

Introducción al análisis de algoritmos



Análisis y Diseño de Algoritmos

Ing. Román Martínez M.

Análisis de algoritmos



- Si se tuvieran 2 programas que hacen lo mismo, ¿cómo se podrían comparar?
 1. Eficiencia:
 - ✓ **Tiempo de ejecución**
 - ✓ Uso de espacios de memoria
 2. Facilidad de lectura, mantenimiento, rapidez para codificarlo.

Medición del tiempo de ejecución



El tiempo de ejecución depende de:

1. La entrada al programa:
 - ✓ Su tamaño
 - ✓ Sus características
2. La calidad del código generado para el programa por el compilador.
3. La rapidez de las instrucciones de máquina.
4. La **complejidad de tiempo del algoritmo.**

¿Cómo medir?



- Cantidad de instrucciones básicas (o elementales) que se ejecutan.
- Ejemplos de instrucciones básicas:
 - asignación de escalares
 - lectura o escritura de escalares
 - saltos (goto's) implícitos o explícitos.
 - evaluación de condiciones
 - llamada a funciones
 - etc.

Ejemplo



<i>cont = 1;</i>	--> 1
<i>while (cont <= n) do</i>	--> n+1
<i>x = x + a[cont];</i>	--> n
<i>x = x + b[cont];</i>	--> n
<i>cont = cont + 1;</i>	--> n
	--> n (goto implícito)

TOTAL: $5n + 2$

Ejemplo



```
for (int i=1; i<=n; i++){  
    x = x + a[i];  
    x = x + b[i];  
}
```

--> 1

--> $n+1$ (condiciones)

--> n (incrementos)

--> n

--> n

--> n (goto implícito)

TOTAL: $5n + 2$

Ejemplo



```
z = 0;  
for (x = 1; x <= n; x++)  
  for (y = 1; y <= n; y++)  
    z = z + a[x,y];
```

--> 1

--> $2n+2$

--> $(2n+2)*n$

--> $n*n$

--> $n*n$ (goto implícito)

--> n (goto implícito)

TOTAL: $4n^2 + 5n + 3$



**¿Qué es lo importante
aprender respecto al
análisis de algoritmos?**

1

2

3

¿Qué es lo importante aprender respecto al análisis de algoritmos?

1

La notación universal que identifica el comportamiento de un algoritmo en cuanto al tiempo de ejecución y el uso de memoria.

Orden de complejidad temporal o espacial
expresado como **$O(f(n))$**

Notación universal



- La **NOTACIÓN ASINTÓTICA** es la propuesta de notación aceptada por la comunidad científica para describir el comportamiento en eficiencia (o complejidad) de un algoritmo.
- Describe en forma sintética el comportamiento de la función que con la variable de entrada, determina el número de operaciones que realiza el algoritmo.

NOTACIÓN ASINTÓTICA



- **COMPLEJIDAD TEMPORAL (y ESPACIAL).**
Tiempo (o espacio) requerido por un algoritmo, expresado en base a una función que depende del tamaño del problema.
- **COMPLEJIDAD TEMPORAL ASINTÓTICA (y ESPACIAL).** Comportamiento límite conforme el tamaño del problema se incrementa. Determina el tamaño del problema que puede ser resuelto por un algoritmo.

NOTACIÓN ASINTÓTICA



Si un algoritmo de complejidad asintótica para un problema de tamaño **n** , procesa entradas en términos de la función **$f(n)$** , la **notación asintótica** del algoritmo, será expresada como **$O(g(n))$** , en donde:

$$f(n) \leq c g(n)$$

siendo **c** una constante
y entendiendo que **$f(n)$ es de orden $g(n)$** .

Definición

- Se dice que la función $f(n)$ "es de orden $g(n)$ " [**$O(g(n))$**], si existen constantes positivas c y n_0 tales que $f(n) \leq c g(n)$ cuando $n \geq n_0$
- Ejemplos:
 - $n+5$ es **$O(n)$** pues $n+5 \leq 2n$ para toda $n \geq 5$
 - $(n+1)^2$ es **$O(n^2)$** pues $(n+1)^2 \leq 4n^2$ para $n \geq 1$
 - $(n+1)^2$ **NO** es **$O(n)$** pues para cualquier **$c > 1$** no se cumple que $(n+1)^2 \leq c \cdot n$

Órdenes más comunes



- $O(1)$ Constante
- $O(n)$ Lineal
- $O(n^2)$ Cuadrático
- $O(n^3)$ Cúbico
- $O(n^m)$ Polinomial
- $O(\log(n))$ Logarítmico
- $O(n \log(n))$ $n \log(n)$
- $O(m^n)$ Exponencial
- $O(n!)$ Factorial

