UTEC

UNIVERSIDAD DE INGENIERÍA
Y TECNOLOGÍA

CS2013
Programación III
Unidad 3 - Semana 7 - Análisis de Complejidad Algorítmica.

Rubén Rivas

Objetivos

Al finalizar la sesión, el alumno analizara la complejidad de los algoritmos y podrá comparar y clasificarlos por su eficiencia.

Temas semana 7

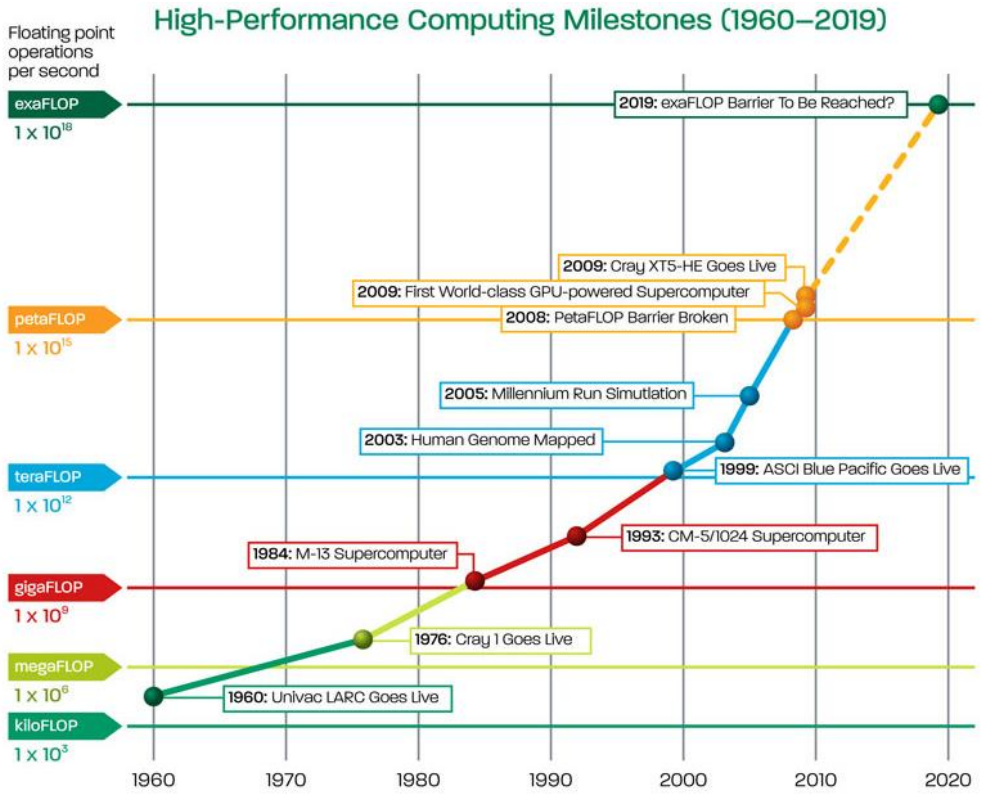
- Motivación
- Análisis de los Algoritmos
- Tipos de análisis
- Notaciones

