Eficiencia y complejidad

Complejidad Temporal, Estructuras de Datos y Algorítmos

Objetivos

- Analizar la COMPLEJIDAD de distintos algorítmos
- Comparar la conveniencia del algorítmo a utilizar en base a su EFICIENCIA.
- Introducir conceptos relacionados con el Cálculo de la EFICIENCIA:
  - Tiempo de ejecución T(N)
  - Orden de magnitud para un T(N) dado.

Tipos de Algoritmos

- Algoritmos iterativos: Son algoritmos que realizan una tarea por medio de la ejecución de una serie de instrucciones

- Algoritmos recursivos: Son algoritmos que para realizar una tarea dependen de si mismos, puesto que se llaman a si mismo.

Ejemplos de tipos de Algoritmos

- Un algorítmo recursivo que resuelve la "Secuencia de Fibonacci"

$$F_n = F_{n-1} + F_{n-2}, \text{ con } F_1 = F_2 = 1$$

```java
public int fibonacci_recursivo(int n){
    if(n<=2){
        return 1;
    }else{
        return (fibonacci_recursivo(n-1)+fibonacci_recursivo(n-2));
    }
}
```

Ejemplos de tipos de Algoritmos

- Un algorítmo iterativo que resuelve la "Secuencia de Fibonacci" $F_n = F_{n-1} + F_{n-2}$, con $F_1 = F_2 = 1$

```
public int fibonacci_iterativo(int n){
    int f1=1;
    int f2=1;
    int aux=1;
    int indice=3;
    while(indice<=n){
        aux=f1+f2;
        f1=f2;
        f2=aux;
        indice++;
    }
    return f2;
}
```

Análisis de algorítmos

- Nos permite comparar algoritmos en forma independiente de una plataforma en particular
- Permite determinar una estimación del tiempo que tendrá la ejecución del algorítmo en función al tamaño de su ENTRADA.
- Llamaremos T(n) al tiempo de ejecución de un algoritmo cuando su entrada es de tamaño n.
  - Por ejemplo, si un algoritmo ordena elementos, el T(n) del mismo brindará una estimación de lo que tardará cuando haya N elementos a ordenar.

Análisis de algorítmos

- Pasos a seguir:
  - Caracterizar los datos de entrada del algoritmo
  - Identificar las operaciones abstractas, sobre las que se basa el algoritmo
  - Realizar un análisis matemático, para encontrar los valores de las cantidades del punto anterior

Peor caso

- Un algorítmo PUEDE realizar mas procesamiento en determinadas situaciones.
- Para el Cálculo del T(n), siempre consideraremos el PEOR CASO del algorítmo
- Por ejemplo, en el peor de los casos, ¿cuántas iteraciones hará el siguiente código?

```
public void salioElCinco(){

    char valor = (char) Console.Read();
    while(valor!= '5'){
        valor = (char) Console.Read();
    }
    Console.Write("Salio el 5");

}
```

¿Cómo se calcula el T(n)?

- Operaciones Constantes
  - Las operaciones o bloques de operaciones que no dependen de la ENTRADA, tienen un tiempo constante.
    - En el siguiente ejemplo, se tiene un BLOQUE de OPERACIONES CONSTANTES.
    - El tiempo asociado a TODO el BLOQUE, también será CONSTANTE

```
char valor = '5';
int aux = 0;
Console.Write(aux);
Console.ReadKey();
valor = (char) Console.Read();
Console.Write(valor);
```

T(n) = cte1

¿Cómo se calcula el T(n)?

- Estructuras IF THEN ELSE
  - Dependiendo la condición, el algoritmo tomará el camino del THEN o del ELSE.
  - Cuando se calcula el T(n) se toma el tiempo asociado a la rama del IF que provoca el PEOR CASO.

```
char valor = '5';
int aux = 0;
Console.Write(aux);
Console.ReadKey();
valor = (char) Console.Read();
Console.Write(valor);
if (valor=='1') {
```

cte1

```
    aux = aux*3/2;
    Console.Write(aux);
    Console.ReadKey();
```

cte2

```
} else {
    valor='0';
```

cte3

```
}
```

¿Cómo se calcula el T(n)?