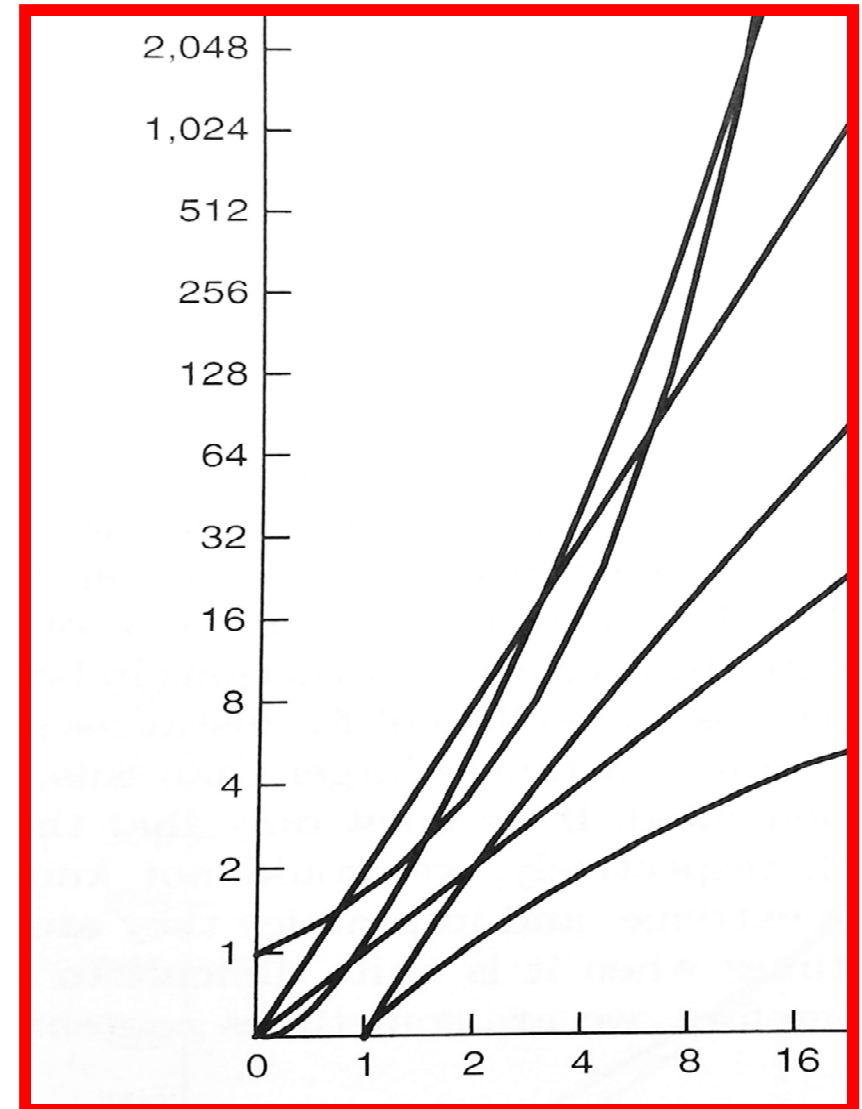
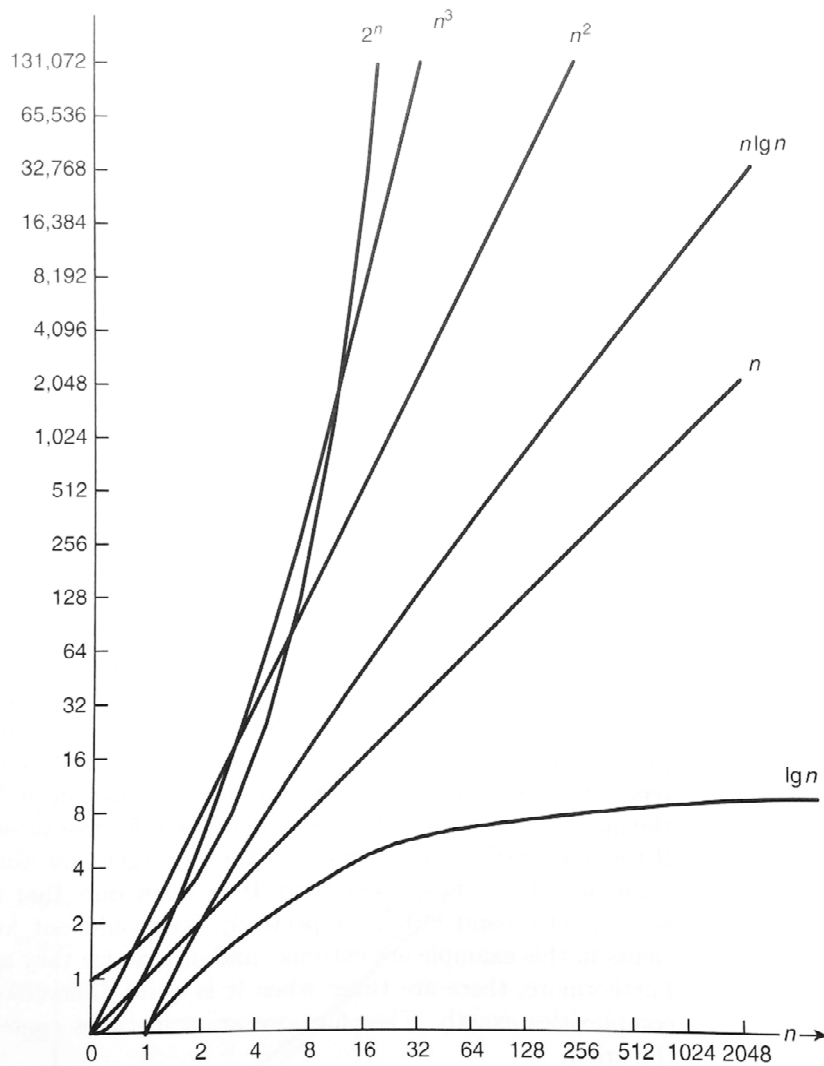
¿Qué es lo importante aprender respecto al análisis de algoritmos?