Motivación

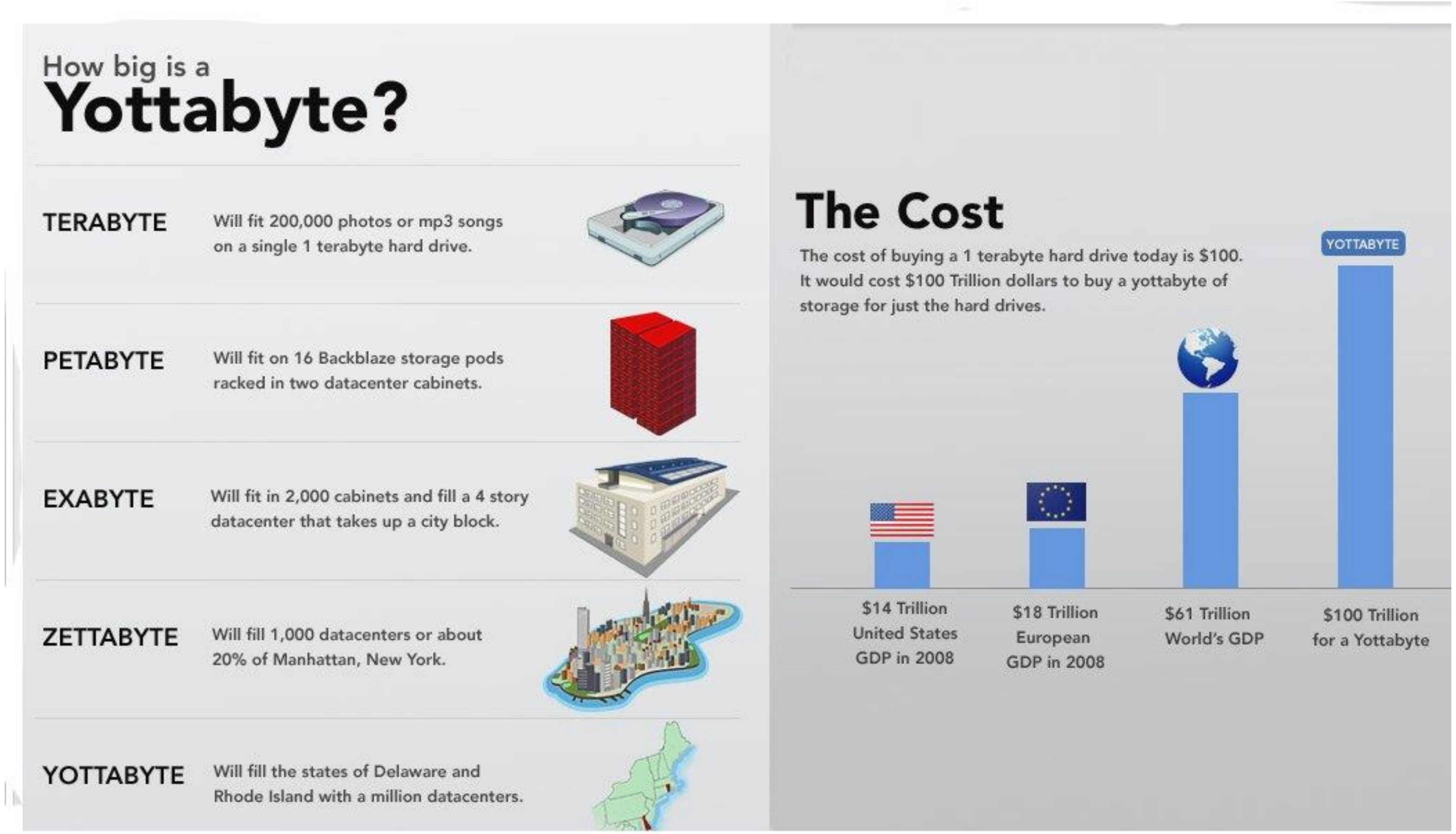
- Mayor capacidad computacional.
- Mayor volumen de información.
- Problemas más complejos.

Capacidad computacional

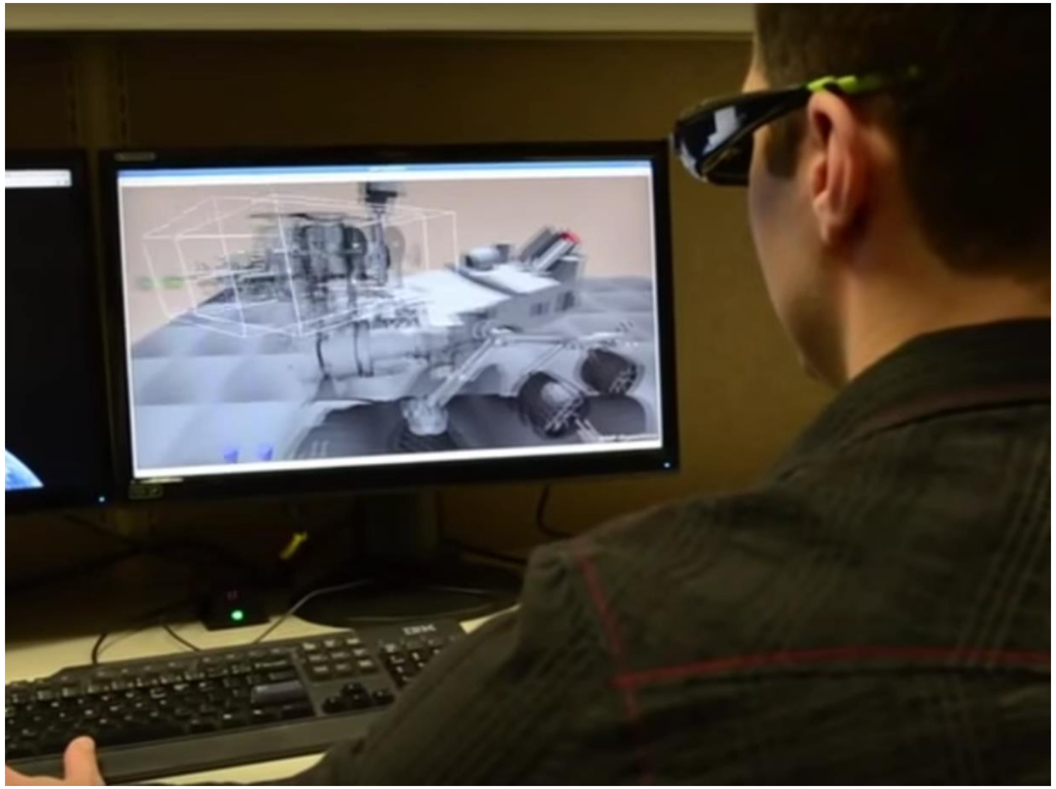
- **Crecimiento Exponencial**
- **Diversidad en los recursos computacionales.**