- Estructuras IF THEN ELSE
  - Si cte2 > cte3 entonces, el T(n) = cte1 + cte2
  -
  - Simplificando podemos decir que T(n) = cte5
    - Donde cte5 = cte1 + cte2

```
char valor = '5';
int aux = 0;
Console.Write(aux);
Console.ReadKey();
valor = (char) Console.Read();
Console.Write(valor);
if (valor=='1') {
    aux = aux*3/2;
    Console.Write(aux);
    Console.ReadKey();
} else {
    valor='0';
}
```

T(n)= cte5

¿Cómo se calcula el T(n)?

- Iteradores FOR
  - Cuando se utiliza un FOR, se debe calcular el tiempo de ejecución asociado al cuerpo del FOR y multiplicar ese tiempo por la cantidad de iteraciones que tenga el FOR.
  - Calcular el T(n) de:

```
public void contar(int n){
    for (int i = 1; i <= n; i++) {
        Console.Write(i);
    }
}
```

$$T(n) = \sum_{i=1}^{n} cte2 = n\ cte2$$

¿Cómo se calcula el T(n)?

- Iteradores FOR (continuación)
  - Calcular el T(n) de:

```
public void contar2(int n){
    for (int i = 0; i <= n; i++) {
        for (int j = 0; j <= n; j++) {
            Console.Write(i+":"+j);
        }
    }
}
```

$$T(n) = cte1 + \sum_{i=0}^{n} \left(\sum_{i=0}^{n} cte3 \right) = cte1 + \sum_{i=0}^{n} ((n+1)\,cte3) = cte1 + \sum_{i=0}^{n} (n\,cte3 + cte3) = cte1 + (n+1)(n\,cte3 + cte3)$$

$$T(n) = cte1 + n^2\,cte3 + n\,cte3 + n\,cte3 + cte3 \qquad = \qquad cte5 + n^2\,cte3 + 2\,n\,cte3$$

$$cte1 + cte3 = cte5$$

¿Cómo se calcula el T(n)?

- Iteradores FOR (continuación)
  - En ocasiones los indices del FOR están anidados

```
for (int i = 1; i <= n; i++) {
    for (int j = 1; j <= i; j++) {
        Console.Write(i+":"+j);
    }
}
```

¿Cómo se calcula el T(n) en estos casos?

¿Cómo se calcula el T(n)?

```
for (int i = 1; i <= n; i++) {
      for (int j = 1; j <= i; j++) {
            Console.Write(i+":"+j);
      }
}
```

$$T(n) = \sum_{i=1}^{n} \left(\sum_{j=1}^{i} c1 \right) = \sum_{i=1}^{n} (i * c1) = c1 * \sum_{i=1}^{n} i = c1 \frac{n(n+1)}{2} = c1 \frac{n^2 + n}{2} = \frac{c1}{2} n^2 + \frac{c1}{2} n$$

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

¿Cómo se calcula el T(n)?

- Iteradores WHILE
  - El Cálculo del tiempo de ejecución de un WHILE es similar al del FOR. Lo mas dificil es estimar la cantidad de iteraciones puesto que a diferencia del FOR, en el WHILE esto puede ser mas complicado
  - Calcular el T(n) de la siguiente funcion:

```
public void tiempoWhile(int n){
    int i=1;
    while (i<n) {
        Console.Write(i);
        i=i*2;
    }
}
```

¿Cómo se calcula el T(n)?

Para resolver el T(n) de la función necesitamos estimar la cantidad de iteraciones que realizará el WHILE

```
public void tiempoWhile(int n){
    int i=1;
    while (i<n) {
        Console.Write(i);
        i=i*2;
    }
}
```

- Iteración 1 - Valor de i = 1
- Iteración 2 - Valor de i = 2
- Iteración 3 - Valor de i = 4
- Iteración 4 - Valor de i = 8
- Iteración 5 - Valor de i = 16
- Iteración k - Valor de i = 2^{k-1}
- Si el WHILE itera “k“ veces, es porque el valor de i alcanzó a n. Por lo tanto $2^{k-1} = n$. Despejando “k” tenemos:

$$k - 1 = \log_2(n) \quad / \quad k = \log_2(n) + 1$$

¿Cómo se calcula el T(n)?

