




Notación Asintótica y complejidad

FACULTAD DE CIENCIAS, UNAM

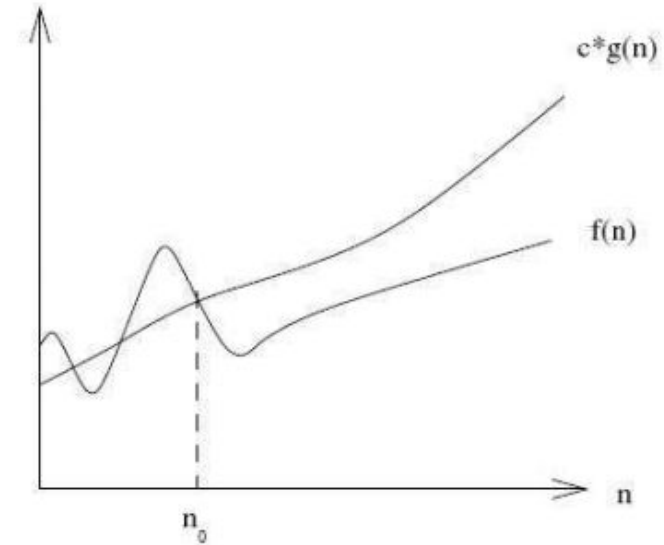
ESTRUCTURAS DE DATOS 2022-1

YESSICA JANETH PABLO MARTÍNEZ

- 
- La notación asintótica que usamos en este curso se denomina O-grande y nos dice lo siguiente: si tenemos una función $f(n)$ que indica el tiempo de ejecución de un algoritmo, decimos que el tiempo es del orden de una función $g(n)$, lo cual denotamos por $O(g(n))$, si para determinado tamaño de la entrada, el algoritmo tiene un tiempo de ejecución acotado superiormente por la función $g(n)$, multiplicada por una constante no negativa, que depende de la implementación particular del algoritmo. En otras palabras $g(n)$ en algún momento tiene un crecimiento mayor que la función $f(n)$.

- **Ejemplo:** La función $f(n) = 5n^4 + 3n^2 + 4$ es $O(n^4)$. Nos quedamos solamente con el término que influye de manera más significativa en el crecimiento de la función. Es más fácil decir que el tiempo de ejecución es del orden de una función polinomial de grado 4 que describen con toda precisión los detalles de $f(n)$.
- Formalmente, $f(n)$ es $O(g(n))$ si existe una constante c real y positiva y un entero no negativo n_0 tal que para todo valor $n \geq n_0$ se cumple que $0 \leq f(n) \leq c \cdot g(n)$.

- La siguiente figura ilustra de forma geométrica que la función $f(n)$ es del orden de $g(n)$, o más corto, $f(n)$ es $O(g(n))$, a partir de cierto tamaño de entrada n_0 , se puede garantizar que la función $f(n)$ queda acotada superiormente por otra función $g(n)$, multiplicada por un factor constante c , que consideramos una constante de implementación, derivada de la imprecisión de nuestro conteo de operaciones elementales.
- La notación asintótica permite determinar una cota superior asintótica al tiempo de ejecución de un algoritmo, lo cual corresponde a la complejidad del algoritmo en el peor caso. Para este curso, diremos simplemente que es la complejidad del algoritmo.



- La notación asintótica permite determinar una cota superior asintótica al tiempo de ejecución de un algoritmo, lo cual corresponde a la complejidad del algoritmo en el peor caso. Para este curso, diremos simplemente que es la complejidad del algoritmo.



Eficiencia Asintótica

- Cuando n es lo suficientemente grande:
- Un algoritmo $O(1)$, es más eficiente que un algoritmo $O(\log n)$
- Un algoritmo $O(\log n)$, es más eficiente que un algoritmo $O(n)$
- Un algoritmo $O(n)$, es más eficiente que un algoritmo $O(n \log n)$
- Un algoritmo $O(n \log n)$, es mas eficiente que un algoritmo $O(n^2)$
- Un algoritmo $O(n^2)$, es más eficiente que un algoritmo $O(n^3)$
- Un algoritmo $O(n^3)$, es más eficiente que un algoritmo $O(2n)$.
- **NOTA:** En ocasiones, un algoritmo más ineficiente puede resultar más adecuado para resolver un problema real ya que en a práctica hay que tener en cuenta otros aspectos además de la eficiencia.