Volumen de Información



Problemas más complejos



Análisis de Algoritmos

*Es el proceso de identificar el **tiempo**, **espacio** u **otro recurso** que es necesario para ejecutar el algoritmo.*

¿Como se identificar el tiempo necesario?

- Medir el tiempo utilizando un cronometro.
- Contando el número de instrucciones.
- Evaluando el orden de crecimiento.

Utilizando un cronometro

```
#include <chrono>
using namespace std;
using namespace std::chrono;
…
auto start = high_resolution_clock::now();
auto f = factorial(100);
auto end = high_resolution_clock::now();
cout  << "factorial de: " << 100
      << " demoro: "
      << duration_cast<microseconds>(end - start).count()
      << endl;
```

¿De que depende?

- Implementación. (lenguaje, instrucción)
- Computadora. (Hardware y Software)
- Instrucción.
- Aplicaciones en ejecución.
- Algoritmo.

Ventajas y desventajas

Ventajas

- Es la forma mas sencilla para sistemas complejos.
- Se ajusta a modelos probabilísticos.

Desventajas

- Hay muchos factores que afectan la medición.
- Mediciones bajo el mismo set de datos pueden variar significativamente.

UTEC
UNIVERSIDAD DE INGENIERÍA Y TECNOLOGÍA

Contando el número de instrucciones

```
int fun1(int n) {
    return n*(n+1)/2;
}
```

$fun1 \to C_1$

```
int fun2(int n) {
    int sum = 0;
    for (int i = 0; i < n; i++) {
        sum += i;
    }
    return sum;
}
```

$fun2 \to C_1 n + C_2$

```
int fun3(int n) {
    int sum = 0;
    for (int i = 1; i < n; i++) {
        for (int j = 1; j <= i; j++) {
            sum++;
        }
    }
    return sum;
}
```

$fun3 \to C_1 n^2 + C_2 n + C_3$

Contando el número de instrucciones

Se asume que todas las operaciones en un programa **consumen el mismo tiempo** y que ese **tiempo unitario es constante**, por ejemplo las siguientes operaciones tiene una duración similar:

- Operaciones matemáticas
- Comparaciones
- Asignaciones
- Accesos a memoria.

Se debe desarrollar una **función matemática** que evalúa la eficiencia basada en el **número de instrucciones**.

Ventajas y desventajas

Ventajas

- Es **independiente** de otros **factores externos**.
- Se obtiene un **único valor** de eficiencia para un mismo set de datos.

Desventajas

- La **ecuación** que se genera podría ser **singular**.
- Puede **dificultar** la **comparación de eficiencias**.

Orden de Crecimiento

- Se basa en el **método de conteo de instrucciones.**
- **No busca ser preciso** en el calculo.
- Busca identifica el comportamiento, la tendencia de crecimiento.
- Expresar la **eficiencia de ejecución** en **función al tamaño del input**.
- Simplifica la función de **conteo de instrucciones** generando una nueva función basada en **el termino o factor de mayor grado**.
- Su objetivo es evaluar la eficiencia en situaciones de gran cantidad de información.

¿Como medir el tamaño del input?

Se determina el factor que incrementa el uso del recurso.

Ejemplos:

- **Número de operaciones (n):**

```
long long factorial(int n);
```

- **El tamaño del vector o data**:

```
long long search(vector<int> data, int value);
```

¿Comó determinar el orden de crecimiento?