- Dijimos que si el WHILE iteraba “k“ veces, era porque el valor de i habia alcanzado el valor n
- Con esta suposición tenemos que $2^{k-1} = n$. Despejando “k” tenemos:
  - $k - 1 = \log_2(n)$ / $k = \log_2(n) + 1$
  - Entonces el tiempo será:

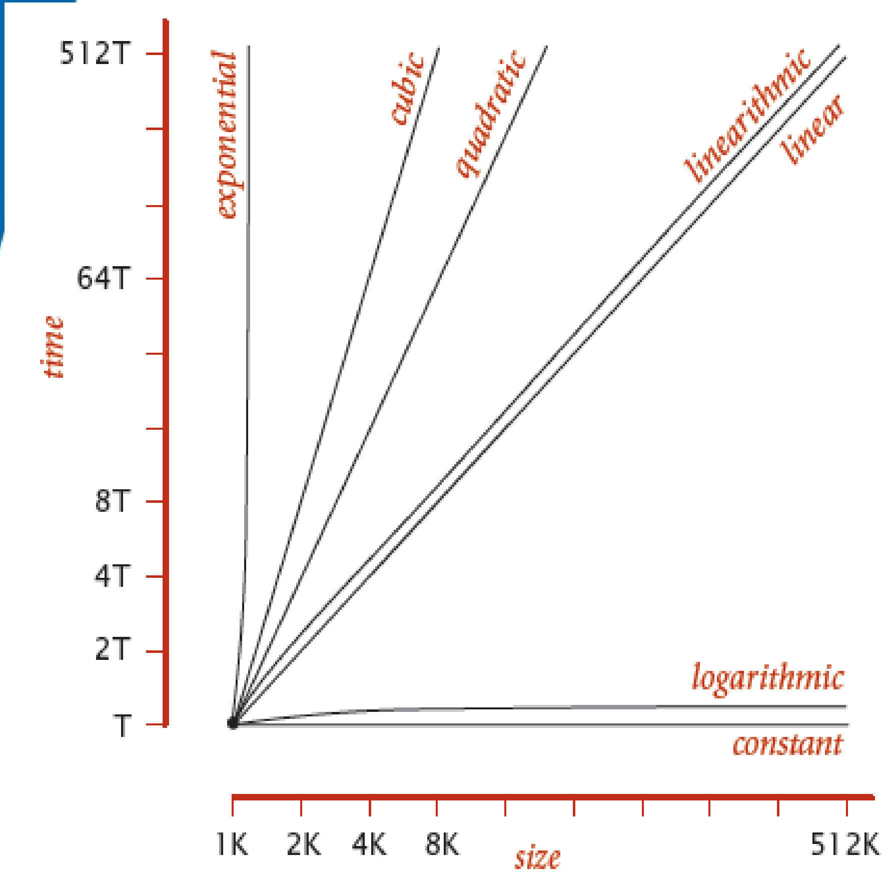
```
public void tiempoWhile(int n){
    int i=1;
    while (i<n) {
        Console.Write(i);
        i=i*2;
    }
}
```

$$T(n) = \sum_{i=1}^{\log(n)+1} cte1 = (\log(n) + 1) * cte1 = cte1 \log(n) + cte1$$

Algunas Funciones

Ordenadas en forma creciente	Nombre
1	Constante
log n	Logaritmo
n	Lineal
n log n	n Log n
n^2	Cuadrática
n^3	Cúbica
c^n $c>1$	Exponencial

Algunas Funciones



Este conjunto de funciones en general es suficiente para describir la tasa de crecimiento de los algoritmos típicos

Cuadro Comparativo

Costo		$n=10^3$	Tiempo	$n=10^6$	Tiempo
Logarítmico	$\log_2(n)$	10	10 segundos	20	20 segundos
Lineal	n	10^3	16 minutos	10^6	11 días
Cuadrático	n^2	10^6	11 días	10^{12}	30.000 años

Orden de ejecución del algoritmo

Cantidad de operaciones

Tiempo total del algoritmo

$n = 10^3$

Cantidad de operaciones

$n = 10^6$

Problema

Considerando que un algoritmo requiere $f(n)$ operaciones para resolver un problema y la computadora procesa 100 operaciones por segundo.

