Estructuras y Tipos de Datos

Análisis de Algoritmos

TEMA II - Análisis de Algoritmos

CONTENIDO

- Tiempo de ejecución
- Eficiencia y Orden de un algoritmo
- Criterios y cálculo de la eficiencia de un algoritmo

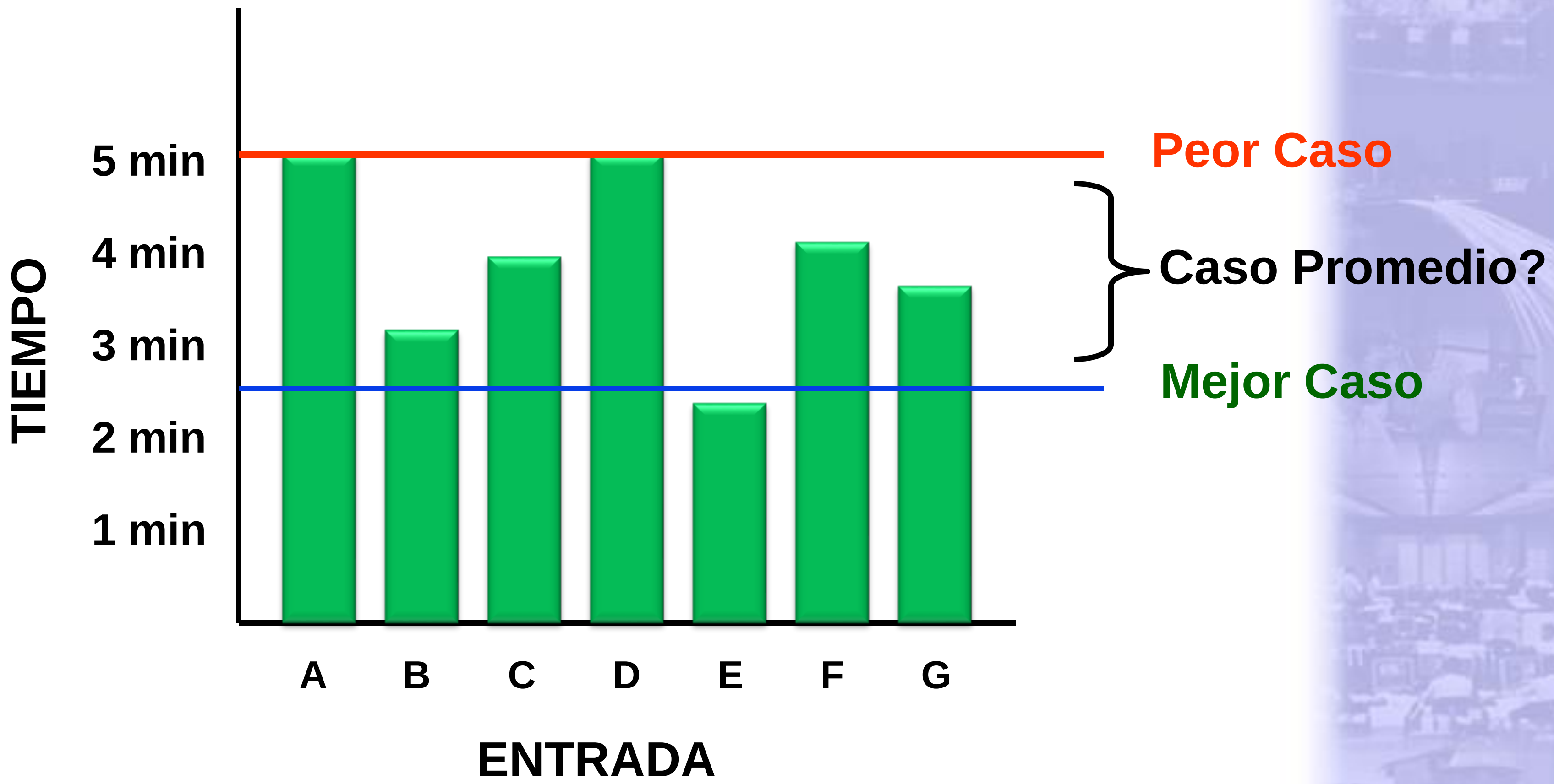
Tiempo de Ejecución

Medidas de desempeño

- *Cantidad de almacenamiento principal requerido por sus variables*
- *Cantidad de tráfico que genera en una red de computadoras*
- *Cantidad de información que debe moverse desde y hacia las unidades de almacenamiento secundario*
- *Simplicidad del algoritmo*
- *Tiempo de cómputo*