- Desarrollar la **ecuación de tiempo de ejecución** o del **recurso** que se quiera **evaluar**.
- Se simplifica la ecuación **eliminando** los factores de **menor grado**, **conservando** el factor de **mayor grado**.
- Evaluar la tendencia de crecimiento.

$$C < \log\log(n) < \log(n) < n^{\frac{1}{3}} < n^{\frac{1}{2}} < n < n^2 < n^3 < n^4 < 2^n < n^n$$

Ventajas y desventajas

Ventajas

- Facilita la comparación entre algoritmos
- Se obtiene un único valor para un mismo set de datos.

Desventajas

- Se **pierde precisión** por la simplificación que **en casos prácticos podría ser muy útiles**.

Orden de crecimiento – Evaluación matemática

La función f(n) se dice que está creciendo rápidamente que g(n) si:

Asumimos lo siguiente:

$$n \geq 0$$
$$Tiempo \geq 0$$
$$f(n),\ g(n) \geq 0$$

$$\lim_{n \to \infty} \frac{f(n)}{g(n)} = \infty \quad \textbf{OR} \quad \lim_{n \to \infty} \frac{g(n)}{f(n)} = 0$$

$$\lim_{n \to \infty} \frac{2*n+5}{2*n^2+3*n+4} = \lim_{n \to \infty} \frac{2/n+5/n^2}{2+3*1/n+4/n^2} = \lim_{n \to \infty} \frac{0+0}{2+0+0} = 0$$

f(n) es un algoritmo poco eficiente (malo) en comparación a g(n) cuando n tiende al infinito

Ejemplos

Calcular el orden de crecimiento y compararlos.

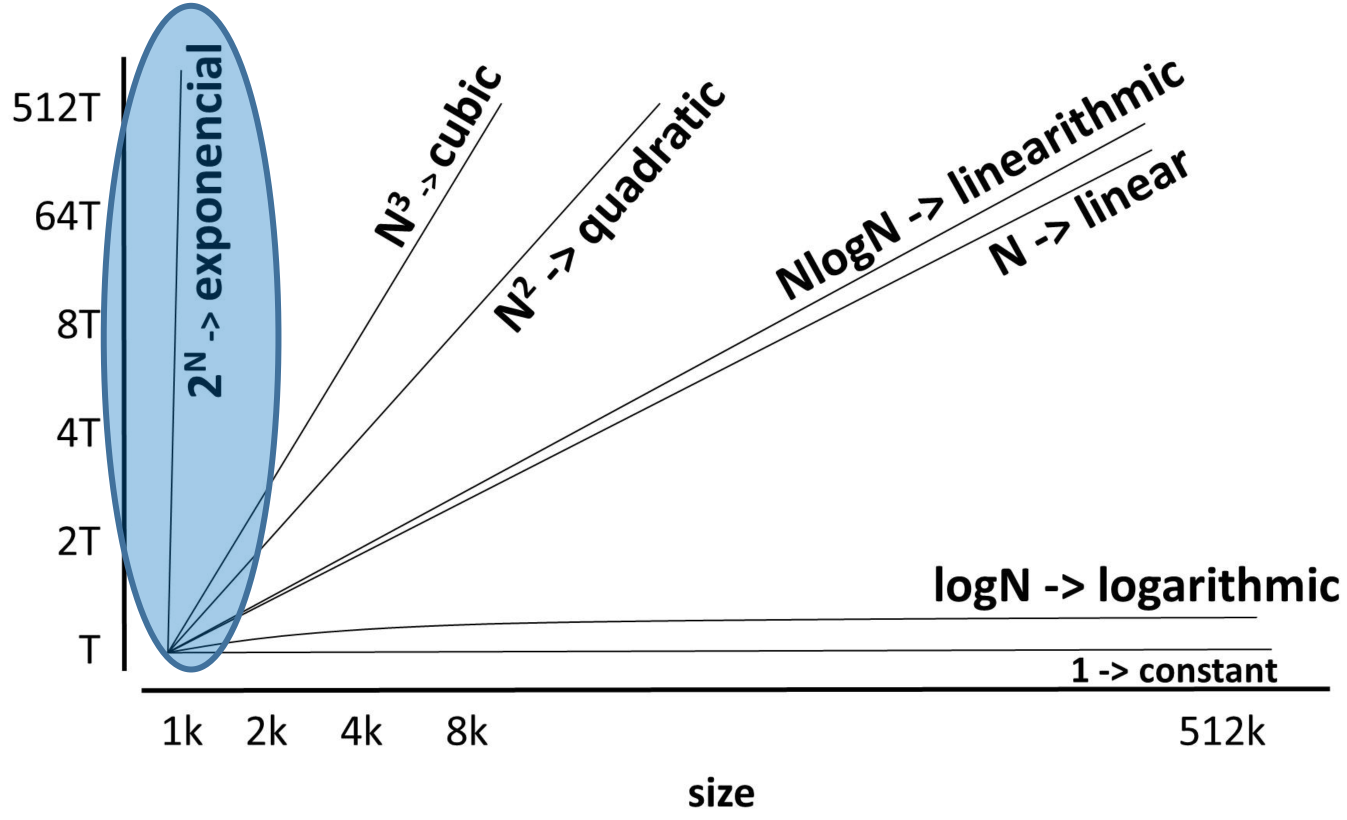
- $n^2 + 2n + 1000$
- $n^3 + 100000000n + 10^{1000}$
- $log(n) + n + 100$
- $0.0000001nlog(n) + 300000n$
- $2n^{20} + 3^n$