Si $f(n)$ es:

a.- $\log_{10} n$

b.- $\sqrt{n}$

Determine el tiempo en segundos requerido por el algoritmo para resolver un problema de tamaño n=10000.

Problema

Suponga que Ud. tiene un algoritmo C# con un tiempo de ejecución exacto de $10n^2$. ¿En cuánto se hace más lento C# cuando el tamaño de la entrada n aumenta:……….?

a.- El doble

b.- El triple

Orden de magnitud

- El T(n) de un algoritmo nos da una estimación del tiempo que tarda para una entrada dada.
- El ORDEN DE MAGNITUD que tiene un T(n) dado, nos permite determinar que tan bueno es ese T(n) respecto de otros T(n).
- Dos algoritmos pueden tener distintos T(n) pero el mismo ORDEN DE MAGNITUD, lo cual nos da la pauta que son algoritmos con la misma EFICIENCIA

Orden de magnitud

- Si se tienen los siguientes T(n) en los siguientes algoritmos
  - Algorítmo 1: $T(n) = n^2 + 20\ n + cte1$
  - Algorítmo 2: $T(n) = 500000\ n + 1000000\ cte1$
  - Algorítmo 3: $T(n) = 3\ n^2 + n$
  - Algorítmo 4: $T(n) = n^3 + cte1$

  … ¿que algoritmo es mejor? ¿cuál es el peor?
- El ORDEN DE MAGNITUD de un algorítmo indica como crece el T(n) de este cuando crece el tamaño de su entrada.

Orden de magnitud (cont)

- Algunos ORDENES DE MAGNITUD que pueden tener los algorítmos (ordenados en forma creciente) son:
  - $O(1)$ se lee, orden constante
  - $O(\log(n))$ orden logaritmico
  - $O(n)$ orden lineal
  - $O(n \log(n))$
  - $O(n^2)$ orden cuadrático
  - $O(n^3)$ orden cúbico
  - $O(c^n)$ con $c > 1$ orden exponencial

¿Cómo se calcula el orden de magnitud?

Para expresar el orden de magnitud de un T(n) dado, utilizaremos la notación de Big-Oh.

- Decimos que T(n) es O(f(n)). Se lee: T(n) tiene orden de magnitud f(n)

  Si existen constantes $c > 0$ y n_0 tales que:

  $T(n) \leq c\, f(n)$ para todo $n \geq n_0$

Por ejemplo T(n) = 4n3 es O(n3) porque existen c=4 y $n_0=1$ tales que $T(n) = 4n^3 \leq 4n^3$ para todo $n \geq 1$

Reglas utiles para el Cálculo del orden de magnitud

Si $T_1(n)$ es $O(f(n)$ y $T_2(n)$ es $O(g(n)$

- **Regla de la suma**

  $T_1(n) + T_2(n)$ es $O(max(f(n),g(n)))$

- **Regla del producto**

  $T_1(n) * T_2(n)$ es $O(f(n)*g(n))$

Cálculo del Orden

- Antes nos preguntamos, cuál de los siguientes T(n) es el peor y cuál el mejor:
  - Algorítmo 1: $T(n) = n^2 + 20\ n + cte1$
  - Algorítmo 2: $T(n) = 500000\ n + 1000000\ cte1$
  - Algorítmo 3: $T(n) = 3\ n^2 + n$
  - Algorítmo 4: $T(n) = n^3 + cte1$

Cálculo del Orden

- Tomemos el primero:
  - Algorítmo 1: $T(n) = n^2 + 20\, n + cte1$

- Primero hay que estimar el orden en base al término con el orden mas grande
- Por ejemplo:
  - 1er término: n^2 es $O(n^2)$
  - 2do término: $20\, n$ es $O(n)$
  - 3er término: Cte1 es O(cte)

Cálculo del Orden (cont)

- Para decir que $T(n) = n^2+20n+cte1$ es $O(n^2)$ tenemos que poder demostrar que:
  - Existen constantes c_0 y n_0 tales que:

    $T(n) \le c_0 n^2$ para todo $n \ge n_0$

Una forma facil de hallar las constantes c_0 y n_0 es hacer la demostración del orden termino a termino.

