Complejidad de Algoritmos

Luis Garreta

Ingeniería de Sistemas y Computación
Pontificia Universidad Javeriana – Cali

29 de julio de 2017

Temas

- Diferencias entre el mejor de los casos, el caso esperado y el peor de los casos de los tiempos de ejecución de un algoritmo.
- Clases de complejidad: constante, logarítmica, lineal, cuadrática y exponencial.
- Comparación informal de la eficiencia de algoritmos (e.g. conteo de operaciones).

Sobre los Algoritmos

- Un algoritmo es un **método** para resolver un problema (computacional).
- Un algoritmo es la **idea** detrás del programa
- Un algorrimo es **independiente** del
  - lenguaje de programación,
  - de la máquina,
  - etc.

Next: *Propiedades Buscadas*

Propiedades buscadas en un Algoritmo

- **Correctitud:**
  - El algoritmo tiene que resover correctamente **todas las instancias** del problema
- **Eficiencia:**
  - El desempeño (**tiempo** y **memoria**) tiene que ser ”aceptable”.

Objetivo General del Curso

Diseñar algoritmos correctos y eficientes y probar que estos cumplen las especificaciones.

Next: *Porqué Preocuparnos*

Para qué analizar el Tiempo de Ejecución?

- **Predición:**
  - Cuanto tiempo necesita un algoritmo para resolver un problema?
  - Cómo este algoritmo escala de acuerdo al tamaño de la entrada?
  - Podemos brindar garantías sobre su tiempo de ejecución?
- **Comparación:**
  - Es un algoritmo **A** mejor que un algoritmo **B**?
  - Dependiendo de las circunstancias, cuál algoritmo utilizar el **A** o el **B**?

Next: *Sobre la Máquina*

Algoritmos y Velocidad de la Máquina (Computador)

Desempeño Algoritmico vs. Velocidad de la Máquina

Para instancias grandes del problema:

- Un "buen" algoritmo que se ejecuta sobre una computadora lenta **siempre será más rápido** que un "mal" algoritmo que se ejecuta sobre una computadora veloz.

Lo que realmente importa es la **taza de crecimiento** del tiempo de ejecución.

Máquina de Acceso Aleatorio (RAM)

- Necesitamos una **máquina para "ejecutar"** nuestros algoritmos
- Su **modelo** debe ser **generico** e **independiente** del lenguaje y de la máquina donde se implemente.
- Consideraremos a la Random Access Machine (RAM):
  - Cada operación simple (ej. $+$, -, $\leftarrow$, $/$, *, **if**) toma **1 paso.**
  - Ciclos y procedimientos, no se consideran operaciones simples.
  - Cada **acceso a memoria** (variables, arreglos) toman también **1 paso.**
- Vamos a medir el tiempo de ejecución **T(n)**:
  - **contando el número de pasos** como función del tamaño de la entrada **(n)**
- Por simplicidad, las operaciones básicas cuestan igual:
  - Aunque, sumar dos enteros tiene un costo diferente que dividir dos reales.

Tiempo de ejecución de un algoritmo: **T(n)**

- Normalmente, el factor más importante que afecta el tiempo de ejecución de un algoritmos es **el tamaño de la entrada.**
- Para un entrada de tamaño **n** se expresa el tiempo **T** para ejecutar un algoritmo como una función de n y se escribe cómo:

  T (n)

- **T(n)** siempre es positivo

Un Ejemplo de Conteo

- Programa Simple:

```
int count = 0;
for ( int i=0; i < n ; i ++)
  if ( v [ i ] == 0)
    count ++
```

Conteo de Operaciones

- Programa Simple:

```
int count = 0;
for ( int i=0; i < n ; i ++)
  if ( v [ i ] == 0)
    count ++
```

- Conteo de las operaciones:

Declaración de Variables	2
Asignaciones	2
Comparaciones "<"	n+1
Comparaciones "=="	n
Accesos a arreglos	n
Incrementos	entre n y $2n$

Peor y Mejor Casos

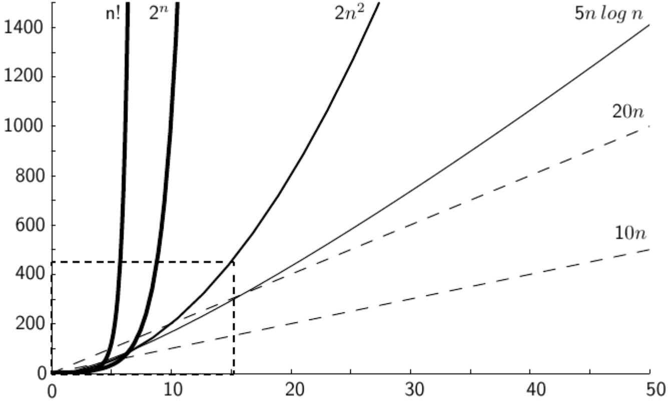
```
int count = 0;
for ( int i =0; i < n ; i ++)
  if ( v [ i ] == 0)
    count ++
```