2

El impacto que tiene el orden de complejidad, independientemente de la velocidad de la computadora.

La computadora más potente puede tardar millones de años en resolver un problema...

Comportamiento de las funciones



Algunas dimensiones de problemas...



- **Genoma humano**

- *Secuenciar significa determinar el orden exacto de los pares de bases en un segmento de ADN. Los cromosomas humanos tienen entre 50,000,000 a 300,000,000 pares de bases.*

- **Búsquedas en Internet**

- *Google indexa 20 mil millones de páginas web diariamente. Maneja más de tres mil millones de búsquedas diarias. Ofrece almacenamiento gratuito de correo electrónico a los 425 millones de usuarios de Gmail. Propone resultados de búsqueda incluso antes de que el usuario haya terminado de escribir la frase.*



**Juguemos con el
simulador...**

SIMULADOR DE TIEMPOS DE EJECUCIÓN DE ALGORITMOS

Cantidad de datos de entrada (n): 100

Orden de complejidad de tiempo del algoritmo	Velocidad de la computadora (instrucciones por unidad de tiempo)						
	1	10	100	1000	10000	100000	1000000
$O(\log n)$	6.643856	0.664386	0.066439	0.006644	0.000664	6.64E-05	6.64E-06
$O(n)$	100	10	1	0.1	0.01	0.001	0.0001
$O(n \log n)$	664.3856	66.43856	6.643856	0.664386	0.066439	0.006644	0.000664
$O(n^2)$	10000	1000	100	10	1	0.1	0.01
$O(n^3)$	1000000	100000	10000	1000	100	10	1
$O(2^n)$	1.27E+30	1.27E+29	1.27E+28	1.27E+27	1.27E+26	1.27E+25	1.27E+24

CONTRASTA:

El algoritmo **más eficiente** con la computadora **más lenta**

VS.

El algoritmo **más ineficiente** con la computadora **más rápida**

Ejemplo del comportamiento



- Considerar que el algoritmo A1 es de orden **$O(n \log n)$** y que procesa 1,000 datos en 1 segundo.
- Su comportamiento al aumentar el tamaño de la entrada sería el siguiente:

$n = 2,000$	2.2 segundos
$n = 5,000$	6.2 segundos
$n = 10,000$	13.3 segundos
$n = 100,000$	2.77 minutos
$n = 1,000,000$	33.3 minutos
$n = 10,000,000$	6.48 horas

Ejemplo del comportamiento



- Considerar que el algoritmo A2 es de orden **$O(n^2)$** y que procesa 1,000 datos en 1 segundo.
- Su comportamiento al aumentar el tamaño de la entrada sería el siguiente:

$n = 2,000$	4 segundos
$n = 5,000$	25 segundos
$n = 10,000$	1.66 minutos
$n = 100,000$	2.77 horas
$n = 1,000,000$	11.5 días
$n = 10,000,000$	3.25 años

Ejemplo del comportamiento

- Si se contara con un algoritmo A3 de orden **exponencial** y que procesa 10 datos en 1 segundo...
- Su comportamiento al aumentar el tamaño de la entrada sería el siguiente:

n = 15

32 segundos

n = 20

17.1 minutos

n = 25

9.1 horas

n = 30

12.1 días

n = 35

1.09 años

n = 50

35,700 años

n = 100

4.02×10^{19} años !!!

¿Cómo afecta la velocidad de la computadora?

- Ejemplo: Suponer que se tienen los siguientes algoritmos con la complejidad de tiempo correspondiente:

A1 con $T(n) = 100n$

A2 con $T(n) = 5n^2$

A3 con $T(n) = n^3/2$

A4 con $T(n) = 2^n$

¿Cuál es el orden de cada algoritmo?

¿Cómo afecta la velocidad de la computadora?

- También suponer que se dispone de 1,000 segundos (aprox. 17 minutos) para ejecutar cada algoritmo...
- ¿Cuál es el tamaño máximo de la entrada que se puede procesar?

A1 con $T(n) = 100n$ **10**

A2 con $T(n) = 5n^2$ **14**

A3 con $T(n) = n^3/2$ **12**

A4 con $T(n) = 2^n$ **10**

¿Cómo afecta la velocidad de la computadora?



- Suponer ahora que se cuenta con una computadora 10 veces más rápida... por lo tanto, es posible dedicar 10,000 segundos a la solución de problemas que antes se les dedicaba 1,000 segundos...
- ¿Cuál es el tamaño máximo de la entrada que se puede procesar?

A1 con $T(n) = 100n$ *100*

A2 con $T(n) = 5n^2$ *45*

A3 con $T(n) = n^3/2$ *27*

A4 con $T(n) = 2^n$ *13*

¿Cómo afecta la velocidad de la computadora?