Tiempo de Ejecución

Medidas de desempeño

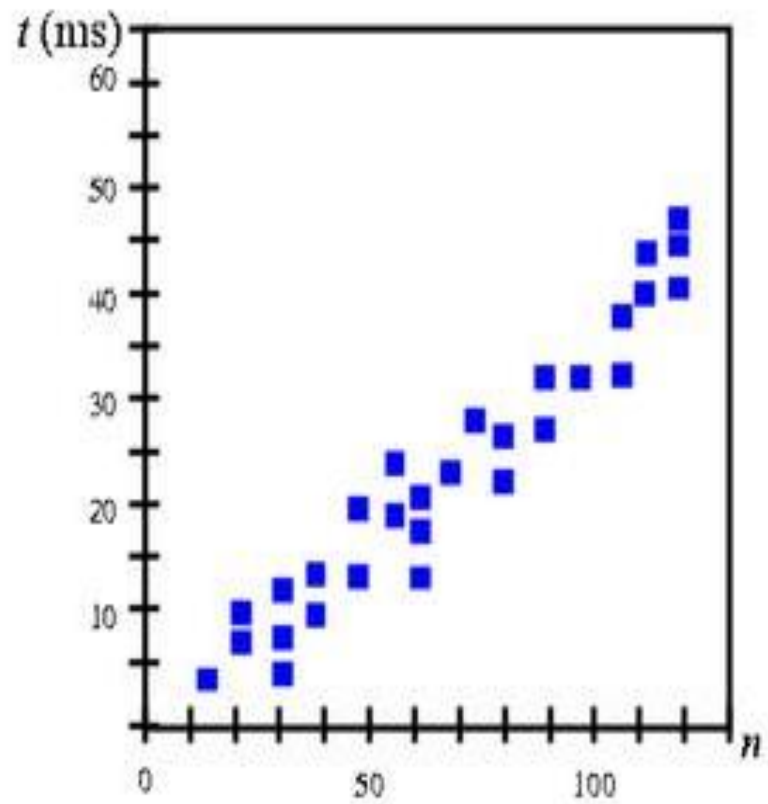


Tiempo de Ejecución

Cómo podemos medir el tiempo de ejecución de un algoritmo?

•***Estudio Experimental:***

- *Escribir un programa que implemente el algoritmo.*
- *Ejecutar el programa con un conjunto de datos de tamaño y composición variable.*
- *Usar algún método para obtener una medida precisa del tiempo de ejecución actual. Ejemplo Java:* `System.CurrenTimeMillis()`