Declaración de Variables	2
Asignaciones	2
Comparaciones "<"	n+1
Comparaciones "=="	n
Accesos a arreglos	n
Incrementos	entre n y $2n$

- **Análisis de Tiempos:**
  - Número total de pasos en el **peor caso:**
    - $T(n) = 2 + 2 + (n + 1) + n + n + 2n$
    - $T(n) = 5 + 5n$
  - Número total de pasos en el **mejor caso:**
    - $T(n) = 2 + 2 + (n + 1) + n + n + n = 5 + 4n$
    - $T(n) = 5 + 4n$

Tasas de Crecimiento de un Algoritmo

La tasas de crecimiento de un algoritmo es la tasa a la cual el costo del algoritmo crece a medida que el tamaño de su entrada crece.



Tipos de Análisis de Algoritmos

Análisis del Peor Caso - **Worst Case** - (El más común)

- $T(n) =$ Máxima cantidad de tiempo para cualquier entrada de tamaño n

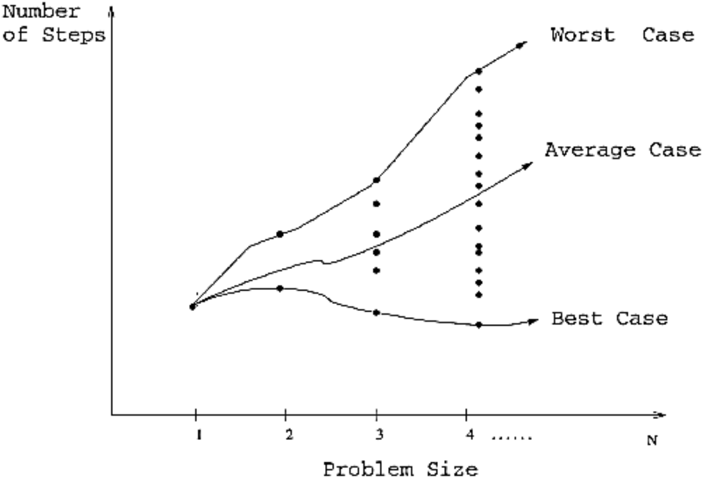
Análisis del Mejor Caso - **Best Case** - ("Engañosa")

- $T(n) =$ Cantidad de tiempo para entradas **"buenas"** de tamaño n
- Es como suponer que un algoritmo es rápido dependiendo de **solo "buenas"** entradas.

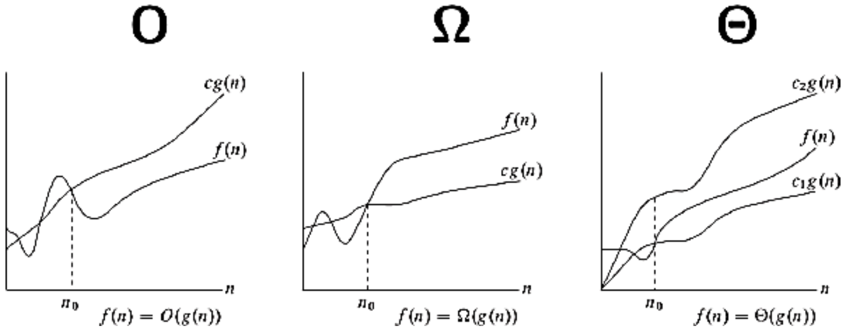
Análisis del Caso Promedio - **Average Case** - (Algunas veces)

- $T(n) =$ Promedio de tiempo sobre cualquier entrada de tamaño n
  - Implica conocer la distribución estadística de las entradas.

Tipos de Análisis de Algoritmos

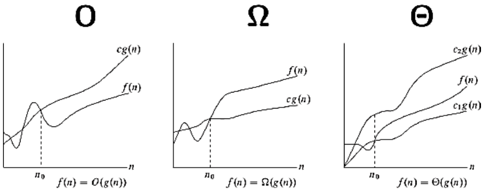


Notación Asintótica: **Descripción Gráfica**



Las definiciones implican un **n** para el cual la función está delimitada.
Los valores de $(n \leq n_0)$ no "importan".

Notación Asintótica: **Definiciones**



$f(n) = O(g(n))$: Significa que $c * g(n)$ es un **límite superior** de $f(n)$

$f(n) = \Omega(g(n))$: Significa que $c * g(n)$ es un **límite inferior de** $f(n)$

$f(n) = \Theta(g(n))$: Significa que $c_1 * g(n)$ es un **límite inferior** de $f(n)$ y $c_2 * g(n)$ es un **límite superior** de $f(n)$

Notación Asintótica: **Formalización**

$f(n) = O(g(n))$

Si existen constantes positivas n_0 y c tal que $f(n) \leq c * g(n)$ para toda $n \geq n_0$

$f(n) = \Omega(g(n))$

Si existen constantes positivas n_0 y c tal que $f(n) \geq c * g(n)$ para toda $n \geq n_0$