Ejemplos

Calcular el orden de crecimiento y compararlos.

- $n^2 + 2n + 1000 \rightarrow \boldsymbol{n^2}$
- $n^3 + \ 100000000n \ + \ 10^{1000} \rightarrow \boldsymbol{n^3}$
- $log(n) + \ n \ + \ 100 \ \rightarrow \boldsymbol{n}$
- $0.0000001nlog(n) + \ 300000n \ \rightarrow \boldsymbol{nlog(n)}$
- $2n^{20} + 3^n \ \rightarrow \boldsymbol{3^n}$

Orden de crecimiento



NP Non-polynomial

Problemas del milenio: P vs NP

Best, average and worst case

```
int getSum(int arr[], int n) {
    int sum = 0;
    if (n%2 != 0)
        return 0;
    for (int i = 0; i < n; i++)
        sum = sum + i;
    return sum;
}
```

Best case: Constante, ocurre cuando el arreglo contiene solo números impares o la cantidad es mínima (0, 1).

Average case: Linear y se basa en el promedio de todos los posibles casos.

Worst case: Linear, ocurre cuando todos los números son pares

Notación

Big O: Representa el límite exacto o límite superior.

Theta: Representa el limite exacto.

Omega: Representa el límite exacto o límite inferior.

Son notaciones que se usan para representa el peor (Big O) el promedio (Theta) y el mejor caso (Omega)

En general se utiliza la notación **Big O** considerando que es factible el peor escenario y evitar minimizar sus efectos.

Notación matemática de Big O

Decimos que $f(n) = O(g(n))$ si existe una constante C y $n0$ tal que $f(n) \leq C*g(n)$ para todos $n \geq n0$

Ejemplo: $f(n) = 2*n + 3$, puede ser escrito como $O(n)$

-> $f(n) \leq C*g(n)$ para todo $n \geq n0$

-> $2*n + 3 \leq C*n$ para todo $n \geq n0$

De forma práctica el valor de C se calcula tomando la constante del mayor término de f(n) y se agrega 1.