Cálculo del Orden (cont)

- $T(n) = n^2+20n+cte1$

$n^2 \leq 1\ n^2$ para todo $n \geq 1$

$20n \leq 20\ n^2$ para todo $n \geq 1$

$cte1 \leq cte1\ n^2$ para todo $n \geq 5$ (por poner algo)

- Si cada uno de los terminos de la izquierda es menor que lo de la derecha, la suma de lo de la izquierda será menor que la suma de lo de la derecha:

$$\underbrace{n^2 + 20n + cte1}_{T(n)} \leq \underbrace{(1+20+cte1)}_{c_0}\ n^2 \text{ para todo } n \geq \underbrace{5}_{n_0}$$

como $T(n) \leq c_0\ n^2$ para todo $n \geq n_0$ entonces $T(n)$ es $O(n^2)$

Cálculo del Orden

- Siguiendo con el mismo razonamiento deberiamos poder demostrar que:
  - Algorítmo 1: $T(n) = n^2 + 20\ n + cte1$
    - es orden $O(n^2)$ La misma eficiencia que Algoritmo 3
  - Algorítmo 2: $T(n) = 500000\ n + 1000000\ cte1$
    - es orden $O(n)$ El MAS conveniente!!!!
  - Algorítmo 3: $T(n) = 3\ n^2 + n$
    - es orden $O(n^2)$ La misma eficiencia que Algoritmo 1
  - Algorítmo 4: $T(n) = n^3 + cte1$
    - es orden $O(n^3)$ El MENOS conveniente

Sobre Big-Oh, Omega y Theta

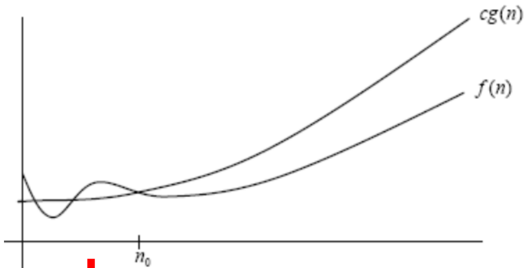
- Cuando decimos que el orden de $T(n)$ es $O(n^2)$ por ejemplo, estamos diciendo que

  $\exists\ c_0$ y n_0 tal que $T(n) \leq c_0\, n^2$ para todo $n \geq n_0$

- n^2 DEBE SER el MENOR de los ordenes posibles que corroboran la desigualdad anterior
- Por simplicidad vamos a tomar a Big-Oh como la cota mas ajustada para $T(n)$. Sin embargo...

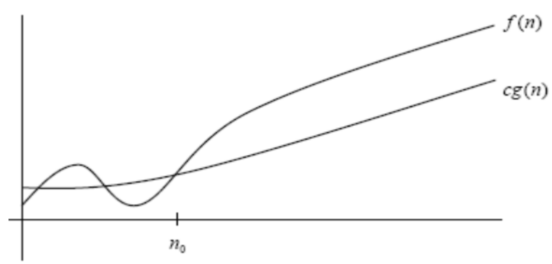
Sobre Big-Oh, Omega y Theta

- Big-Oh: O(f(n))
  - Decimos que T(n) es O(f(n))

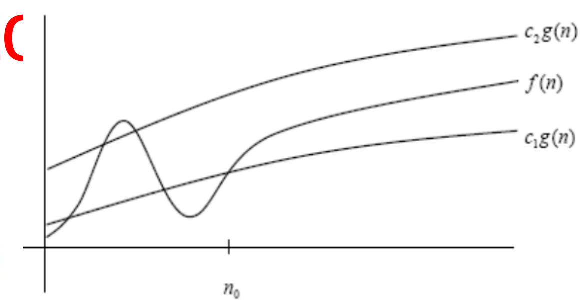


$\exists\ c_0$ y n_0 tal que $T(n) \leq c_0\ n^2$ para todo $n \geq n_0$

- Omega: Ω(f(n))
  - Decimos que T(n) es Ω(f(n))



