Complejidad de Algoritmos

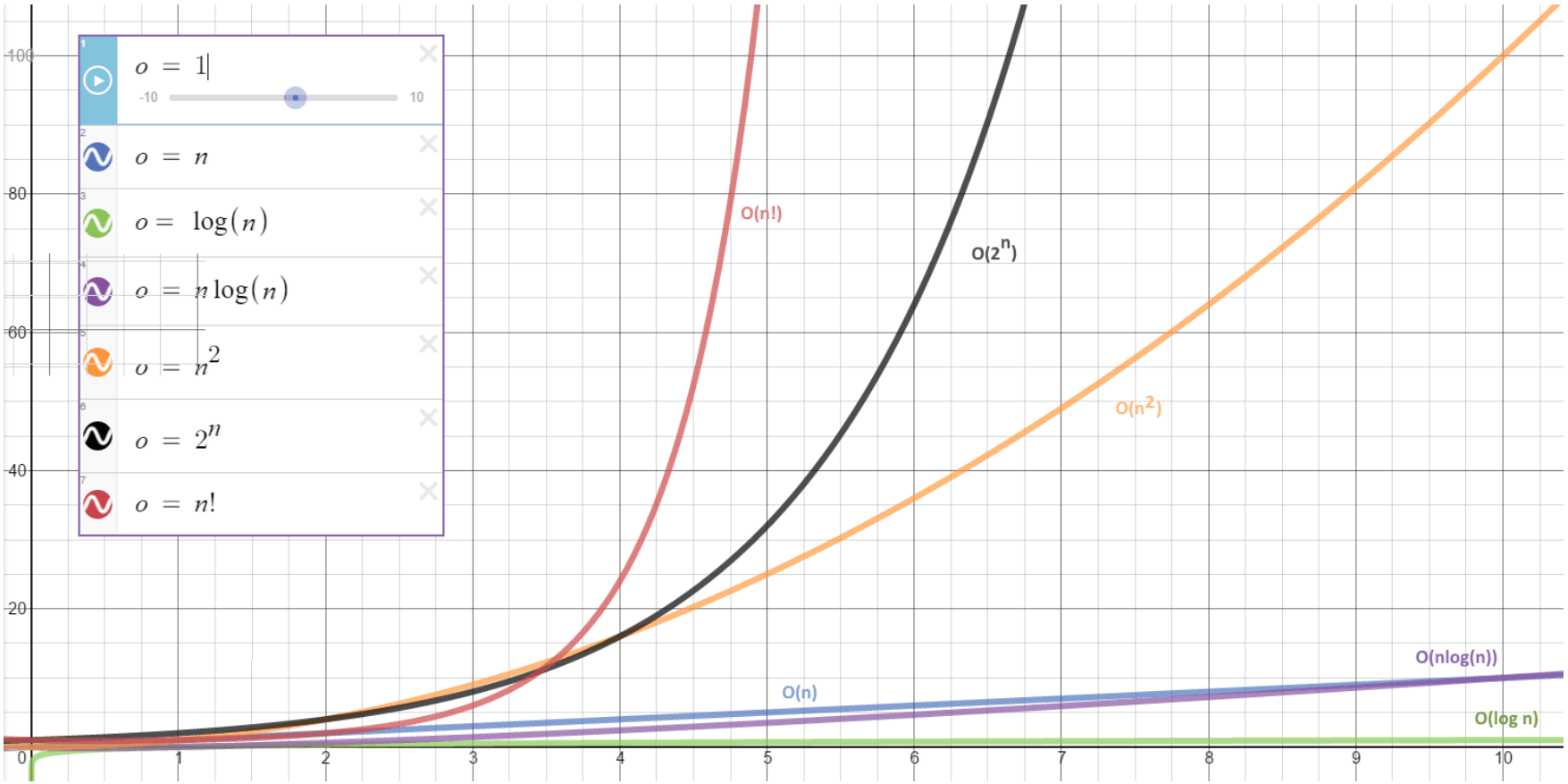
Segunda Parte

Luis Garreta

Ingeniería de Sistemas y Computación
Pontificia Universidad Javeriana – Cali

31 de julio de 2017

Clases de Complejidad Algorítmica



Clases de Complejidad Algoritmica

- **O(1)**: constante. La operación no depende del tamaño de los datos. Es el caso ideal, pero a la vez probablemente el menos frecuente.
- **O(n)**: lineal. El tiempo de ejecución es directamente proporcional al tamaño de los datos. Crece en una línea recta.
- **O(*logn*)**: logarítmica. por regla general se asocia con algoritmos que "trocean" el problema para abordarlo, como por ejemplo una búsqueda binaria.
- **O(*nlogn*)**: en este caso se trata de funciones similares a las anteriores, pero que rompen el problema en varios trozos por cada elemento, volviendo a recomponer información tras la ejecución de cada "trozo". Por ejemplo, el algoritmo de búsqueda Quicksort.
- **O(n^2)**: cuadrática. Es típico de algoritmos que necesitan realizar una iteración por todos los elementos en cada uno de los elementos a procesar. Por ejemplo el algoritmo de ordenación de burbuja. Si tuviese que hacer la iteración más de una vez serían de complejidad O(n^3), O(n^4), etc... pero se trata de casos muy raros y poco optimizados.
- **O(2^n)**: exponencial. Se trata de funciones que duplican su complejidad con cada elemento añadido al procesamiento. Son algoritmos muy raros pues en condiciones normales no debería ser necesario hacer algo así. Un ejemplo sería, por ejemplo, el cálculo recursivo de la serie de Fibonacci, que es muy poco eficiente (se calcula llamándose a sí misma la función con los dos números anteriores: F(n)=F(n-1)+F(n-2)).
- **O(n!)**; explosión combinatoria. Un algoritmo que siga esta complejidad es un algoritmo totalmente fallido. Una explosión combinatoria se dispara de tal manera que cuando el conjunto crece un poco, lo normal es que se considere computacionalmente inviable. Solo se suele dar en algoritmos que tratan de resolver algo por la mera fuerza bruta.

Θ Notation

- Las definiciones para **O y** Ω nos permiten el límite superior e inferior para un algoritmos.
- Cuando el límite superior e inferior son los mismos dentro de un factor constante, indicamos esto usando la notación Θ.