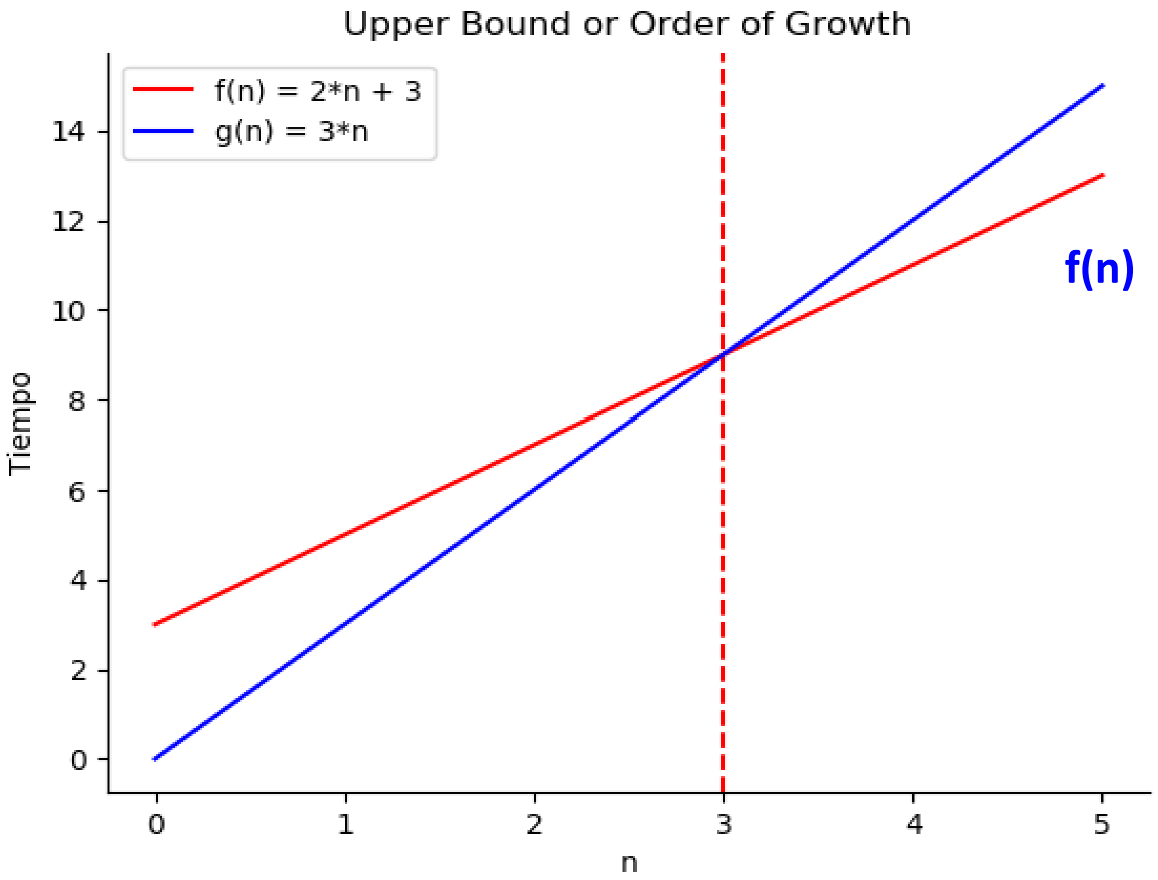
-> $C = 3$

-> $2*n + 3 \leq 3*n$

-> $3 \leq n$

-> $n0 = 3$

Notación Big O



f(n) ∈ O(n)

Omega Ω Notation - Lower Bound

f(n) = Ω(g(n)) si existe una constante positiva C y n0 tal que 0 ≤ C*g(n) ≤ f(n) para todo n ≥ n0

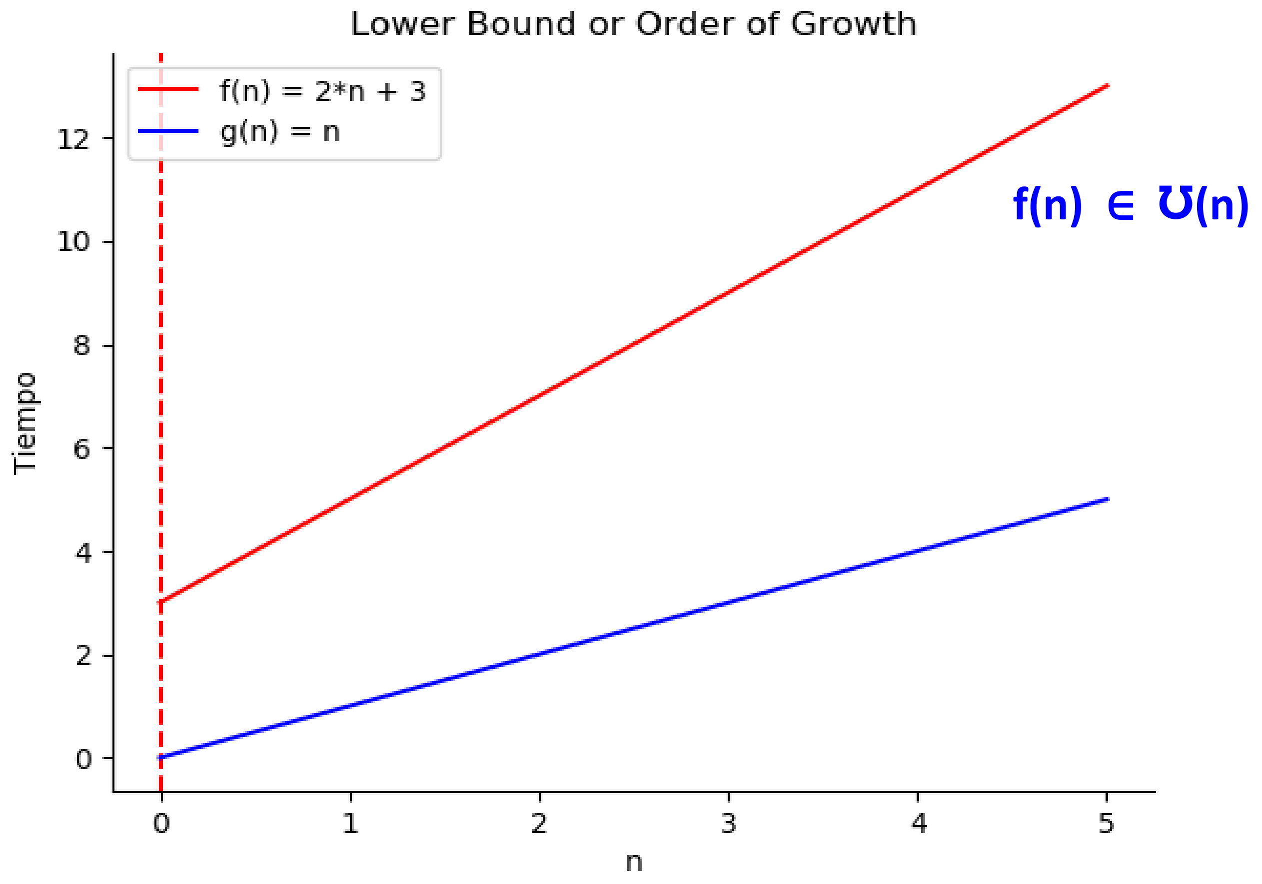
Ejemplo: f(n) = 2*n + 3, puede ser escrito como Ω(n)