Tiempo de Ejecución

Problemas del Estudio Experimental

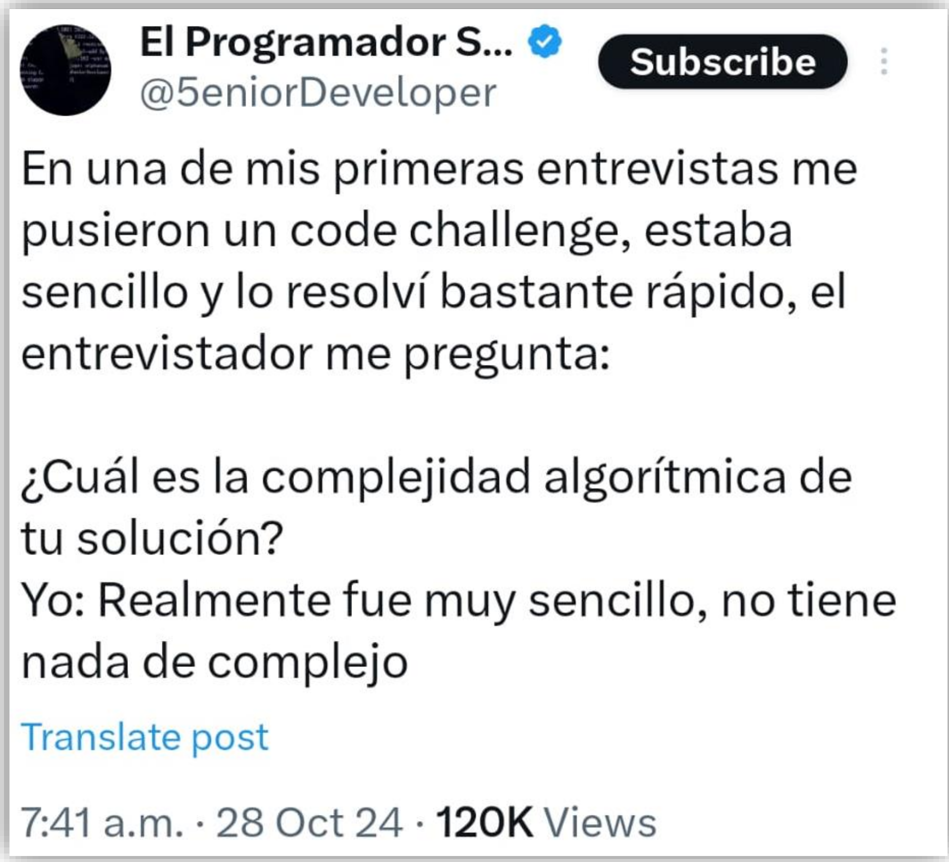
- *Es necesario implementar y probar el algoritmo para determinar su tiempo de ejecución.*
- *El experimento puede ser hecho sólo sobre un conjunto limitado de entradas, por lo que puede no ser indicativo sobre otros conjuntos de entradas no contempladas en el experimento.*
- *Para poder comparar dos algoritmos, debe usarse el mismo ambiente de hardware y software.*

Alternativa

Utilizar una metodología general para analizar el tiempo de ejecución de un algoritmo que:

- *Use una descripción de alto-nivel del algoritmo en vez de una implementación.*
- *Tome en consideración todos los posibles conjuntos de entrada*
- *Permita evaluar la eficiencia de un algoritmo, independientemente de la plataforma de hardare y software en el que se implemente.*

Eficiencia y Orden de un algoritmo



Eficiencia y Orden de un Algoritmo

https://www.youtube.com/watch?v=XMUe3zFhM5c

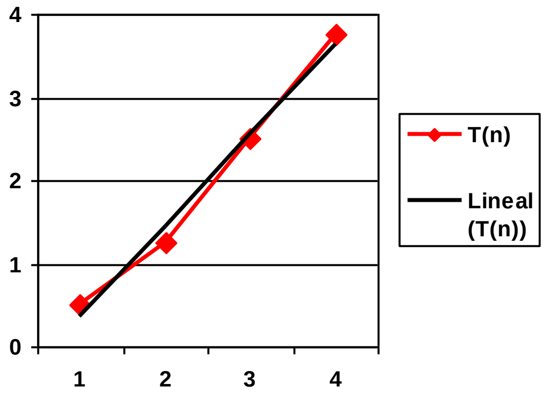
Eficiencia y Orden de un algoritmo

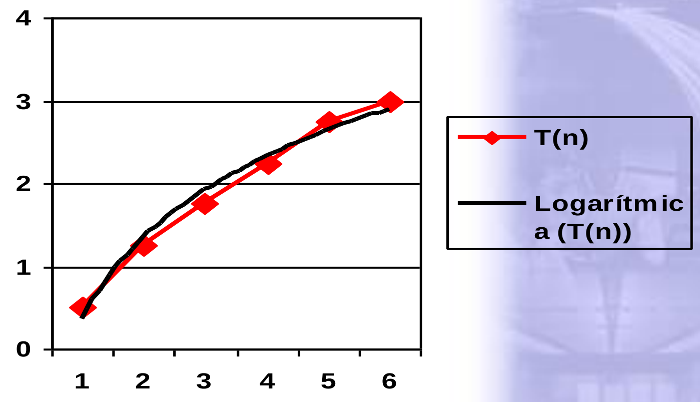
Eficiencia – elementos de análisis

Complejidad

Está referida a conseguir una función que acote el tiempo de ejecución del algoritmo T(n), a esta la llamamos orden del algoritmo O(n).