- ¿Cuál fue el aumento en la capacidad de procesamiento de cada algoritmo con el incremento de 10 veces en la velocidad de la computadora?
- ¿Cuál es el tamaño máximo de la entrada que se puede procesar?

A1 con $T(n) = 100n$	<i>10 --> 100</i>	<i>10</i>
A2 con $T(n) = 5n^2$	<i>14 --> 45</i>	<i>3.2</i>
A3 con $T(n) = n^3/2$	<i>12 --> 27</i>	<i>2.3</i>
A4 con $T(n) = 2^n$	<i>10 --> 13</i>	<i>1.3</i>

Ejemplo:

¿Influye la computadora?

- Para una Cray-1 utilizando Fortran, un algoritmo se procesa en $3n^3$ nanosegundos...
 - Para $n = 10$, se tardaría 3 microsegundos...
 - Para $n = 100$, se tardaría 3 milisegundos...
 - Para $n = 1000$, se tardaría 3 segundos...
 - Para $n = 2500$, se tardaría 50 segundos...
 - Para $n = 10000$, se tardaría 49 minutos...
 - Para $n = 1000000$, se tardaría 95 años!!

Ejemplo:

¿Influye la computadora?

- Para una TRS-80 (computadora personal Tandy de los 80's) utilizando Basic, un algoritmo se procesa en $19,500,000n$ nanosegundos...
 - Para $n = 10$, se tardaría .2 segundos...
 - Para $n = 100$, se tardaría 2 segundos...
 - Para $n = 1000$, se tardaría 20 segundos...
 - Para $n = 2500$, se tardaría 50 segundos...
 - Para $n = 10000$, se tardaría 3.2 minutos...
 - Para $n = 1000000$, se tardaría 5.4 horas...

Cita



- *“A medida de que los computadores aumenten su rapidez y disminuyan su precio, como con toda seguridad seguirá sucediendo, también el deseo de resolver problemas más grandes y complejos seguirá creciendo. Así la importancia del descubrimiento y el empleo de algoritmos eficientes irá en aumento, en lugar de disminuir...”*

Aho, Hopcroft, Ullman, 1983

¿Qué es lo importante aprender respecto al análisis de algoritmos?

3

Dado un algoritmo, identificar su orden de complejidad para evaluar su eficiencia.

Se tienen que contar las veces que se ejecutan las instrucciones... pero con estrategia...

- Mejor caso
- Peor caso
- Caso promedio

¿Cómo analizar un algoritmo?



- El conteo de operaciones puede ser:
 - **Igual (constante)** para todas las entradas posibles.
 - ✓ *Ejemplos: Sumar los elementos de un arreglo, Multiplicar 2 matrices, etc.*
 - **Variable** según el contexto de la entrada.
 - ✓ *Ejemplos: Búsqueda secuencial, binaria, etc.*
 - ✓ El análisis se apoya en encontrar la complejidad del algoritmo para el **peor caso** y el caso promedio en algunos algoritmos en los que sea posible.

Ejemplo: Encontrar la sumatoria de elementos PARES (o IMPARES) en un rango

¿Cuál de los 2 algoritmos es más eficiente?

VERSIÓN 1

```
sum = 0;
for (i = inf; i<=sup; i++)
    if (i%2 == 0)
        sum += i;
```

-> 1
-> n+2
-> n
-> n/2
-> 2n (goto e incremento)

Total = $4.5n + 3$
 $O(n)$

VERSIÓN 2

```
sum = 0;
if (inf%2 != 0) inf++;
for (i = inf; i<=sup; i+=2)
    sum += i;
```

-> 1
-> 1
-> n/2+2
-> n/2
-> 2n/2 (goto e incremento)

Total = $2n + 4$
 $O(n)$

Ejemplo: Sumar los datos de un arreglo



```
suma = 0;  
for (i=1; i<=n; i++)  
    suma = suma + arreglo[i];
```

¿Cómo obtener la complejidad de tiempo del algoritmo?

- Operación básica: `suma = suma + arreglo[i];`
- Se realiza n veces
- Algoritmo de orden lineal: **$O(n)$**

Ejemplo: Sort por intercambio



- Toma la primera posición del arreglo, y compara su contenido contra el resto de los valores del arreglo. Cada vez que se encuentra un elemento menor al de la posición, lo intercambia. Esto asegura que el dato que queda en la primera posición está ordenado.
- El proceso se repite con la segunda posición, la tercera y así sucesivamente hasta la penúltima posición, siempre comparando solamente contra los elementos desordenados.