Funciones de Complejidad

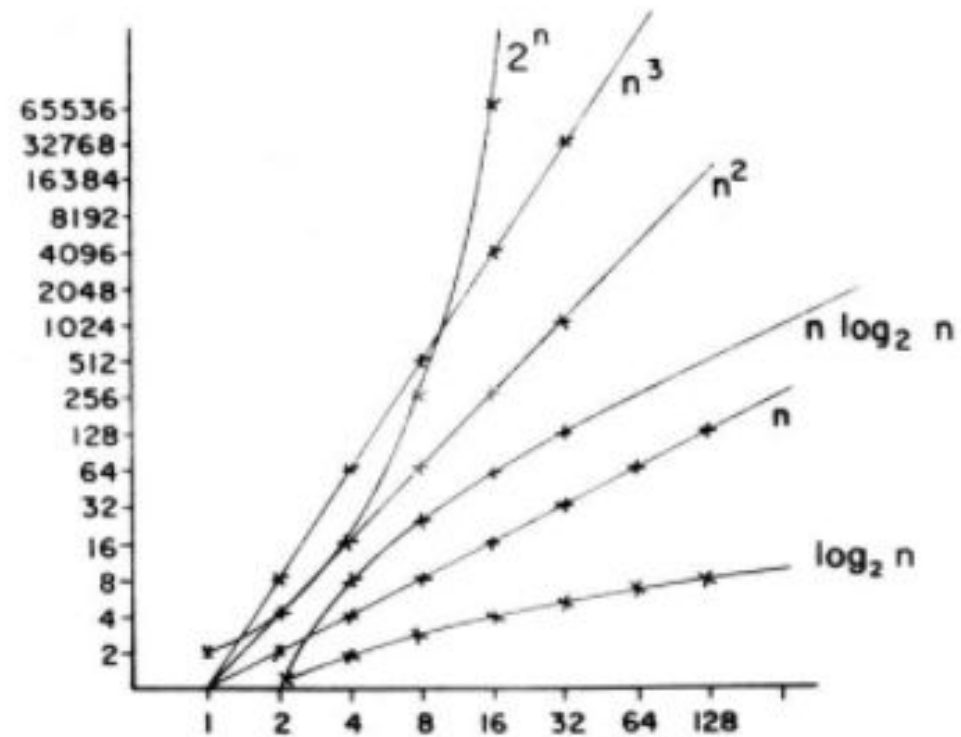
Son aquellas que representan tiempos de ejecución de algoritmos. Las complejidades (órdenes de magnitud) más comunes que estaremos viendo durante el curso son las siguientes:

- **Complejidad constante.** $O(1)$ son las funciones que tienen un número constante de operaciones, independientemente del tamaño de la entrada.
- **Complejidad logarítmica.** $O(\log n)$. Con esta complejidad aparecen algoritmos que de alguna forma descartan una parte constante de la entrada en cada fase (muy comúnmente la mitad).
- **Complejidad lineal.** $O(n)$. Con esta complejidad aparecen algoritmos que requieren examinar todos los elementos de la entrada de tamaño n .
- **Complejidad superlineal.** $O(n \log n)$. Con esta complejidad, aparecen algoritmos que involucran repetir un cómputo de complejidad logarítmica n veces, donde n es el tamaño de la entrada.

Funciones de Complejidad

- **Complejidad cuadrática.** $O(n^2)$. Con esta complejidad, por ejemplo, aparecen algoritmos que involucran repetir un cómputo de complejidad lineal n veces, donde n es el tamaño de la entrada.
- **Complejidad exponencial.** $O(b^n)$, con $b > 1$. El caso más común es $b = 2$. Algunos algoritmos de esta complejidad son, por ejemplo, aquellos que requieren examinar todos los subconjuntos que se pueden obtener de la entrada de n elementos.

- Tiempos más comunes de los algoritmos:
- $O(1) < O(\log n) < O(n) < O(n \log n) < O(n^2) < O(n^3) < O(2^n)$



Regla de la suma y regla del producto

Vamos a considerar dos reglas para obtener la complejidad en algoritmos, con base en su tiempo de ejecución, que ayudan a simplificar los cálculos de tiempos de ejecución:

Sean A_1, A_2 algoritmos con tiempo de ejecución $f_1(n)$ y $f_2(n)$, respectivamente.

- **Regla de la suma.** Para evaluar la complejidad del algoritmo resultante de ejecutar A_1 seguido de A_2 , sumamos los tiempos de ejecución $f_1(n) + f_2(n)$ y la complejidad es $O(f_1(n) + f_2(n))$. Esta regla se aplica para fragmentos independientes de código, uno se ejecuta después del otro.
- **Regla del producto.** Para evaluar la complejidad del algoritmo resultante de ejecutar A_2 dentro de A_1 , se multiplican los tiempos de ejecución $f_1(n) \cdot f_2(n)$ y la complejidad es $O(f_1(n) \cdot f_2(n))$. Esta regla se aplica para estructuras repetitivas (ciclos).