Ejemplo





El número de comparaciones

Es la cantidad de comparaciones entre claves que realiza el algoritmo durante su ejecución.

El número de movimientos

Es la cantidad de intercambios de registros de datos que realiza el algoritmo durante su ejecución.

Eficiencia y Orden de un algoritmo

Complejidad temporal

Es el tiempo requerido (unidades de tiempo) por un algoritmo para procesar una entrada de tamaño *n*.

Sea $T(n)$ el tiempo de corrida de algún programa, debemos asumir:

1. El argumento *n* es un entero no negativo, y
2. $T(n)$ es no negativo para todos los argumentos de *n*

Casos Complejidad

- **Complejidad en el mejor de los casos**: se define T(n) como el tiempo mínimo de corrida para todas las entradas de tamaño n.
- **Complejidad en el peor de los casos**: se define T(n) cómo el tiempo máximo de corrida para todas las entradas de tamaño n.
- **Complejidad promedio:** otra medida común de desempeño es Tavg(n), el tiempo promedio de corrida del programa sobre todas las entradas de tamaño n. Aún cuando el tiempo promedio es una medida más realista, en la práctica es más difícil de calcular que el peor de los casos.

Medición del Tiempo de Ejecución de un Algoritmo

- **Operaciones primitivas:** operaciones de bajo nivel que pueden ser comúnmente implementadas en la mayoría de los lenguajes de programación y que pueden ser identificadas en pseudo-código.
  - Llamar a un procedimiento o función;
  - Ejecutar una operación aritmética (ej. una suma);
  - Comparar dos números, etc.

- Inspeccionando el pseudo-código se puede **contar** el número de operaciones primitivas ejecutadas por un algoritmo.

Medición del Tiempo de Ejecución de un Algoritmo

Ejemplo:

```
Algoritmo ArregloMaximo(A,n)
    Entrada: un arreglo A guardando números enteros
    Salida: el elemento máximo de A
Max ← A[0]                          (1 vez)
for i ← 1 to n - 1 do               (n veces)
    if Max < A[i] then              (n-1 veces en peor caso)
        Max ← A[i]                  (n-1 veces en peor caso)
return Max                          (1 vez)
```

Medición del Tiempo de Ejecución de un Algoritmo

Casos de Complejidades

- ***Complejidad en el mejor de los casos*:** se define T(n) como el tiempo mínimo de corrida para todas las entradas de tamaño n.
- ***Complejidad en el peor de los casos*:** se define T(n) cómo el tiempo máximo de corrida para todas las entradas de tamaño n.
- ***Complejidad promedio*:** otra medida común de desempeño es $T_{avg}(n)$, el tiempo promedio de corrida del programa sobre todas las entradas de tamaño n. Aún cuando el tiempo promedio es una medida más realista, en la práctica es más difícil de calcular que el peor de los casos.

Medición del Tiempo de Ejecución de un Algoritmo

Casos de Complejidades- El peor de los casos

- Es un límite superior del tiempo de corrida
- Frecuentemente ocurre
- Es más fácil de calcular que el promedio.

Notación Asintótica

Cuando observamos entradas de tamaño lo suficientemente grandes para que sólo el orden de crecimiento de la complejidad sea relevante, estamos estudiando la eficiencia asintótica de los algoritmos.

