# Tema III – Análisis de la complejidad de los algoritmos

En los capítulos anteriores hemos establecido los fundamentos conceptuales para diseñar y especificar Tipos Abstractos de Datos. Ahora, al comenzar a implementar algoritmos y estructuras de datos concretas, surge una pregunta fundamental: ¿cómo evaluamos y comparamos diferentes soluciones al mismo problema? La respuesta nos lleva al estudio de la complejidad algorítmica.

La complejidad de un algoritmo mide su eficiencia para resolver un problema determinado, considerando dos recursos críticos: el tiempo de ejecución y el espacio de memoria requerido. Estas mediciones no son absolutas (no se expresan en segundos o megabytes concretos), sino relativas al tamaño del problema y, lo que es más importante, independientes de factores externos como la velocidad del procesador, la plataforma de ejecución o el lenguaje de programación utilizado.

Consideremos un ejemplo práctico: si necesitamos buscar un elemento en una colección de datos, podríamos optar por una búsqueda lineal (revisando elemento por elemento) o una búsqueda binaria (dividiendo repetidamente el espacio de búsqueda). Intuitivamente, sabemos que la búsqueda binaria es "más rápida", pero ¿cómo cuantificamos esta diferencia? ¿Cómo determinamos cuándo un algoritmo es lo suficientemente eficiente para un problema de cierto tamaño? El análisis de complejidad nos proporciona las herramientas matemáticas para responder estas preguntas de manera rigurosa y objetiva.

Este capítulo nos introducirá en las técnicas para analizar y comparar algoritmos, centrándonos especialmente en la notación O grande (Big-O), el lenguaje universal para describir cómo escala el tiempo de ejecución de un algoritmo a medida que crece el tamaño de la entrada.

## 1. Conceptos Fundamentales

Como ya hemos dicho, la complejidad de un algoritmo mide su eficiencia para resolver un problema, lo cual puede ser analizado desde dos perspectivas diferentes: cuánto tiempo tarda en ejecutarse y cuánto espacio de memoria necesita para ello.

### Complejidad Temporal

La complejidad temporal de un algoritmo se refiere a cómo crece el tiempo de ejecución en función del tamaño de los datos de entrada, denotado típicamente como $n$. En lugar de medir tiempo real en segundos (que variaría según el hardware y otros factores), contamos el número de operaciones básicas que realiza el algoritmo. Estas operaciones básicas incluyen asignaciones, comparaciones, sumas, multiplicaciones, accesos a memoria, etc., asumiendo que cada una requiere un tiempo aproximadamente constante.

Supongamos que tenemos un array ordenado de 10 elementos y queremos determinar si contiene un valor específico. Una búsqueda lineal, en el peor caso (el elemento buscado está al final o no está presente), nos obliga a examinar los 10 elementos. Decimos pues que realiza $n$ operaciones de comparación, donde $n=10$. Por el contrario, una búsqueda binaria compara el elemento buscado con el elemento central, descarta la mitad de los elementos, y repite estos pasos hasta encontrar el elemento o darse cuenta de que no se halla en el array. En nuestro ejemplo, necesitamos como máximo 4 comparaciones pues matemáticamente, el número máximo de comparaciones que hace este algoritmo para $n=10$ es aproximadamente $\log_2(n)$, donde en este caso $\log_2(10) \approx 4$.

La diferencia es muy importante cuando tratamos con colecciones grandes: para $n=1.000.000$, la búsqueda lineal podría requerir un millón de comparaciones, mientras que la búsqueda binaria solo

requiere aproximadamente 20. Esta diferencia cuantitativa ilustra la importancia de analizar la complejidad de los algoritmos como paso previo a seleccionar el idóneo en cada caso.

### El caso más desfavorable (Worst-Case Analysis)

Cuando analizamos algoritmos, consideramos siempre el peor escenario posible. Esto puede parecer pesimista, pero proporciona una garantía: no importa qué entrada específica recibamos, el algoritmo nunca será peor que aquello que nuestro análisis predice. Esta aproximación es crucial para sistemas donde el rendimiento predecible es esencial (sistemas de tiempo real, aplicaciones críticas, etc.).

