1. Historia: bug de rendimiento por ignorar complejidad

Contexto
Empresa: *e-commerce mediano* (tipo Mercado Libre / Shopify partner).
Sistema en **Node.js** que calculaba recomendaciones de productos.

El problema
En producción, el endpoint de recomendaciones empezó a tardar **10–15 segundos** cuando había campañas grandes. En staging funcionaba "bien".

Código problemático (simplificado):

```
for (let user of users) {
  for (let product of products) {
    if (user.history.includes(product.id)) {
      score++
    }
  }
}
```

Complejidad real: **$O(n^2)$**
Con 100 usuarios y 100 productos nadie lo notó.
Con **50,000 usuarios × 20,000 productos**, el sistema murió.

Cómo lo descubrimos

- Alertas de latencia (Datadog)
- CPU al 100%
- Profiling mostró que ese doble for consumía ~80% del tiempo

La solución

- Cambiar includes por un Set
- Preprocesar datos
- Reducir a **O(n)**

Resultado:
Latencia bajó de **15s a 120ms**.

Lección profesional:

"Que funcione" no significa "que escale".

2. ¿En qué situaciones reales se estima complejidad?

Durante *code reviews*

Sí, todo el tiempo.
Especialmente cuando veo:

- Ciclos anidados
- Operaciones dentro de loops (sort, filter, includes)
- Queries a base de datos dentro de ciclos

Frase típica en review:

"Esto funciona, pero ¿qué pasa si esto crece 10x?"

Al diseñar sistemas nuevos

Muchísimo.

Ejemplo:

- ¿Lista en memoria o índice?
- ¿Búsqueda lineal o estructura hash?
- ¿Precalcular o calcular bajo demanda?

Antes de escribir código, se pregunta:

"¿Cómo crece esto cuando tengamos millones de registros?"

Al optimizar código existente

Casi siempre llegas aquí **por dolor**:

- Sistema lento
- Costos altos en la nube
- Timeouts

Ahí el análisis de complejidad guía **dónde atacar primero**.

3. ¿Cuándo es aceptable usar $O(n^2)$?

Sí, **a veces es totalmente válido**.

Casos reales donde $O(n^2)$ es aceptable:

- n es **pequeño y fijo** (ej. máximo 50 elementos)
- Código de:
    - Validación
    - Scripts internos
    - Herramientas administrativas
- Cuando **claridad > optimización**
- Cuando optimizar más complica sin beneficio real

Regla práctica en industria:

$O(n^2)$ es malo cuando **n puede crecer sin control**.

4. Complejidad en entrevistas técnicas

Cómo se evalúa realmente

No esperan solo que digas "O(n log n)".

Evalúan:

- Cómo razonas
- Si justificas
- Si detectas cuellos de botella

Pregunta típica

"Dado este código, ¿cuál es su complejidad y cómo la mejorarías?"

Ejemplo esperado de respuesta buena:

- "Este algoritmo es $O(n^2)$ porque por cada elemento recorre toda la lista."
- "Podría reducirlo a O(n) usando una estructura hash."
- "En el peor caso, con entradas grandes, esto sería un problema."

Eso pesa **más** que dar la respuesta exacta rápido.

5. Herramientas reales que usan los profesionales

No nos quedamos solo con Big-O.

Herramientas comunes:

- **Chrome DevTools (Performance / Profiler)**
- **Node.js Profiler**

- **Benchmark.js**
- **Datadog / New Relic**
- **JProfiler / VisualVM** (Java)
- **pprof** (Go)

En la vida real:

Big-O te dice *dónde mirar*
Profiling te dice *qué está pasando de verdad*

Mapa conceptual (texto, listo para pasar a diagrama)

Complejidad Algorítmica (Big-O)
→ Evaluar crecimiento del costo computacional
→
Aplicación profesional

- Diseño de sistemas escalables
- Prevención de bugs de rendimiento
- Optimización de código en producción
  →
  Herramientas / Técnicas
- Profiling
- Benchmarking
- Code reviews
- Pruebas con datos grandes

3 situaciones de tu futura carrera donde aplicarás esto

1. **Desarrollo de sistemas con muchos usuarios o datos**
   - APIs, bases de datos, procesamiento masivo
2. **Optimización de código que "funciona pero es lento"**
   - Especialmente en backend
3. **Revisión de código en equipos**
   - Detectar problemas antes de que lleguen a producción

Herramientas que deberías investigar (muy buena elección)

- **Chrome DevTools – Performance**
- **Node.js Profiler**
- **Benchmark.js**