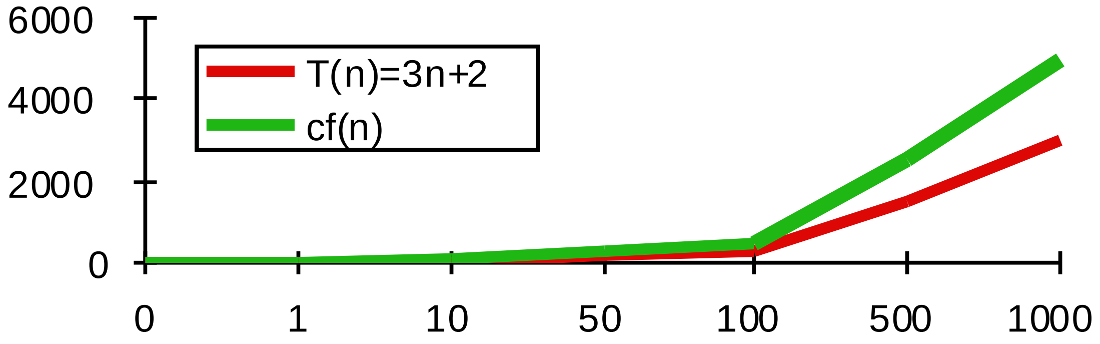
Medición del Tiempo de Ejecución de un Algoritmo

Casos de Complejidades- El peor de los casos – Notación O

•Para especificar la **cota superior** de la razón de crecimiento de T(n) se usa la notación **O(*f*(*n*)).**

Formalmente, decimos que **T(*n*) es O(*f*(*n*))** si existe un entero n_0 y una constante $c > 0$ tal que para todos los enteros $n \geq n_0$, tenemos que

$$0 \leq T(n) \leq cf(n).$$

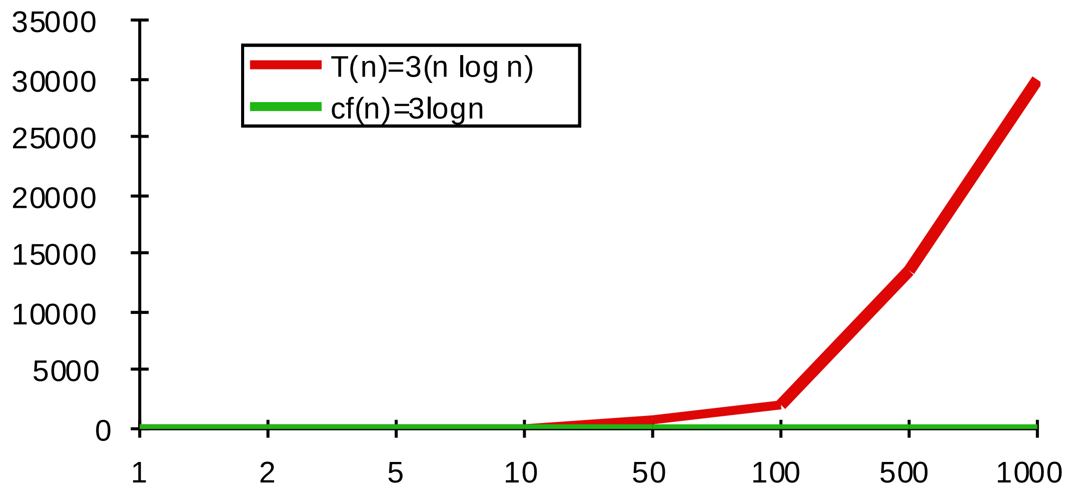


Medición del Tiempo de Ejecución de un Algoritmo

Casos de Complejidades- El mejor de los casos – Notación Ω

- Para especificar la **cota inferior** de la razón de crecimiento de T(n) se usa la notación $\Omega(f(n))$.
- Formalmente, decimos que **T(*n*) es Ω(*f*(*n*))** si existe un entero n_0 y una constante $c > 0$ tal que para todos los enteros $n \geq n_0$, tenemos que

