Algoritmos 1

Apéndice 3 - Complejidad Algorítmica

Primer Semestre, 2026

Contents

3.1 Introducción

Los microprocesadores que tenemos en nuestras computadoras personales y celulares se basan en una unidad central de procesamiento (CPU) que ejecuta un cierto número de *threads* (hilos) en paralelo, que ejecutan un código secuencial de instrucciones. A lo largo de la historia, estas CPUs se fueron llevando a límites de rendimiento cada vez mayores, donde gracias a la miniaturización de los componentes, la mayor cantidad de núcleos, la mayor velocidad del reloj, mejores formas de enfriamiento y la mejora en la eficiencia energética, se logró aumentar la cantidad de procesamiento que se podía hacer en un solo chip.

La idea del análisis de complejidad algorítmica es pensar en la cantidad de recursos que se necesitan para resolver un problema, y cómo esta cantidad de recursos se relaciona con el tamaño de la entrada del problema. Esto es un componente fundamental del diseño y la implementación de cualquier algoritmo, ya que el análisis de complejidad algorítmica nos permite entender qué algoritmo es más eficiente para resolver un problema dado, y cómo se comportará el algoritmo a medida que el tamaño de la entrada crezca.

3.2 Complejidad algorítmica

En el repaso de algoritmos (y materias anteriores esta) vieron sólo algoritmos secuenciales. Sin embargo antes de avanzar con la programación paralela, tenemos que estudiar *complejidad algorítmica* más en detalle.

La complejidad algorítmica es un parámetro con el cual podemos medir la eficiencia de un algoritmo en términos de la cantidad de recursos que va a utilizar para completar la tarea. Por un lado tenemos la complejidad en **tiempo**, que podemos pensarla como la cantidad de ciclos de CPU que necesita un algoritmo para obtener el resultado esperado, y por el otro lado, tenemos la complejidad en **espacio**, que podemos pensarlo como cantidad de memoria que necesita un algoritmo para obtener el resultado esperado.

Ambas complejidades se expresan en función del tamaño del *input* del algoritmo y nos permiten comparar diferentes algoritmos independientemente del lenguaje en que estén implementados. Esto significa que si tenemos dos lenguajes de programación diferentes, la complejidad tanto en **tiempo** como en **espacio** de un mismo algoritmo será la misma, aunque el tiempo de ejecución y la cantidad de memoria pueden ser diferentes ya que los lenguajes de programación tienen diferentes niveles de optimización.

3.2.1 Complejidad en Tiempo

Cuando diseñamos un algoritmo, es importante tener en cuenta la cantidad de recursos que se van a utilizar, tratando de estimar cuál será su complejidad en tiempo y en espacio de acuerdo a la solución que hayamos propuesto. Hay diferentes métricas de estimación, aunque las más comunes son estimar **el caso promedio** y, aún más importante, **el peor caso posible**. Por ejemplo, agregar un elemento al final de un *array* dinámico en cualquier lenguaje de programación de alto nivel, tiene una **complejidad promedio** constante, ya que cuando se crea un array dinámico, el lenguaje lo crea de un tamaño dado (aún si el usuario no lo especifica). Por el otro lado, si el *array* dinámico estuviera lleno, estaríamos en el *peor caso* posible, ya que el lenguaje, tendría que conseguir más memoria para añadir ese nuevo elemento y luego copiar todos los elementos a la nueva posición de memoria.

En esta materia vamos a ver sólo la notación de complejidad para el peor caso, que se representa con la notación **Big O**. En palabras simples, lo que representa la complejidad algorítmica en el peor caso es: *la cantidad máxima de tiempo de CPU o cantidad máxima de memoria que se requiere para resolver un problema en función del tamaño de la entrada.* Esta función **Big O** nos da una cota asintótica superior de la cantidad de recursos que se necesitan para resolver un problema en función del tamaño de la entrada.

En la figura 1, podemos observar diferentes funciones de complejidad algorítmica en función del tamaño de la entrada (`n`).