No obstante, existen otros tipos de análisis:

- Caso promedio: Considera la distribución típica de entradas.
- Mejor caso: Considera la entrada más favorable.

Sin embargo, el análisis del peor caso es el más común en la práctica porque proporciona una cota superior garantizada, evita tener que hacer suposiciones sobre la distribución de las entradas y desde luego, es más fácil de calcular que el caso promedio en la mayoría de algoritmos.

### Complejidad espacial

La complejidad espacial mide la cantidad de memoria adicional que necesita un algoritmo, más allá de la memoria ocupada por la entrada misma. Se expresa también en función de `n` (tamaño de la entrada) e incluye la memoria para variables locales, las estructuras de datos auxiliares y la pila de llamadas en algoritmos recursivos.

Aunque en este texto nos centraremos principalmente en la complejidad temporal, es importante recordar que tiempo y espacio son recursos intercambiables en el diseño de algoritmos. A veces podemos hacer un algoritmo más rápido usando más memoria (*caching*, tablas de búsqueda), o podemos ahorrar memoria a costa de un tiempo de ejecución mayor. Este equilibrio se conoce como *trade-off* (solución de compromiso) entre tiempo y espacio.

## 2. Notación O grande (Big-O)

La notación O grande (Big-O) es la notación utilizada comunmente para describir cómo crece el tiempo de ejecución de un algoritmo a medida que crece el tamaño de la entrada. Formalmente, decimos que una función `f(n)` es `O(g(n))` si existen constantes positivas `c` y $n_0$ tales que:

$$f(n) \leq c \cdot g(n) \text{ para todo } n \geq n_0$$

En términos más intuitivos, `O(g(n))` establece una cota superior asintótica para la función que cuenta las operaciones de nuestro algoritmo. Nos dice: "Más allá de cierto tamaño de problema, el tiempo de ejecución crecerá como mucho tan rápido como `g(n)`, multiplicado por alguna constante".

Cuando utilizamos esta notación ignoramos las constantes pero, ¿Por qué? En notación Big-O, simplificamos $O(3n^2+2n+100)$ a $O(n^2)$. Esto es así porque para valores grandes de `n`, el término $n^2$ domina completamente el comportamiento. Las constantes (3, 2, 100) y los términos de menor orden (2n) se vuelven insignificantes comparados con $n^2$ cuando `n` es suficientemente grande.

Las complejidades algorítmicas se organizan en una jerarquía clara, desde las más eficientes hasta las menos escalables:

- **O(1) - Complejidad Constante:** El tiempo de ejecución no depende del tamaño de la entrada. Es el caso ideal. Ejemplos: acceder a un elemento de un array por índice, realizar una operación aritmética simple.

- **O(log n) - Complejidad Logarítmica:** El tiempo crece logarítmicamente con `n`. Extremadamente eficiente para problemas grandes. Ejemplos: búsqueda binaria, operaciones en árboles binarios balanceados.
- **O(n) - Complejidad Lineal:** El tiempo es proporcional al tamaño de la entrada. Considerada "aceptable" para muchos problemas prácticos. Ejemplos: recorrer un array o lista, búsqueda lineal.
- **O(n log n) - Complejidad Cuasi-lineal:** Ligeramente peor que lineal pero mucho mejor que cuadrática. Aparece en algoritmos óptimos de ordenación como Merge Sort y Quick Sort.