$$\mathbf{0 \leq cf(n) \leq T(n).}$$



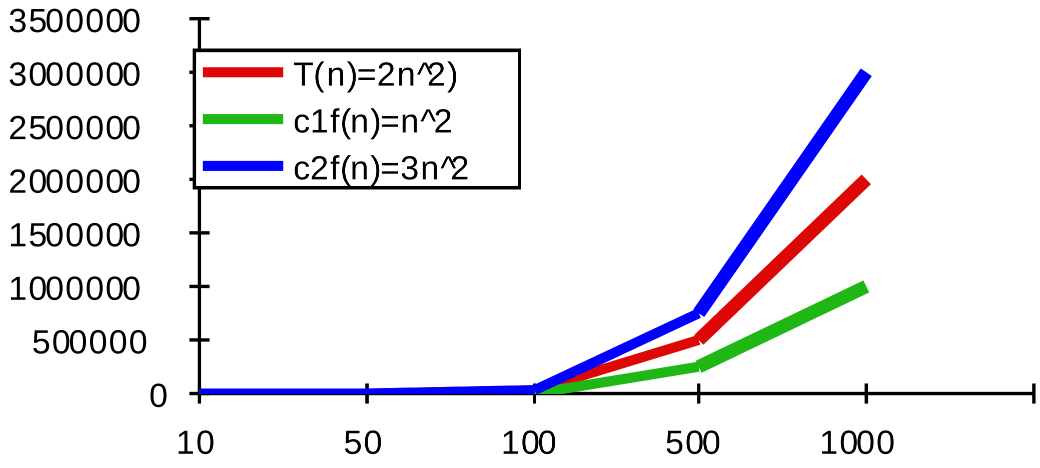
Medición del Tiempo de Ejecución de un Algoritmo

Casos de Complejidades- Promedio – Notación Θ

•Formalmente, decimos que **T(*n*) es Θ(*f*(*n*))** si existe un entero n_0 y las constantes $c1$ y $c2 > 0$ tal que para todos los enteros $n \geq n_0$, tenemos que:

$$0 \leq c1 f(n) \leq T(n) \leq c2 f(n)$$

•Ejemplo: $0 \leq n^2 \leq 2n^2 \leq 3n^2$



Criterios para el Cálculo de la Complejidad de un Algoritmo

1.- Instrucciones simples:

s_i; el T(n) es constante

•***Complejidad es constante O(1) = c***

2.- Instrucciones secuenciales:

Sean i_1; i_2; …. ; i_k con T_1; T_2; …. ; T_k respectivamente

El $T(n) = \Sigma T_i$

•***Complejidad es O(1)***

3.- Instrucciones secuenciales simples:

s_1; s_2; …. ; s_k

•***Suma de constantes es constante. Complejidad es O(1)***

4.- Ciclos simples:

•Se determina el nro de iteraciones

```
for(i=0;i<n;i++) { s; }
```

donde s is *O*(1)

•**Complejidad es** ***O(n) = 1*** **mejor caso**

O(n) = n **peor caso**

Criterios para el Cálculo de la Complejidad de un Algoritmo

5.- Ciclos anidados:

- Ciclos internos están afectados por los ciclos externos
- Se analizan de la parte mas interna hacia la parte externa
- Se determina cuantas veces iteran los ciclos internos y después como son afectados por los ciclos externos

(Nro iteraciones Internas * Nro Iteraciones externas)

6.- Ciclos con índices que varían linealmente

```
for(i=0;i<n;i++)
  for(j=0;j<n;j++) { s; }
```

•Complejidad es $O(n) = n^2$

7.- Ciclos con índice que no varían linealmente

```
h = 1;
while ( h <= n ) {
        s;
        h = 2 * h;
    }
```

- h *toma valores* 1, 2, 4, … *hasta que excede a n*
- *Hay* $c + \log_2 n$ iteraciones