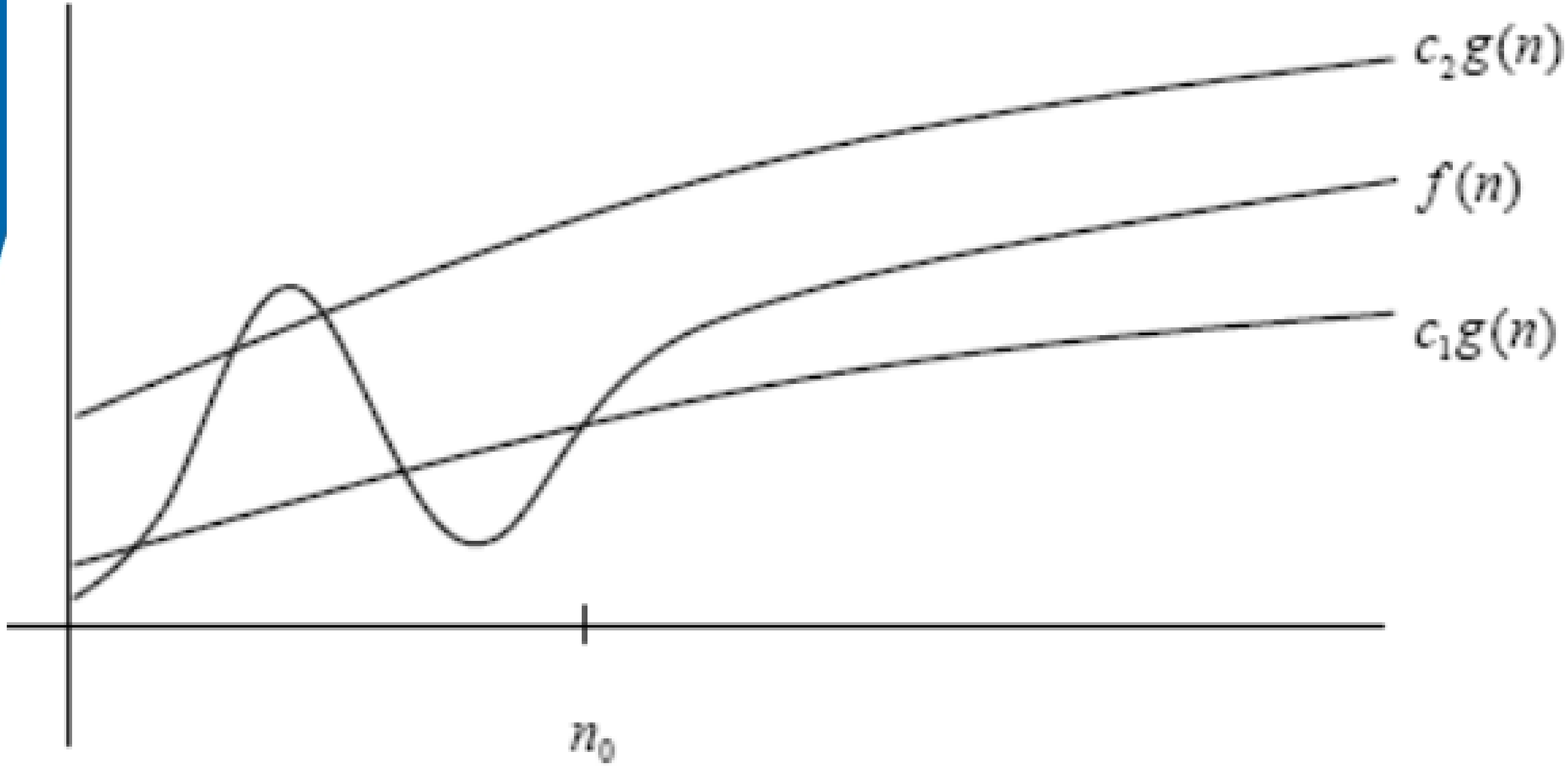
$\exists\ c_0$ y n_0 tal que $T(n) \geq c_0 n^2$ para t



- Theta: Θ(f(n))
  - Decimos que T(n) es Θ(f(n))

Si y solo si T(n) es O(f(n)) y T(n) es Ω(f(n))

Theta - Θ(f(n))



Fin