Ejemplo: Sort por intercambio



- 8 3 5 9 2 7 4
- 3 8 5 9 2 7 4
- 2 8 5 9 3 7 4
- 2 8 5 9 3 7 4
- 2 5 8 9 3 7 4
- 2 3 8 9 5 7 4
- 2 3 8 9 5 7 4
- 2 3 5 9 8 7 4
- 2 3 4 9 8 7 5

- 2 3 4 9 8 7 5
- 2 3 4 8 9 7 5
- 2 3 4 7 9 8 5
- 2 3 4 5 9 8 7
- 2 3 4 5 9 8 7
- 2 3 4 5 8 9 7
- 2 3 4 5 7 9 8
- 2 3 4 5 7 9 8
- 2 3 4 5 7 8 9

Ejemplo: Sort por intercambio



```
for (i=1; i<n-1; i++)  
    for (j=i+1; j<=n; j++)  
        if (arreglo[j]<arreglo[i])  
            intercambia(arreglo[i], arreglo[j]);
```

- Operación básica: `arreglo[j]<arreglo[i]`
- Se realiza $(n-1)+(n-2)+\dots+1 = (n-1)*n/2$ veces
- Algoritmo de orden cuadrático : **$O(n^2)$**

Ejemplo: Multiplicación de matrices

$$\begin{pmatrix} a_{11} & a_{12} & \dots & a_{1n} \\ a_{21} & a_{22} & \dots & a_{2n} \\ \dots & \dots & \dots & \dots \\ a_{m1} & a_{m2} & \dots & a_{mn} \end{pmatrix} \times \begin{pmatrix} b_{11} & b_{12} & \dots & b_{1m} \\ b_{21} & b_{22} & \dots & b_{2m} \\ \dots & \dots & \dots & \dots \\ b_{n1} & b_{n2} & \dots & b_{nm} \end{pmatrix} =$$

$$\begin{pmatrix} c_{11} & c_{12} & \dots & c_{1m} \\ c_{21} & c_{22} & \dots & c_{2m} \\ \dots & \dots & \dots & \dots \\ c_{m1} & c_{m2} & \dots & c_{mm} \end{pmatrix}$$

$$c_{11} = a_{11} * b_{11} + a_{12} * b_{21} + \dots + a_{1n} * b_{n1}$$

$$c_{12} = a_{11} * b_{12} + a_{12} * b_{22} + \dots + a_{1n} * b_{n2}$$

...

$$c_{21} = a_{21} * b_{11} + a_{22} * b_{21} + \dots + a_{2n} * b_{n1}$$

...

$$c_{mm} = a_{m1} * b_{1m} + a_{m2} * b_{2m} + \dots + a_{mn} * b_{nm}$$

$$c_{ij} = \sum_{k=1}^n a_{ik} b_{kj}$$

Ejemplo: Multiplicación de matrices



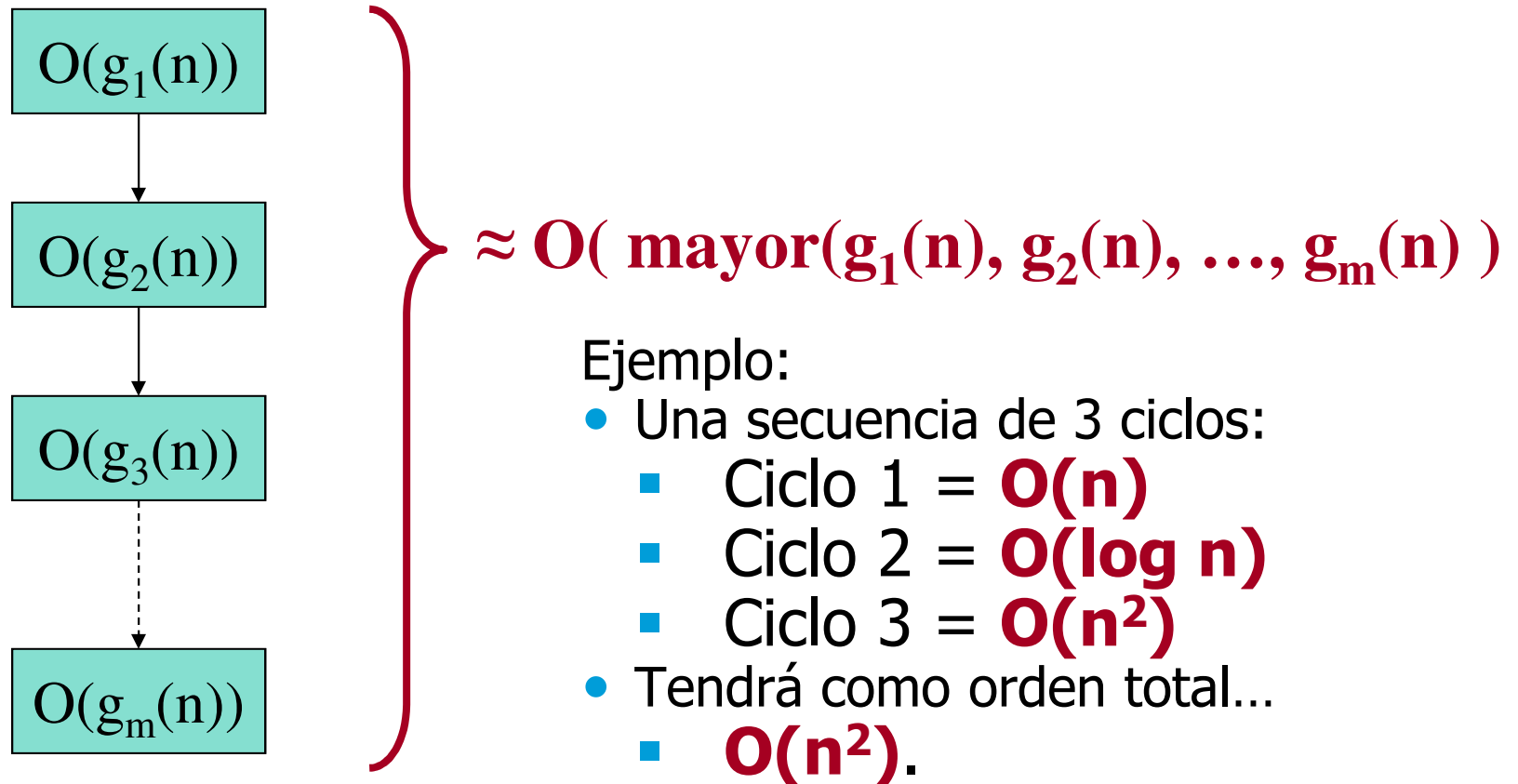
```
for (i=1; i<=n; i++)
  for (j=1; j<=n; j++)
  {
    C[i,j] = 0;
    for (k=1; k<=n; k++)
      C[i,j] = C[i,j]+A[i,k]*B[k,j];
  }
```

