

Análisis de Algoritmos

Ingeniería Civil Informática
Departamento de Computación e Industrias
Facultad de Ciencias de la Ingeniería

Mg. Hugo Araya Carrasco



Análisis de Algoritmos

- Es una herramienta para hacer la evaluación de un diseño.
- Permite establecer la calidad de un programa y compararlo con otros programas que resuelven un mismo problema.
- Suponga que existen dos programas P1 y P2 para resolver el mismo problema.

¿Cuál de los dos es mejor?.

Análisis de Algoritmos

Respuesta inicial:

Implementar ambos programas y medir el tiempo que cada uno de ellos consume para resolver el problema.

Modificar los datos de entrada, promediar al final su desempeño para establecer su comportamiento en el caso promedio.

Análisis de Algoritmos

Problemas:

- Pueden existir muchos algoritmos para resolver el problema.
- Costoso y casi imposible implementar todos los programas.
- Modificación de los datos puede ser una labor sin sentido.

Recursos a considerar en Informática

Hay dos **recursos** que son fundamentales en informática, los cuales debemos tener en consideración cuando desarrollamos algoritmos.

Tiempo : Cuanto demora mi algoritmo (velocidad).

Espacio: Cuanto consume mi algoritmo (en términos de memoria y uso de almacenamiento)

Además, en la actualidad también se esta midiendo el uso de energía (aplicaciones amigables con el medio ambiente)

Análisis de Algoritmos

Objetivo:

“Establecer una medida de la calidad de los algoritmos, que permita compararlos sin la necesidad de implementarlos”

Esto implica asociar a cada algoritmo una función matemática que mida su eficiencia (en términos de ...).

Tiempo de ejecución de un algoritmos

Factores que tienen influencia en dicho valor.

- Velocidad de procesamiento.
- El compilador utilizado (calidad del código generado).
- La estructura del algoritmo.

¿Cuáles de estos factores no son parte de la solución?.

Tiempo de ejecución

Además de la estructura del algoritmo, se debe tener en cuenta que el número de datos con los cuales trabaja un programa influye en su tiempo de ejecución.

Un programa de ordenamiento de un arreglo, se demora menos en ordenar 100 elementos que 500.

“El tiempo de ejecución de un algoritmo debe medirse en función del tamaño de los datos de entrada que debe procesar”.

Tiempo de ejecución de un algoritmos

$T_A(n)$ Se define como el tiempo empleado por el algoritmo A en procesar una entrada de tamaño n y producir una solución al problema.

Nota: El tiempo no lo medimos en unidades de tiempo estándares. Nosotros mediremos el tiempo en función de la cantidad de instrucciones que deben ejecutarse para alcanzar el objetivo del problema.

El ideal es encontrar una función matemática que describiera de manera exacta $T_A(n)$.

Sin embargo, en muchos casos, el cálculo de esta función no se puede realizar, ya que depende de otro factor no considerado y que es casi imposible de medir: *la calidad de la entrada*.

Tiempo de ejecución

Ejemplo: Consideremos el algoritmo utilizado para determinar si un elemento se encuentra en un arreglo unidimensional (vector o lista) de n posiciones.

```
// elem: es el elemento buscado  
i=0;  
while ((i<N) and (vec[i] != elem)):  
    i+=1
```

Tiempo de ejecución de un algoritmos

Análisis puramente teórico.

- Verificar la influencia que tienen los datos específicos de la entrada (no solamente su cantidad).
- Supongamos que fijamos $N = 6$ y que la evaluación de cada línea del programa toma t microsegundos.
- Consideremos el vector siguiente:

0	1	2	3	4	5
5	6	7	8	9	10

Tiempo de ejecución

Si elem = 5, cuantos microsegundos consume

<code>i = 0;</code>	<i>t ms.</i>
<code>((0<6) and (vec[0] != 5))</code>	<i>t ms.</i>
<code>i+=1</code>	----
<i>Total</i>	<i>2 t ms.</i>

Tiempo de ejecución de un algoritmos

Si elem = 9,

¿Cuántos microsegundos consume el algoritmo?

<code>i = 0;</code>	<i>t ms.</i>
<code>((0<6) and (vec[0] != 9))</code>	<i>t ms.</i>
<code>i+=1</code>	<i>t ms.</i>
<code>((1<6) and (vec[1] != 9))</code>	<i>t ms.</i>
<code>i+=1</code>	<i>t ms.</i>
<code>((2<6) and (vec[2] != 9))</code>	<i>t ms.</i>
<code>i+=1</code>	<i>t ms.</i>
<code>((3<6) and (vec[3] != 9))</code>	<i>t ms.</i>
<code>i+=1</code>	<i>t ms.</i>
<code>((4<6) and (vec[4] != 9))</code>	<i>t ms.</i>
<i>Total</i>	<i>10 t ms.</i>

Tiempo de ejecución

Aunque se conozca el tamaño de los datos de entrada, es imposible para muchos problemas determinar el tiempo de ejecución para cada una de las posibles entradas.

Por esta razón se debe trabajar con el tiempo utilizado por el algoritmo en el peor de los casos ya que es mucho más fácil definir este peor caso.

Con este antecedente se redefine $T_A(n)$.

$T_A(n)$ = Tiempo que se demora el algoritmo A en el peor de los casos, para encontrar una solución a un problema de tamaño n .

Tiempo de ejecución de un algoritmos

Para el ejemplo anterior:

¿cual es el peor caso?