$f(n) = \Theta(g(n))$

Si existen constantes positivas n_0, c_1, c_2 tal que $c_1 * g(n) \geq f(n) \geq c_2 * g(n)$ para toda $n \geq n_0$

Notación Asintótica: **Analogías**

Comparación entre las funciones **f** y **g** y dos numeros **a** y **b**

$f(n) = O(g(n))$ es como $a \leq b$

$f(n) = \Omega(g(n))$ es como $a \geq b$

$f(n) = \Theta(g(n))$ es como $a = b$

Notación Asintótica: **Reglas Prácticas**

- Multiplicar por una constante no afecta:
  $\Theta(c \times f(n)) = \Theta(f(n))$
  $99 \times n^2 = \Theta(n^2)$
- En un polinomio de la forma $a_x n^x + a_{x-1} n^{x-1} + ... + a_2 n^2 + a_1 n + a_0$ debemos enfocarnos en el términos con **mayor exponente**
  $3\mathbf{n^3} - 5n^2 + 100 = \Theta(n^3)$
  $6\mathbf{n^4} - 20^2 = \Theta(n^4)$
  $0.8\mathbf{n} + 224 = \Theta(n)$
- En una suma, debemos enfocarnos en el **término dominante:**
  $\mathbf{2^n} + 6n^3 = \Theta(2^n)$
  $\mathbf{n}! - 3n^2 = \Theta(n!)$
  $n \log n + 3\mathbf{n^2} = \Theta(n^2)$

Notación Asintótica: **Relaciones de Dominancia**

Cuando una función es **mejor** que otra?

- Si se quiere minimizar tiempo, las funciones **"pequeñas"** son mejores.
- Una función domina a otra si a medida que crece n esta sigue aumentando
- Matemáticamente:
  - $f(n) \gg g(n)$ if $\lim_{n \to \infty} g(n)/f(n) = 0$

Algunas relaciones de dominancia:

$n! \gg 2^n \gg n^3 \gg n^2 \gg n \log n \gg n \gg \log n \gg 1$

Notación Asintótica: **Una vista práctica**

Si una operación toma 10^{-9} segundos, entonces:

	$\log n$	n	$n \log n$	n^2	n^3	2^n	$n!$
10	$< 0.01s$	$< 0.01s$	$< 0.01s$	$< 0.01s$	$< 0.01s$	$< 0.01s$	$< 0.01s$
20	$< 0.01s$	$< 0.01s$	$< 0.01s$	$< 0.01s$	$< 0.01s$	$< 0.01s$	77 years
30	$< 0.01s$	$< 0.01s$	$< 0.01s$	$< 0.01s$	$< 0.01s$	$1.07s$	
40	$< 0.01s$	$< 0.01s$	$< 0.01s$	$< 0.01s$	$< 0.01s$	18.3 min	
50	$< 0.01s$	$< 0.01s$	$< 0.01s$	$< 0.01s$	$< 0.01s$	13 days	
100	$< 0.01s$	$< 0.01s$	$< 0.01s$	$< 0.01s$	$< 0.01s$	10^{13} years	
10^3	$< 0.01s$	$< 0.01s$	$< 0.01s$	$< 0.01s$	$1s$		
10^4	$< 0.01s$	$< 0.01s$	$< 0.01s$	$0.1s$	16.7 min		
10^5	$< 0.01s$	$< 0.01s$	$< 0.01s$	$10s$	11 days		
10^6	$< 0.01s$	$< 0.01s$	$0.02s$	16.7 min	31 years		
10^7	$< 0.01s$	$0.01s$	$0.23s$	1.16 days			
10^8	$< 0.01s$	$0.1s$	$2.66s$	115 days			
10^9	$< 0.01s$	$1s$	$29.9s$	31 years			

Notación Asintótica: **Funciones Comunes**

Function	Name	Examples
1	constante	summing two numbers
$\log n$	logarithmic	binary search, inserting in a heap
n	linear	1 cycle to find maximum value
$n \log n$	linearithmic	sorting (ex: mergesort, heapsort)
n^2	quadratic	2 cycles (ex: verifying, bubblesort)
n^3	cubic	3 cycles (ex: Floyd-Warshall)
2^n	exponential	exhaustive search (ex: subsets)
$n!$	factorial	all permutations

Ejemplos de Algoritmos y Conteo

Desarrollar las funciones (en lenguaje C) y analizar los tiempos para los siguientes problemas.

1. Una función que busque el valor mas grande dentro de un arreglo. Ingresa el arreglo y su tamaño, retorna el valor más grande.
2. Una función que busca un elemento dentro de un arreglo ordenado. Ingresa el arreglo y el valor a busca, retorna 0: Falso, 1: Verdad.
3. Una función que verifica si dos arreglos contienen los mismos elementos. Ingresan los dos arreglos, retorna 0: Falso, 1: Verdad.
4. Una función que ordene un arreglo de forma ascendente. Ingresa el arreglo, retorna 0: Falso, 1: Verdad.