- Operación básica: $C[i,j] = C[i,j] + A[i,k] * B[k,j]$;
- Se realiza $n*n*n$ veces
- Algoritmo de orden cúbico: **$O(n^3)$**

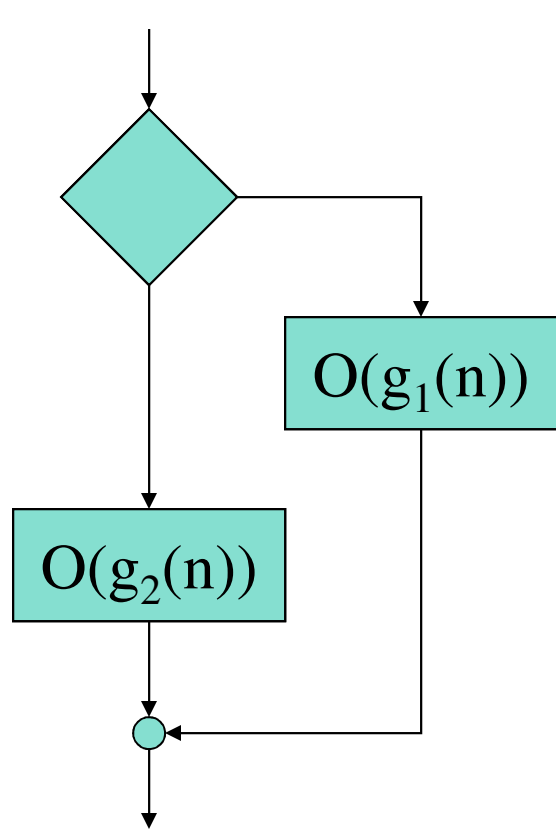


Reglas generales para el cálculo de la complejidad

Regla 1: Secuencia de instrucciones



Regla 2: Decisiones

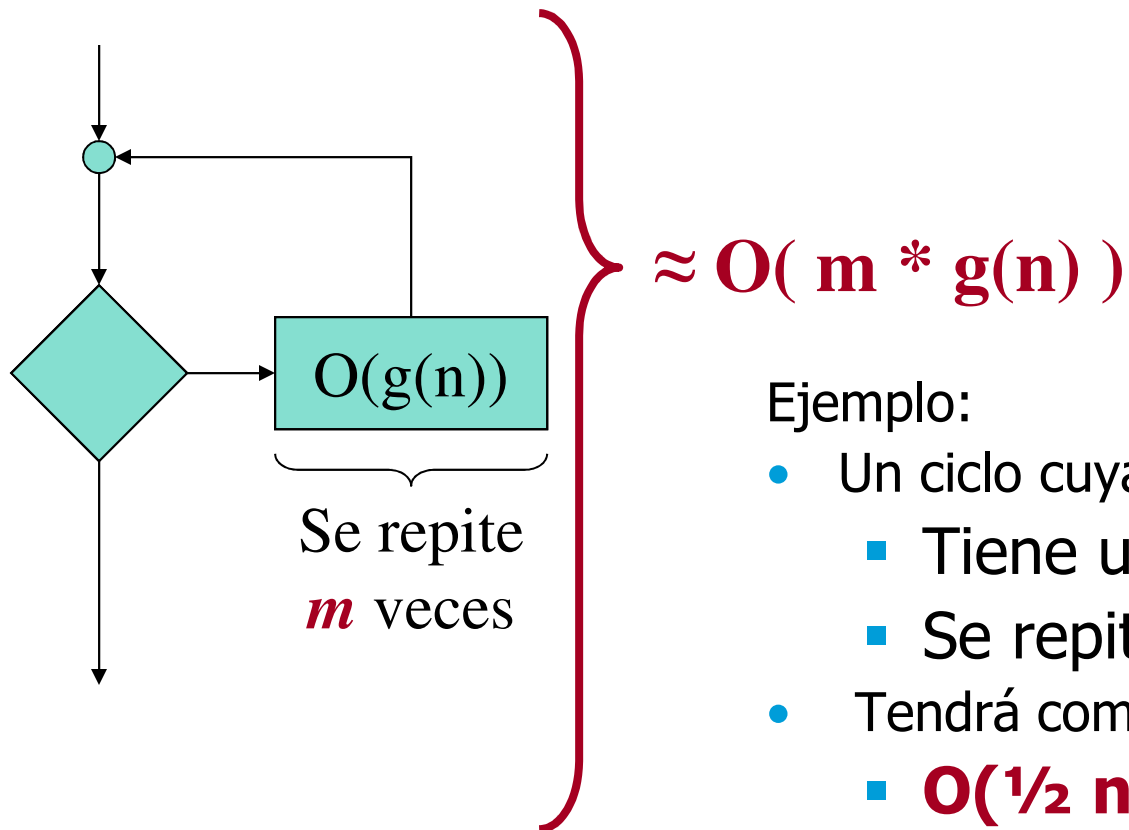


$\approx O(\text{mayor}(g_1(n), g_2(n)))$

Ejemplo:

- Una decisión con:
 - Rama then = **$O(n \log n)$**
 - Rama else = **$O(\log n)$**
- Tendrá como orden total...
 - **$O(n \log n)$** .

Regla 3: Ciclos



Ejemplo:

- Un ciclo cuya instrucción:
 - Tiene un **$O(\log n)$**
 - Se repite **$n/2$** veces
- Tendrá como orden total...
 - **$O(1/2 n \log n) = O(n \log n)$** .

Complejidad más común en los ciclos



- Si la variable de control se incrementa o decrementa con un valor constante: **Orden LINEAL**.
- Si la variable de control se multiplica o divide por un valor constante: **Orden LOGARÍTMICO**.
- En decisiones y ciclos anidados:
 - ✓ Analizar el código desde la instrucción más interna hacia el más externa.
 - ✓ **Orden CUADRÁTICO, CÚBICO o POLINOMIAL**, dependiendo de la cantidad de ciclos lineales que estén anidados.


Regla 4: Recursividad



- La complejidad de tiempo se obtiene contando la cantidad de veces que se hace la llamada recursiva.
- Casos de complejidad más comunes:
 - **Orden LINEAL** si sólo se tiene una llamada recursiva, con incrementos o decrementos en el parámetro de control.
 - **Orden LOGARÍTMICO** si sólo se tiene una llamada recursiva, con multiplicaciones o divisiones en el parámetro de control.
 - Si hay más de una llamada recursiva, el orden puede tender a ser **EXPONENCIAL**.



