Apuntes OIFem 2020

Big O

Qué es y por qué es útil

Cuanto mayor sea el input (entrada) de un programa, más tardará este en ejecutarse (por regla general). La notación Big O es una forma que tenemos de hacer este análisis de forma más rigurosa, es decir, es nuestra manera de describir cómo de rápido aumenta el tiempo que tarda un programa en ejecutarse en función del crecimiento de sus variables.

Acciones con complejidad constante

Hay una serie de acciones que podemos hacer como programadores de coste prácticamente gratuito. Estas son acciones cuya rapidez no varía con el tamaño de la entrada y, por tanto, se dice que tienen una complejidad constante, que escribimos como $O(1)$.

Algunos ejemplos son la suma de dos números o el añadir un número al final de un vector, operación cuya eficiencia no depende de la longitud del vector.

Además, es importante saber que $O(1) + O(1) = O(1)$, o sea, ejecutar dos acciones con complejidad constante en vez de una no altera que la complejidad de esa composición sea constante, ya que su rapidez sigue sin depender de ninguna variable.

Acciones con complejidad lineal

Por ejemplo, si vamos elemento a elemento de un vector con longitud n, guardando la suma de los elementos positivos dentro del vector (como sería el caso del problema Comprando Chuches, del primer entrenamiento), esto tiene un complejidad lineal.

Una complejidad lineal, $O(n)$, corresponde a aquellas funciones cuyo tiempo de ejecución es proporcional a una variable n, que suele ser la longitud de una lista. Estas son funciones que hacen un número fijo de acciones con complejidad constante a cada elemento de la lista.

De forma similar al caso anterior, $O(n) + O(n) = O(n)$, es decir, el ejecutar dos funciones lineales sobre la misma lista no cambia que la complejidad de esa composición sea lineal, ya que su rapidez sigue dependiendo de la longitud de la lista.

Acciones con complejidades exponenciales

Una función suele tener una complejidad de $O(n^k)$, donde k es una constante, cuando hay k bucles anidados con una complejidad de $O(n)$. Se puede obtener lo mismo con una solución equivalente escrita de forma recursiva.

Acciones con complejidad logarítmica

Estas son las acciones con una estructura arbórea. Es decir, algoritmos como la búsqueda binaria o la bisección, que van partiendo intervalos, además de las estructuras de datos como montículos o mapas ordenados cuyos datos se guardan en una estructura de árbol balanceado.

Combinando complejidades y simplificando

Si a una lista de longitud n le aplicas dos funciones `func1()` y `func2()`, ambas con una complejidad de $O(n)$, la complejidad de su composición será $O(n)$ también, ya que lo que frena que el programa sea más eficiente no deja de ser la variable n.

Por otra parte, si `func1()` tiene una complejidad de $O(n^2)$ y `func2()` de $O(n)$, su composición tendrá una complejidad de $O(n^2)$, ya que lo que frena su eficiencia es ese n^2, que siempre será mayor o igual a n.

Por lo tanto:

$O(kf(n)) = O(f(n))$, donde k es una constante y f una función sobre n que define la complejidad del programa.

$O(f(n) + k) = O(f(n))$, donde k es una constante y f una función sobre n que define la complejidad del programa.

$O(f(n) + g(n)) = O(f(n))$ si $g(n) \leq f(n)$ for all n.

¿Qué complejidades son mejores?

En orden de mejor a peor, donde k es una constante positiva y n es variable:

- $O(1)$
- $O(\log n)$
- $O(n^{1/k})$
- $O(n)$
- $O(n \log n)$
- $O(n^k)$
- $O(k^n)$