De forma práctica el valor de C se calcula tomando la constante del mayor término de f(n) y se disminuye en 1.

0 ≤ n ≤ 2*n + 3 -> n0 = 0 , C = 1

Nota: Tanto como O y Ω cubren lo mismo, entonces podemos poner el mismo orden de crecimiento.

Omega Ω Notation



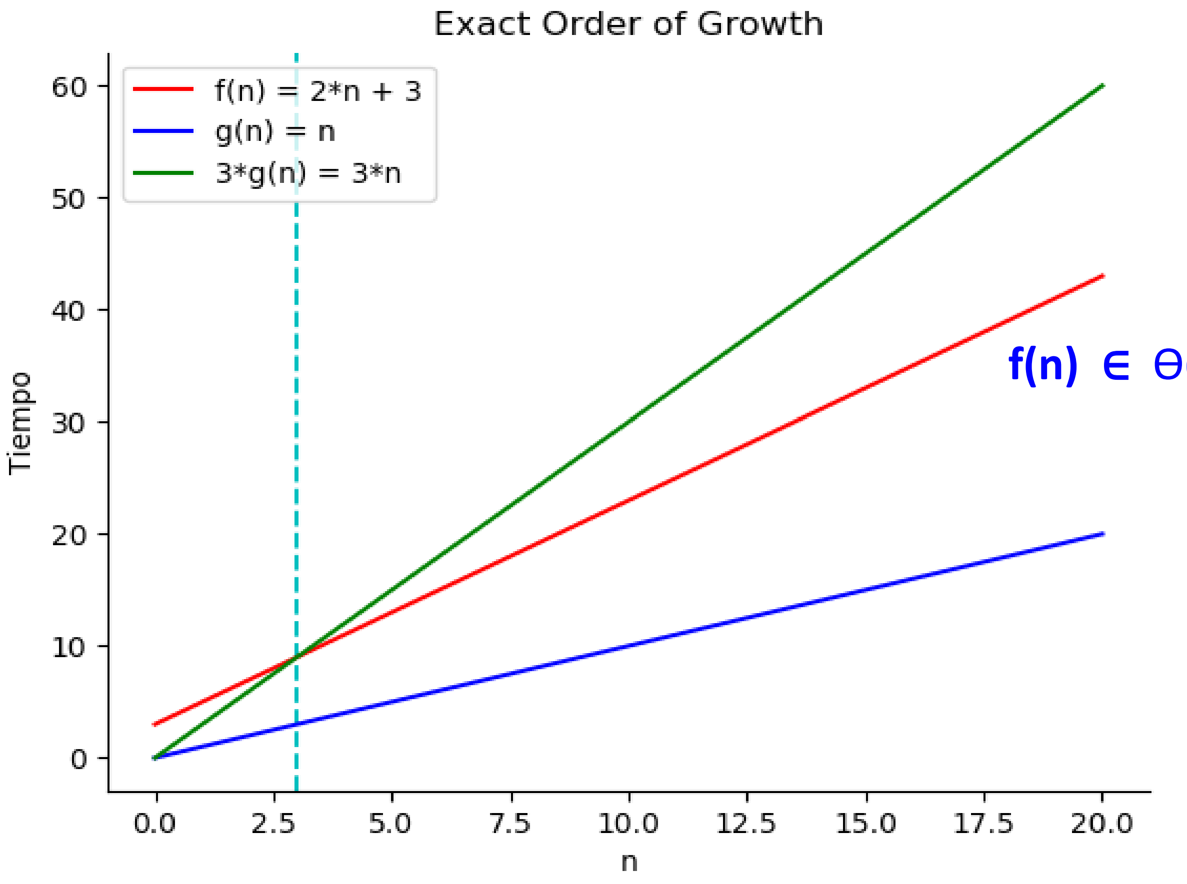
Theta ϴ Notation - Exact Order of Growth

