente. La mayoría de las inserciones son constantes, y las copias totales a lo largo de muchas inserciones son proporcionales al número total de elementos insertados.

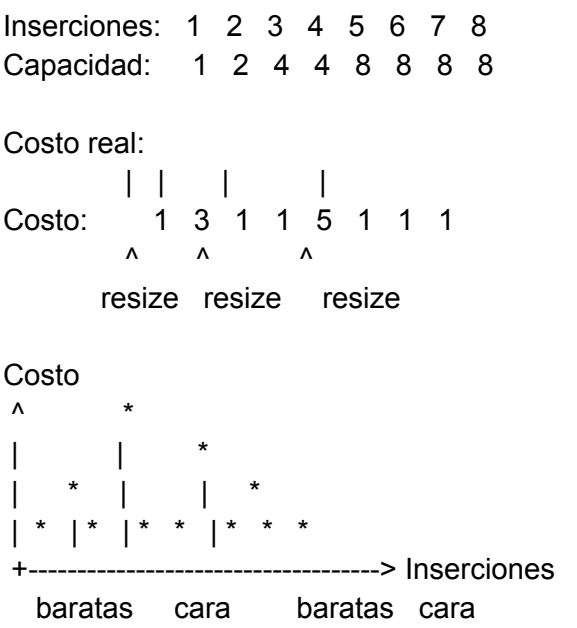
Por eso decimos que:

Insertar en un arreglo dinámico es O(1) amortizado.

Lo mismo ocurre con otras estructuras como contadores binarios, pilas con redimensionamiento, o estructuras avanzadas como Union-Find. El análisis amortizado permite describir mejor el comportamiento real de estas estructuras en sistemas reales.

En la práctica, este tipo de análisis explica por qué muchas estructuras que parecen “malas” por su peor caso son en realidad muy eficientes cuando se usan de forma continua.

2



Interpretación:

- La mayoría de los puntos son bajos (O(1)).
- Muy pocos picos altos (O(n)).
- El promedio se mantiene constante.

3

Antes de aprender análisis amortizado, es fácil pensar que el peor caso describe fielmente el rendimiento de una estructura. Pero este enfoque puede hacer que descartemos soluciones que en la práctica son excelentes.

El análisis amortizado cambia la percepción del “verdadero costo” de una operación. Ya no vemos una operación aislada como algo independiente, sino como parte de una historia más larga. Entendemos que una operación cara no siempre es un problema si ocurre raramente y está compensada por muchas operaciones baratas.