- **Complejidad es** $O(\log n)$ **ó** $O(n) = \log n$

Criterios para el Cálculo de la Complejidad de un Algoritmo

8.- Regla de la suma

Suma de instrucciones con distintas complejidades

→ la complejidad total es determinada por el término de mayor grado

Ejemplo

$O(n) = n + n^2 + n^3 \quad => \quad O(n) = n^3$

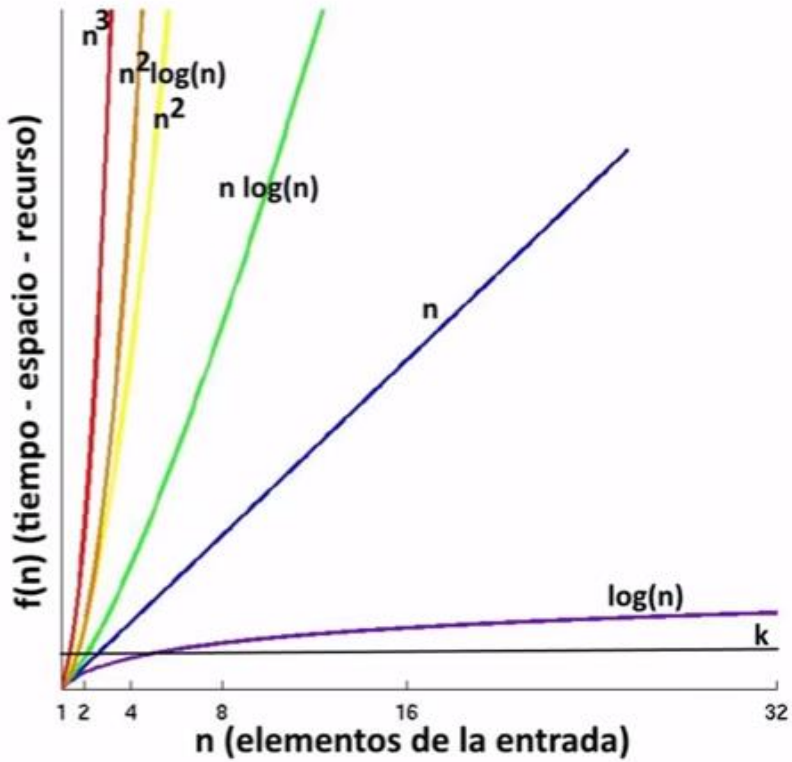
Complejidad de un Algoritmo

https://www.youtube.com/watch?v=__vX2sjlpXU

Ejemplos de Orden

Orden de ...

Big-O	Nombre
O(k)	Constante
O(log n)	Logarítmica
O(n)	Lineal
O(n log n)	Log-lineal
$O(n^2)$	Cuadrática
$O(n^3)$	Cúbica
$O(2^n)$	Exponencial
O(n!)	Factorial



Utilidad del cálculo del Orden

Crecimiento Exponencial o Factorial

n	$O(2^n)$	$O(n!)$
50	1,13E+15	3,04E+64
60	1,15E+18	8,32E+81
100	1,27E+30	9,33E+157

Imaginemos un microprocesador que puede realizar 1000 instrucciones por nanosegundo
En 1 segundo ejecuta 10^{12} instrucciones

n=50 - $O(2^n)$ -> 19 minutos

n=60 - $O(2^n)$ -> 19.215 minutos ~ 13 días

n=100 - $O(2^n)$ -> 40.196.936.841 años

Orden de Algoritmos

Muchas veces nos encontramos con casos como el siguiente:

```
For  (i= 1;  i <= n; i++)
     For ( j=1; j <= i; j++)
          sum = sum + (i*j);
```

Si observan la estructura del ciclo, aunque el ciclo externo se ejecuta n veces, el ciclo interno varía en cada iteración. En general el algoritmo se ejecuta 1 vez en la primera iteración, 2 veces en la segunda iteración y así continua.

Es decir el número de veces que se ejecuta es:

$1 + 2 + 3 + 4 + 5 + \ldots\ldots\ldots + n$

Orden de Algoritmos

Esta serie se resuelve con la siguiente formula:

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

Usando notación asintótica la serie se puede simplificar a n^2

Aunque intuitivamente la solución nos daba n^2 sin hacer estos cálculos, es importante al momento de calcular el orden, explicar de donde se obtiene el valor