Representa el límite exacto del orden de crecimiento.

$f(n) = \Theta(g(n))$ si existe constantes positivas C1, C2 y n0 tal que,
$0 \le C1*g(n) \le f(n) \le C2*g(n)$, para todo $n \ge n0$

Ejemplo: $f(n) = 2*n + 3$, asumiendo que el orden de crecimiento es $\Theta(n)$
-> $C1*g(n) \le f(n) \le C2*g(n)$, para todo $n \ge n0$
-> $C1 = 1$, $C2 = 3$
-> $n \le 2*n + 3 \le 3*n$
-> $n \ge 0$ and $n \ge 3$
-> $n0 = 3$

Theta Θ Notation



$f(n) \in \Theta(n)$

Space Complexity

Orden de crecimiento de la memoria (o RAM) en términos del tamaño de la entrada.

```
int fun1(int n) {
    return n*(n+1)/2;
}
```

```
int fun2(int n) {
    int sum = 0;
    for (int i = 0; i < n; i++) {
        sum += i;
    }
    return sum;
}
```

Space Complexity

Orden de crecimiento de la memoria (o RAM) en términos del tamaño de la entrada.

```
int fun1(int n) {
    return n*(n+1)/2;
}
```

Variables: n

Complejidad espacial: Constante

O(1) o Θ(1)

```
int fun2(int n) {
    int sum = 0;
    for (int i = 0; i < n; i++) {
        sum += i;
    }
    return sum;
}
```

Variables: sum, i, n

Complejidad espacial: Constante

O(1) o Θ(1)

Space Complexity

```
int sumArr(int arr[], int n) {
    int sum = 0;
    for (int i = 0; i < n; i++) {
        sum += arr[i];
    }
    return sum;
}
```

Variables: arr, n, i

Complejidad espacial: Linear $\Theta(n)$

Auxiliary Space

Orden de crecimiento del espacio de memoria o espacio temporal en términos del tamaño de la entrada.

```
int sumArr(int arr[], int n) {
    int sum = 0;
    for (int i = 0; i < n; i++) {
        sum += arr[i];
    }
    return sum;
}
```

Espacio extra: $\Theta(1)$

Complejidad espacial: $\Theta(n)$

Combinando ordenes de crecimiento secuenciales

Se aplica la ley de adición

$$O\big(f(n)\big) + O\big(g(n)\big) = O\big(f(n) + g(n)\big)$$

ejemplo:

$$f(n) = \log(n)$$

$$g(n) = n^2$$

$$O(log(n)) + O(n^2) = O(n^2)$$

Combinando ordenes de crecimiento anidadas

Ley de multiplicación de O().

- Usado con una secuencia de instrucciones.

$$O\big(f(n)\big) * O\big(g(n)\big) = O\big(f(n) * g(n)\big)$$

ejemplo:

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

$$g(n) = O(n^2)$$

$$O(n) * O(n^2) = O(n^3)$$

Explorando lo aprendido

- **Analisis Asintotico**
- **Big O - Upper Bound**
- **Omega Ʊ - Lower Bound**
- **Theta ɵ - Exact Bound**
- **Complejidad espacial**

Bibliografía:

- **B. Stroustrup The c++ Programming Language** 4t edition
- **C++ Primer, Fifth Edition; 2013**; Stanley B. Lippman, Josée Lajoie, Barbara E. Moo