- Se dice que un lagorimo es Θ**(h(n))** si está en **O(h(n))** y está en Ω**(h(n)).**

Ejemplo: Algoritmo de Búsqueda Secuencial

```
// Retorna la posición del valor más grande en "A" de tamano "n"
int posicionValorMayor(int A[], int n) {
  int posicion = 0; // Holds largest element position
  for (int i=1; i<n; i++)  // For each array element
    if (A[posicion] < A[i]) // if A[i] is larger
      posicion = i;  //  remember its position
  return posicion;  // Return largest position
}
```

- Ya que el algoritmo de búsqueda secuencial está tanto en O(n) y en Ω(n), decimos que es Θ**(n).**
- Generalmente se habla de que un algoritmo está **"en el orden de" O (o grande)** de alguna función, pero es generalmente mejor usar la notación Θ que la de **O** siempre y cuando se tenga suficiente conocimiento del algoritmo para asegurar que los límites superior e inferior son los mismos.
- Si no, se tiene ese conocimiento entonces se puede usar la notación **O** que es menos restrictiva que la Θ**.**

Reglas de Simplificación

- Una vez se determina la ecuación del tiempo de ejecución para un algoritmo, es simple derivar las expresiones **O, Ω, y Θ** para las ecuaciones.
- No se necesita recurrir a las definiciones formales del análisis asintótico, más bien seguir algunas de las siguientes reglas::

1. If f (n) está en O(g(n)) y g(n) está en O(h(n)), entonces f (n) está en O(h(n)).
2. If f (n) está en O(kg(n)) para cualquier constante k $>$ 0, entonces f (n) está en O(g(n)).
3. If f1 (n) está en O(g1 (n)) y f2(n) está en O(g2(n)), entonces f1(n) + f2 (n) está en O(max(g1 (n), g2(n))).
4. If f1 (n) está en O(g1 (n)) y f2 (n) está en O(g2 (n)), entonces f1(n)*f2 (n) está en O(g1 (n)*g2 (n)).

Clasificación de las Funciones

- Dado funciones f (n) y g(n) cuyas tasas de crecimiento son expresadas como ecuaciones algebraicas, queremos determinar si una crece más rápido que otra.
- La major manera es tomar el límite de las dos funciones a medida que **n** crece hacia el infinito:

$$\lim_{n\to\infty} \frac{f(n)}{g(n)}.$$

- Si el límite va hasta ∞, entonces f (n) está en Ω(g(n)) porque f (n) crece más rápido.
- Si el límite va hasta cero, entonces f (n) está en O(g(n)) porque g(n) crece más rápido.
- Si el límite va hasta alguna constante diferente de cero, entonces f (n) = Θ(g(n)) porque ambas crecen a la misma tasa.