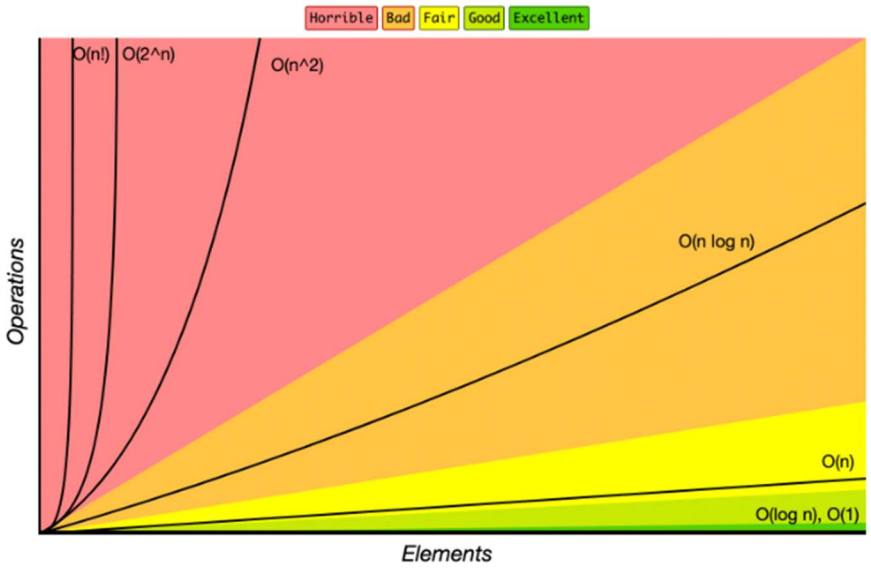


*Ilustración 6. Jerarquí de complejidades algorítmicas en notación O grande*

- **O($n^2$) - Complejidad Cuadrática:** El tiempo es proporcional al cuadrado del tamaño. Se vuelve problemático para entradas moderadamente grandes. Ejemplos: algoritmos de ordenación simples (Bubble Sort, Insertion Sort), procesamiento de matrices con bucles anidados.
- **O($2^n$) - Complejidad Exponencial:** El tiempo se duplica con cada incremento en el tamaño de entrada. Prácticamente intratable excepto para entradas muy pequeñas. Ejemplos: algunos algoritmos de fuerza bruta para problemas NP-completos.
- **O(n!) - Complejidad Factorial:** Aún peor que exponencial. Solo manejable para entradas muy limitads. Ejemplo: algoritmo de fuerza bruta para el problema del viajante.

Para apreciar las diferencias, consideremos cuántas operaciones requiere cada complejidad para diferentes tamaños de problema:

| n | O(1) | O(log n) | O(n) | O(n log n) | O($n^2$) | O($2^n$) | O(n!) |
|---|---|---|---|---|---|---|---|
| 10 | 1 | ~3 | 10 | ~30 | 100 | 1,024 | 3,628,800 |
| 100 | 1 | ~7 | 100 | ~700 | 10,000 | $1.27\times10^{30}$ | $9.33\times10^{157}$ |
| 1000 | 1 | ~10 | 1,000 | ~10,000 | 1,000,000 | $1.07\times10^{301}$ | $4.02\times10^{2567}$ |

Como se observa, las diferencias pasan a ser astronómicas para tamaños de n moderadamente grandes. Un algoritmo O(n!) para n=100 es completamente impracticable, mientras que un algoritmo O(n log n) para el mismo n es perfectamente manejable. Resumiendo, podemos decir que, en la práctica, los algoritmos con complejidad O($n^3$) o peor rara vez son útiles para problemas de tamaño significativo, mientras que los algoritmos O(n log n) o mejores son los que buscamos para problemas de tamaño considerable.

## 3. Reglas para calcular complejidades

Se consideran **operaciones simples** cuya complejidad es O(1) las asignaciones, las expresiones aritméticas, los accesos a elementos de arrays, la lectura/escritura de datos simples, y las llamadas a funciones que ejecutan en tiempo constante.

En el cálculo de complejidad hay que ignorar las constantes multiplicativas. Así, consideraremos que $O(5n)=O(n)$, $O(n/3)=O(n)$, $O(1000)=O(1)$, ya que las constantes no afectan el comportamiento asintótico.

Hay dos reglas más que debemos conocer: la de la suma y la del producto.