**¿Cómo influye el DISEÑO
de un algoritmo en su
complejidad?**

- 
- Es importante aprender a **analizar** algoritmos...
 - Pero es más importante conocer técnicas para **diseñar** algoritmos eficientes...
 - Y así, ante un problema, tener capacidad de decidir y aplicar el mejor algoritmo de solución...

EJEMPLO



- Encontrar el n-ésimo elemento de la **serie de Fibonacci....**
- Algoritmos de solución:
 - Iterativo (técnica de la **programación dinámica**).
 - Recursivo (técnica de "**divide y vencerás**").
- Complejidad de los algoritmos (análisis):
 - Iterativo: **$O(n)$**
 - Recursivo: **$O(2^{n/2})$**

Caso Fibonacci...



En una máquina que procese una operación básica en 1 nanosegundo...

- El algoritmo Iterativo, tardaría **81 nanosegundos** para encontrar el **80**avo. elemento de la serie...
- El algoritmo Recursivo, tardaría **18 minutos** para encontrar el **80**avo. elemento de la serie...
- El algoritmo Iterativo, tardaría **121 nanosegundos** para encontrar el **120**avo. elemento de la serie...
- El algoritmo Recursivo, tardaría **36 años** para encontrar el **120**avo. elemento de la serie...

Algoritmo Fibonacci



```
#include <iostream.h>
```

```
long int fibo1 (int n)
{ long int ant = 1, act = 1, aux;
  for (int i=3; i <= n; i++)
  { aux = ant + act;
    ant = act;
    act = aux; }
  return act;
}
```

Iterativo

```
long int fibo2 (int n)
{ if (n <3)
  return 1;
  else
  return (fibo2(n-1) + fibo2(n-2));
}
```

Recursivo

```
main()
{ int n;
  do
  { cout << "Dame el valor de n: ";
    cin >> n;
    cout << "El n-esimo numero de la serie de Fibonacci es: " << fibo1(n) << endl;
  } while (n>1);
}
```

Ejemplo: Fibonacci (Iterativo)