Ejemplo: Determinación de la tasa de crecimiento

Si f (n) = $2n \log n$ y g(n) = n^2 , en donde está f (n), en **O**(g(n)), Ω(g(n)), or Θ(g(n))?
Because

$$\frac{n^2}{2n \log n} = \frac{n}{2 \log n},$$

vemos fácilmente que

$$\lim_{n \to \infty} \frac{n^2}{2n \log n} = \infty$$

porque **n** crece más rápido than **2 log n**. Entonces, n^2 está en Ω(2n log n).

Cálculos simples del tiempo de ejecución

Una asignación simple.

```
a = b;
```

Es $\Theta(1)$.

Un ciclo simple.

```
sum = 0;
for (i=1; i<=n; i++)
  sum += n;
```

$T(n) = c + n$

Es $\Theta(n)$

Ejemplo 4: Varios ciclos, algunos de ellos anidados

```
sum = 0;
for (i=1; i<=n; i++)  // First for loop
  for (j=1; j<=i; j++)  //  is a double loop
    sum++;

for (k=0; k<n; k++)  // Second for loop
  A[k] = k;
```

- El primer doble ciclo es especial ya que el ciclo anidado no llega hasta n, llega hasta i.
- Y el valor de i está cambiando a cada paso del ciclo externo.

Ejemplo 4: Varios ciclos, algunos de ellos anidados

```
c1    :    sum = 0;
n     :    for (i=1; i<=n; i++)  // First for loop
             for (j=1; j<=i; j++)  //  is a double loop
c3*i :            sum++;

n     :    for (k=0; k<n; k++)  // Second for loop
             A[k] = k;
```

- Observe que i inicia en 1, despúes en 2, ..., hasta $i = n$
- Estos pasos de i se deben ir sumando 1+2+...+n
- De las formulas para resolver sumatorias:

$$\sum_{i=1}^{n} i = \frac{n(n+1)}{2},$$

el cual $\Theta(n2)$

Formulas para Sumatorias

$$\sum_{i=1}^{n} i = \frac{n(n+1)}{2}. \tag{2.1}$$

$$\sum_{i=1}^{n} i^2 = \frac{2n^3+3n^2+n}{6} = \frac{n(2n+1)(n+1)}{6}. \tag{2.2}$$

$$\sum_{i=1}^{\log n} n = n \log n. \tag{2.3}$$

$$\sum_{i=0}^{\infty} a^i = \frac{1}{1-a} \text{ for } 0 < a < 1. \tag{2.4}$$

$$\sum_{i=0}^{n} a^i = \frac{a^{n+1}-1}{a-1} \text{ for } a \neq 1. \tag{2.5}$$

Ejemplo 5: Dos ciclos anidados

```
sum1 = 0;
for (i=1; i<=n; i++)  // First double loop
  for (j=1; j<=n; j++)  //  do n times
    sum1++;

sum2 = 0;
for (i=1; i<=n; i++)  // Second double loop
  for (j=1; j<=i; j++)  //  do i times
    sum2++;
```

Ejemplo 5: Dos ciclos anidados

```
sum1 = 0;
for (i=1; i<=n; i++)  // First double loop
  for (j=1; j<=n; j++)  //  do n times
    sum1++;

sum2 = 0;
for (i=1; i<=n; i++)  // Second double loop
  for (j=1; j<=i; j++)  //  do i times
    sum2++;
```

- El primer ciclo es $\Theta(n2)$
- El segundo ciclo cuesta aprox $(1/2n^2)$,
- Al final, el costo total es $\Theta(n^2)$

Not all doubly nested for loops are $\Theta(n2)$

```
sum1 = 0;
for (k=1; k<=n; k*=2)     // Do log n times
  for (j=1; j<=n; j++) // Do n times
    sum1++;

sum2 = 0;
for (k=1; k<=n; k*=2)     // Do log n times
  for (j=1; j<=k; j++) // Do k times
    sum2++;
```

- Asumimos que n es potencia de 2:

Not all doubly nested for loops are Θ(n2)

```
sum1 = 0;
for (k=1; k<=n; k*=2)     // Do log n times
  for (j=1; j<=n; j++) // Do n times
    sum1++;

sum2 = 0;
for (k=1; k<=n; k*=2)     // Do log n times
  for (j=1; j<=k; j++) // Do k times
    sum2++;
```

- Asumimos que n es potencia de 2:
- En el primer ciclo:
  - El primer fragmento de código ejecuta su ciclo externo en $logn + 1$
  - El ciclo anidado siempre se ejecuta n veces
  - Entonces se tiene la siguiente sumatoria:

    $$\sum_{i=0}^{\log n} n.$$

  - Que da como resultado $\Theta(nlogn)$
- En el segundo ciclo?