### Regla de la suma

Cuando nos encontremos ante una secuencia de instrucciones, habremos de seguir la **Regla de la Suma** que se enuncia del modo siguiente:

*"Si un algoritmo ejecuta el bloque A seguido del bloque B, la complejidad total es $O(max(f(n), g(n)))$, donde $f(n)$ y $g(n)$ son las complejidades de A y B respectivamente. Solo importa el término dominante (el de mayor crecimiento)."*

Si nos encontramos ante **estructuras selectivas** (if/else) tomaremos la máxima complejidad entre todas las ramas posibles.

### Regla del producto

Si estamos analizando bucles, aplicaremos la **Regla del Producto:**

*"Si un bucle de complejidad O(f(n)) contiene en su interior código de complejidad O(g(n)), la complejidad total es $O(f(n) \times g(n))$."*

Ejemplo:

```
for i := 1 to n do
    instrucción_simple  { O(1) }
```

La complejidad de este bucle será: $n \times O(1) = O(n)$

Otro ejemplo:

```
for i := 1 to 100 do  { 100 es constante, no depende de n }
    instrucción_simple  { O(1) }
```

La complejidad de este bucle es: $100 \times O(1) = O(1)$

Un último ejemplo, con bucles anidados, sería:

```
for i := 1 to n do          { O(n) }
    for j := 1 to n do      { O(n) }
        instrucción_simple  { O(1) }
```

Aquí la complejidad será: $O(n) \times O(n) \times O(1) = O(n^2)$ ya que los bucles son independientes. Si fuesen dependientes, como en el siguiente caso:

```
for i := 1 to n do              { O(n) }
    for j := 1 to i do          { O(i) que en promedio es O(n/2) }
        instrucción_simple      { O(1) }
```

entonces la complejidad total debería tener en cuenta que el bucle interno ejecuta $1 + 2 + 3 + \ldots + n = n(n+1)/2$ veces, por lo que su complejidad es: $O(n(n+1)/2) = O(n^2/2 + n/2) = O(n^2)$

## Análisis de algoritmos recursivos

Para algoritmos recursivos, la complejidad se determina mediante ecuaciones de recurrencia. Consideremos dos ejemplos. El primero analiza el cálculo del Factorial recursivo:

```
function Factorial(n: integer): integer;
begin
    if n = 0 then
        Factorial := 1            { caso base: O(1) }
    else
        Factorial := n * Factorial(n-1);  { llamada recursiva }
end;
```

La ecuación de recurrencia para este algoritmo sería: $T(n) = T(n-1) + O(1)$

Para determinar la complejidad, desarrollamos y tenemos:

$$
\begin{aligned}
T(n) &= T(n-1) + 1 \\
&= [T(n-2) + 1] + 1 = T(n-2) + 2 \\
&= [T(n-3) + 1] + 2 = T(n-3) + 3 \\
&\ldots \\
&= T(0) + n = 1 + n
\end{aligned}
$$

Es decir, la complejidad del algoritmo es: $O(n)$

El segundo ejemplo analizaría el algoritmo de Búsqueda Binaria Recursiva:

```
function BusquedaBinaria(arr: array; inicio, fin, objetivo: integer): boolean;
var
    medio: integer;
begin
    if inicio > fin then
        BusquedaBinaria := false          { O(1) }
    else begin
        medio := (inicio + fin) div 2;    { O(1) }
        if arr[medio] = objetivo then
            BusquedaBinaria := true       { O(1) }
        else if arr[medio] > objetivo then
            BusquedaBinaria := BusquedaBinaria(arr, inicio, medio-1, objetivo)
        else
            BusquedaBinaria := BusquedaBinaria(arr, medio+1, fin, objetivo);
    end;
end;
```

Aquí la ecuación de recurrencia es: $T(n) = T(n/2) + O(1)$ y puesto que, en cada llamada, el problema se reduce a la mitad su complejidad es $O(\log n)$:

$$
\begin{aligned}
T(n) &= T(n/2) + 1 \\
&= [T(n/4) + 1] + 1 = T(n/4) + 2 \\
&= [T(n/8) + 1] + 2 = T(n/8) + 3 \\
&\ldots \\
&= T(1) + \log_2(n) \approx 1 + \log_2(n)
\end{aligned}
$$