<i>ant = 1;</i>	--> 1
<i>act = 1;</i>	--> 1
<i>while (n>2)</i>	--> n-2 + 1
<i>aux = ant + act;</i>	--> n-2
<i>ant = act;</i>	--> n-2
<i>act = aux;</i>	--> n-2
<i>n = n - 1;</i>	--> n-2
<i>write (act);</i>	--> n-2 + 1

$$T(n) = 6n-8$$

*Por lo tanto el orden
del algoritmo es*

$$O(n)$$

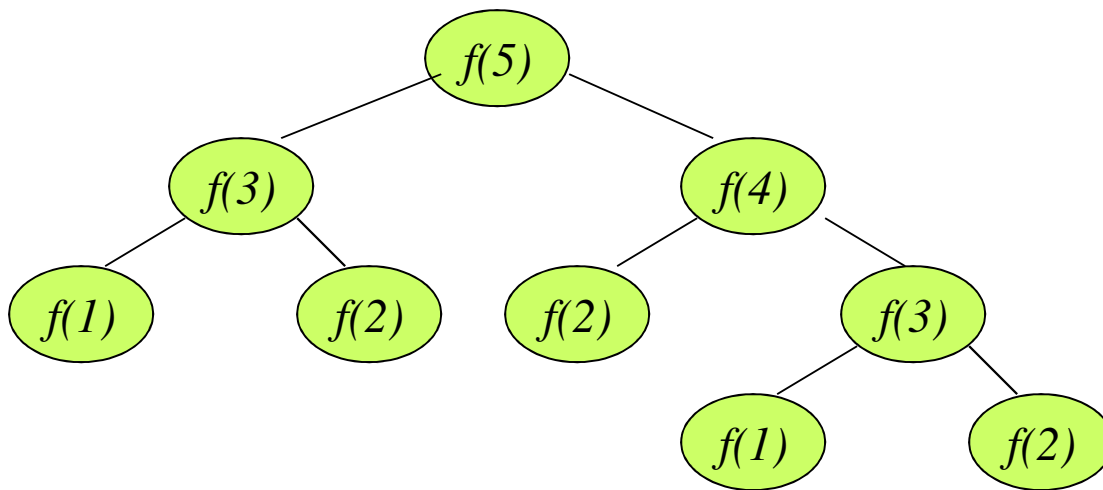
Ejemplo: Fibonacci (recursivo)



```
Function fibonacci (n:int): int;  
    if (n < 3) return 1;  
    else  
        return fibonacci(n-1)+fibonacci(n-2);
```

- ¿Cómo obtener la complejidad de tiempo del algoritmo?
- Operación básica: *Cálculo de cada término de la serie.*
- Algoritmo de orden: **$O(2^{n/2})$**

Análisis de Fibonacci (recursivo)



Relación:

El término $T(n)$ requiere
 $T(n-1)+T(n-2)+1$ términos
para calcularse.

¿Cuántos términos
se requieren para
calcular:

$f(5)? \rightarrow 9$

$f(4)? \rightarrow 5$

$f(3)? \rightarrow 3$

$f(2)? \rightarrow 1$

$f(6)? \rightarrow 15$

Análisis de Fibonacci

- Si el término **$T(n)$** requiere **$T(n-1)+T(n-2)+1$** términos para calcularse...
- se puede decir que **$T(n) > 2 * T(n-2)$** ...
- y por lo tanto: **$T(n) > 2 * 2 * T(n-4)$** ...
- y **$T(n) > 2 * 2 * 2 * T(n-6)$** ...
- y así sucesivamente hasta:

$$T(n) > \underbrace{2 * 2 * 2 * \dots * 2}_{n/2 \text{ veces}} * T(1)$$

Por lo tanto:
 $T(n) > 2^{n/2}$
y podemos decir
que el orden del
algoritmo es
 $O(2^{n/2})$

Ejemplo: Búsqueda secuencial

```
pos = 1;  
while(pos<=n) and (arreglo[pos] < dato) do  
    pos = pos + 1;  
if (pos > n) or (arreglo[pos]<>dato) then  
    pos = 0;
```

- Mejor caso: **1**
- Peor caso: ***n***
- Caso promedio: depende de probabilidades
 $3n/4 + 1/4$

Por lo tanto:

$O(n)$

Ejemplo: Búsqueda binaria

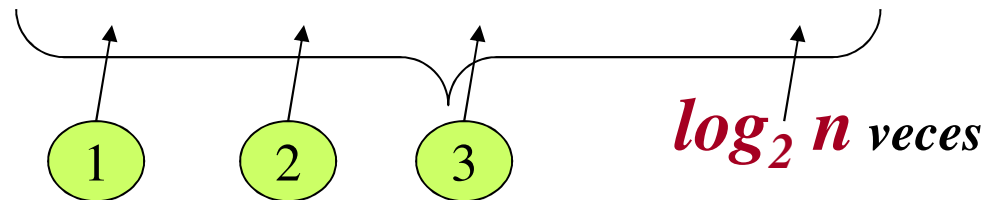
```
inicio =1; fin = n; pos = 0;
while (inicio<=fin) and (pos == 0) do
    mitad = (inicio+fin) div 2;
    if (x == arreglo[mitad]) then pos = mitad;
    else if (x < arreglo[mitad]) then fin = mitad-1
        else inicio = mitad+1;
```

- Operación Básica: $x == \text{arreglo}[\text{mitad}]$
- Mejor caso: **1**

Ejemplo: Búsqueda binaria

- Peor caso: No encontrar el dato
- Suponiendo que n es potencia de 2:

$$n/2 + n/4 + n/8 + \dots + n/n$$



- Caso Promedio: Un análisis detallado lleva a encontrar la complejidad de: $\lfloor \log_2 n \rfloor \pm 1/2$
- Por lo tanto, el orden del algoritmo es: **$O(\log n)$**