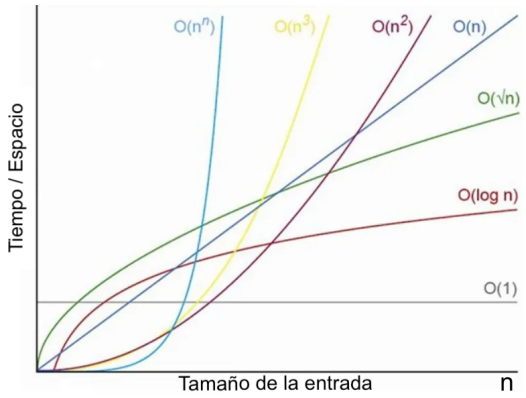

Figure 1: Big O - Comparación de complejidades

Diferentes implementaciones de un mismo algoritmo para resolver un problema pueden tener diferentes complejidades. Por ejemplo, existen varios algoritmos que resuelven el problema de ordenar un *array* (arreglo) de elementos de tamaño n. El más popular por su facilidad de implementación es el algoritmo de burbujeo que, si bien ordena los elementos, tiene una complejidad de $O(n^2)$, pero si utilizamos un algoritmo de ordenamiento más eficiente, nos encontramos con el *merge sort* que tiene una complejidad de $O(n \cdot \log\ n)$. Ambos algoritmos resuelven el mismo problema (ordernar un array), pero uno es mucho más eficiente que el otro. En la figura 2 podemos ver la comparación de ambas complejidades.

Descubrir un algoritmo más eficiente que otro no fácil.

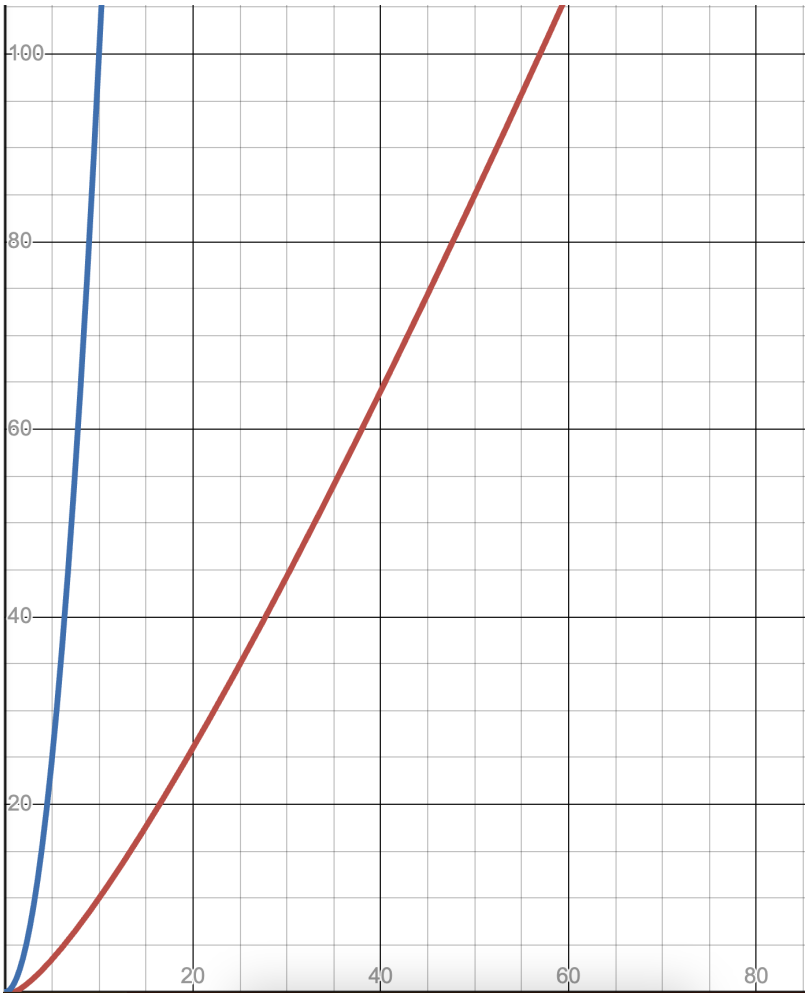

Figure 2: Big O - Comparación de complejidad algorítmica para Bubble Sort vs Quick Sort

3.2.2 Definición: Big O

En todos los gráficos que vimos la variable `n` es la cantidad de elementos que tenemos como *input* de nuestro algoritmo que se ejecutará con una cierta función de tiempo $f(n)$.