## 4. Implicaciones en el diseño de estructuras de datos

El análisis de complejidad no es solo un ejercicio teórico; tiene consecuencias directas en cómo diseñamos y seleccionamos las estructuras de datos que emplearemos en nuestros programas. Cada operación fundamental (inserción, búsqueda, eliminación) tiene una complejidad característica dependiendo de la estructura elegida:

| Operación | Array | Lista Enlazada | Árbol Binario Balanceado | Tabla Hash (promedio) |
|---|---|---|---|---|
| **Búsqueda** | O(n) | O(n) | O(log n) | O(1) |
| **Inserción** | O(n)* | O(1)** | O(log n) | O(1) |
| **Eliminación** | O(n)* | O(1)** | O(log n) | O(1) |

**En arrays: O(n) si hay que desplazar elementos; O(1) si es al final y hay espacio.*
***En listas enlazadas: O(1) si tenemos referencia al nodo; O(n) si debemos buscarlo primero.*

Esta tabla explica por qué no existe una estructura de datos perfecta para todos los casos: según las operaciones que más frecuentemente realicemos, elegiremos una estructura diferente. Por ejemplo, si necesitamos muchas búsquedas: tabla hash o árbol balanceado. Pero si necesitamos inserción/eliminación frecuentes en posiciones aleatorias utilizaremos una lista enlazada, mientras que si necesitamos acceso aleatorio por índice lo óptimo es utilizar un array.

### Optimización prematura vs. análisis informado

Un principio importante en ingeniería del software es evitar la optimización prematura. No debemos gastar tiempo optimizando algoritmos sin antes:

- Medir el rendimiento real del sistema
- Identificar los cuellos de botella (usando profilers)
- Confirmar que la optimización mejorará significativamente el rendimiento

Sin embargo, el análisis de complejidad no es optimización prematura, sino diseño informado, ya que nos permite seleccionar algoritmos apropiados desde el inicio, predecir cómo se comportará nuestro sistema con datos más grandes y evitar diseños que serán insostenibles al escalar.

Como ejemplo, pensemos en la selección del mejor algoritmo de ordenación para un cierto caso. Supongamos que debemos ordenar una colección de datos y para ello tenemos varias opciones:

- Método de la burbuja: $O(n^2)$ - Simple pero ineficiente para $n > 1000$
- Método de inserción: $O(n^2)$ - Eficiente para arreglos casi ordenados o pequeños ($n < 50$)
- Quick Sort: $O(n \log n)$ promedio, $O(n^2)$ peor caso - Muy eficiente en la práctica
- Merge Sort: $O(n \log n)$ garantizado - Estable pero requiere memoria adicional

Mediante una decisión informada podremos establecer criterios de selección como:

- Si $n \leq 50$: Insertion Sort (constantes pequeñas, código simple)
- Si $50 < n \leq 10{,}000$: Quick Sort (rápido en promedio)
- Si $n > 10{,}000$ y necesitamos garantías: Merge Sort
- Y si los datos ya están casi ordenados podemos optar por Inserción.

En resumen, esta decisión se basa en comprender no solo las complejidades asintóticas, sino también las constantes ocultas y las características específicas del problema. Esto demuestra que el análisis de complejidad es mucho más que una técnica matemática; es una herramienta fundamental de diseño que todo ingeniero de software debe dominar, ya que nos permite predecir el comportamiento de nuestros algoritmos ante entradas grandes, comparar objetivamente diferentes soluciones al mismo problema y seleccionar las estructuras de datos más apropiadas para cada contexto.

No obstante, debemo tener en cuenta que la complejidad algorítmica es solo una dimensión de la calidad del software. Un algoritmo O(n log n) con código ilegible y lleno de bugs es peor que un algoritmo $O(n^2)$ correcto y bien documentado. La complejidad debe equilibrarse con otros factores como claridad, robustez y mantenibilidad. El ingeniero de software competente deberá saber encontrar el equilibrio óptimo entre eficiencia y otras cualidades del software.