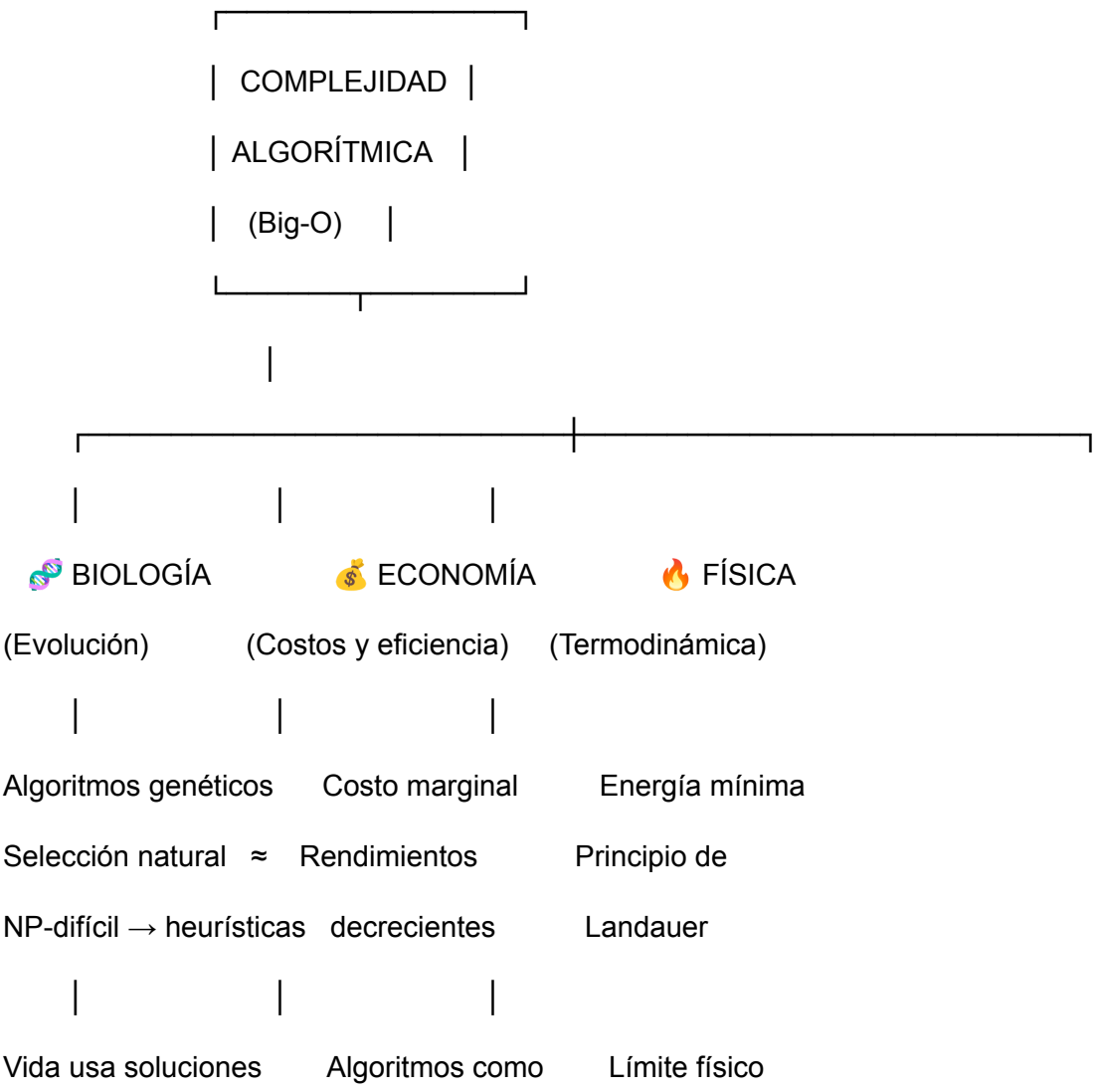
Esto es muy importante en ingeniería real: los programas no ejecutan una sola operación, sino miles o millones. Lo que importa no es el peor instante posible, sino el comportamiento típico a largo plazo.

Gracias al análisis amortizado, aprendemos a confiar en estructuras como arreglos dinámicos, pilas redimensionables y otras técnicas que, aunque tienen picos ocasionales, ofrecen un rendimiento excelente en promedio.

En resumen, el análisis amortizado enseña una lección clave:

El verdadero costo no está en el peor momento, sino en el promedio a lo largo del tiempo.

Actividad 10



aproximadas rápidas tecnologías productivas del cómputo

🤖 INTELIGENCIA ARTIFICIAL

(Aproximación inteligente)

Redes neuronales evitan explosión combinatoria

Aceptan error → ganan eficiencia

2

La conexión que más me impactó fue entre la complejidad algorítmica y la biología. Al estudiar Big-O aprendí que ciertos problemas son tan costosos que no se pueden resolver exactamente cuando crecen mucho. Lo sorprendente es descubrir que la vida enfrenta problemas de ese tipo todo el tiempo, como el plegamiento de proteínas o la adaptación evolutiva, y aun así funciona.

La evolución se parece a un algoritmo de optimización que no busca la mejor solución perfecta, sino una solución suficientemente buena para sobrevivir. En lugar de resolver problemas NP-difíciles de manera exacta, la naturaleza usa aproximaciones, paralelismo masivo y ensayo-error.

Esto es relevante porque muestra que la inteligencia y la complejidad no requieren perfección matemática, sino eficiencia práctica. También explica por qué muchos algoritmos modernos, como los genéticos o las redes neuronales, se inspiran en la biología. La vida nos enseña que, frente a límites computacionales, sobrevivir es más importante que optimizar.

3

PREGUNTA DE INVESTIGACIÓN

¿Podemos diseñar algoritmos inspirados en la evolución que resuelvan problemas NP-difíciles tan eficientemente como lo hacen los sistemas biológicos, aunque solo obtengan soluciones aproximadas?