Definición: Diremos que el algoritmo se comporta con una complejidad asintótica de $O(g(n))$ si existe una constante C y un valor n_0 tal que el valor absoluto de $f(n)$ es menor o igual a C por el valor absoluto de $g(n)$ para todo n mayor a n_0. (Wilf [1]). Formalmente:

$$f(n) = O(g(n))\ (n \ \rightarrow \ \infty)\ si\ \exists\ C, n_0\ /\ |f(n)| \leq C\,|g(n)|\ (\forall\ n > n_0)$$

Las complejidades algorítmicas nos dan una idea de si un algoritmo puede ser realizable o no por una determinada unidad de procesamiento; por *realizable* queremos decir que la capacidad de resolver las instancias deseadas de un problema estén dentro de nuestros recursos disponibles. En la práctica, la factibilidad es muy dependiente del contexto y no es particularmente *portable* (portátil) entre diferentes problemas y situaciones. Un principio común se mantiene en casi todas las situaciones: **una tasa de crecimiento exponencial en el consumo de algún recurso limita la aplicación del uso de ese método a todas las instancias, excepto a las más pequeñas**. En otras palabras, si nuestros algoritmos tienden a tener complejidades

exponenciales (o más) en el tamaño de la entrada, no importa cuántos recursos tengamos, no podremos resolver instancias grandes del problema.

Por lo tanto, la factibilidad ha llegado a significar que la tasa de crecimiento del recurso está acotada por un una complejidad polinomial en el tamaño de la entrada. Esto nos da la noción común de que un problema tiene una solución secuencial factible sólo si tenemos un algoritmo de tiempo polinomial, es decir, sólo si cualquier instancia de tamaño n del problema se puede resolver en tiempo $n^{O(1)}$. Aunque ampliamente reconocido como muy simplista, la dicotomía entre algoritmos polinomiales y no polinomiales ha demostrado ser un discriminador poderoso entre aquellos cálculos que son factibles en la práctica y aquellos que no lo son (Greenlaw, Hoover, and Ruzzo [2]).

Lógicamente hay problemas para los cuales no se han encontrado algoritmos polinomiales, por ejemplo, la factorización entera de un número muy grande es un problema que no puede resolverse en tiempo polinomial con las CPUs y GPUs actuales. Por eso es que la seguridad de muchos sistemas de encriptación se basan en ello. Si bien excede lárgamente el contenido de esta materia, hay algoritmos cuánticos que pueden resolver este problema en tiempo $O((\log n)^{O(1)})$ ([3]).

3.2.3 Complejidad en Espacio

La complejidad en espacio es la cantidad de memoria requerida para resolver un algoritmo. Es, también, una función que depende de la entrada del algoritmo y de la cantidad de memoria adicional que se necesite para resolver el problema.

Como en la complejidad en tiempo, se utiliza la notación **Big O** ya que en este caso también se trata de una cota asintótica superior de la cantidad de memoria que se necesita para resolver un problema en función del tamaño de la entrada. Esto incluye tanto la memoria utilizada estáticamente como la memoria utilizada dinámicamente, sin embargo, no se tiene en cuenta la memoria utilizada como *input*. Por ejemplo, un algoritmo que cuenta la cantidad de números pares de un vector tiene una complejidad en tiempo de $O(n)$ y una complejidad en espacio de $O(1)$ ya que sólo necesita una variable para contar la cantidad de números pares.

Algorithm 1 Contar números pares

Define: countPares(a)
Input: $a = \{a_0, a_1, a_2, \ldots, a_n\}$
Initialization: $count = 0$
1: **Output:** $count$
2: **for** $i = 0$ to n **do**
3: **if** $a_i \mod 2 = 0$ **then**
4: $count = count + 1$

Vemos aquí que la complejidad en tiempo es $O(n)$ ya que sólo hay un ciclo que recorre todos los elementos del vector de tamaño n pero la complejidad en espacio es $O(1)$ ya que sólo utilizamos una variable `count` para guardar el contador con la cantidad de números pares. Esto significa que si el tamaño del vector a se duplica, la cantidad de memoria utilizada no cambia, pero la cantidad de ciclos de CPU necesarios para completar la tarea se duplica.