Complejidad

La idea detrás del concepto de complejidad es tratar de encontrar una función $f(n)$, fácil de calcular y conocida, que acote el crecimiento de la función de tiempo, para poder decir:

“ $T_A(n)$ crece aproximadamente como f ”

○

“En ningún caso $T_A(n)$ se comportará peor que f al aumentar el tamaño del problema”.

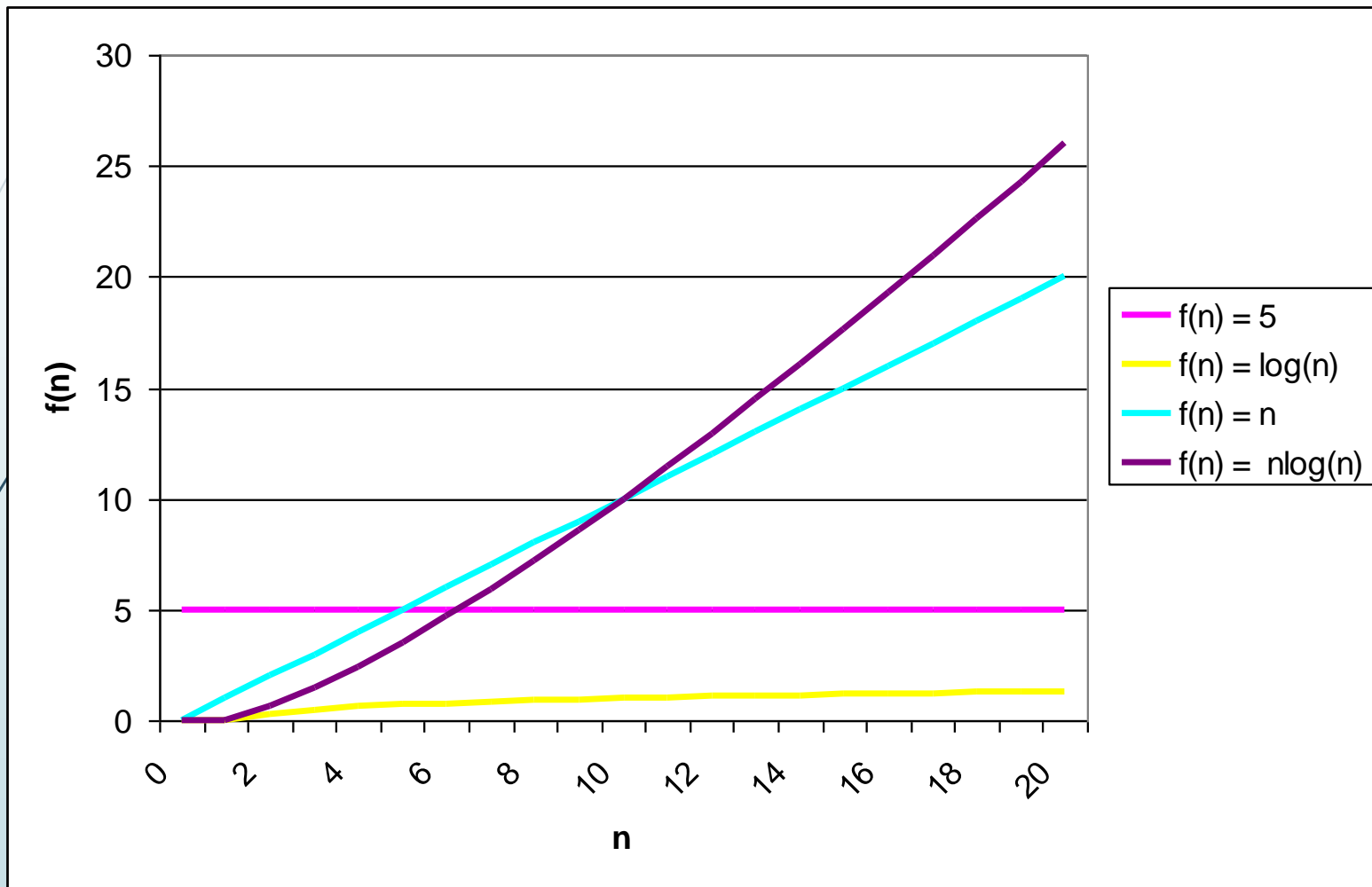
Complejidad

A modo de ejemplo se pueden mencionar algunas funciones típicas de complejidad de algoritmos (dicho de otra forma que acotan superiormente el comportamiento del tiempo de ejecución).

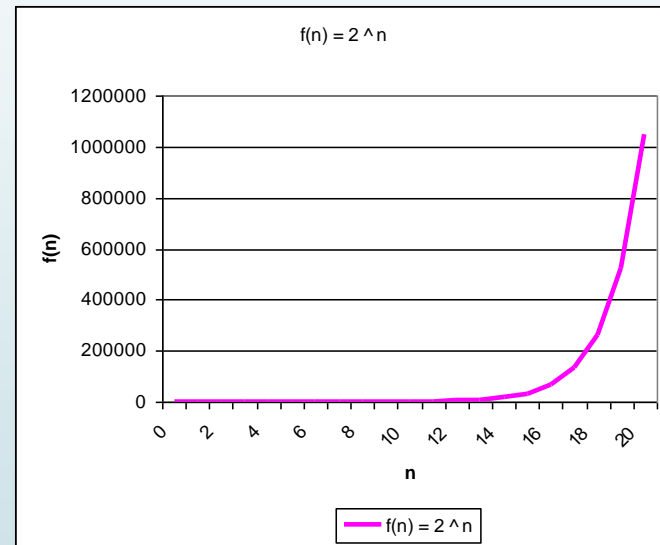
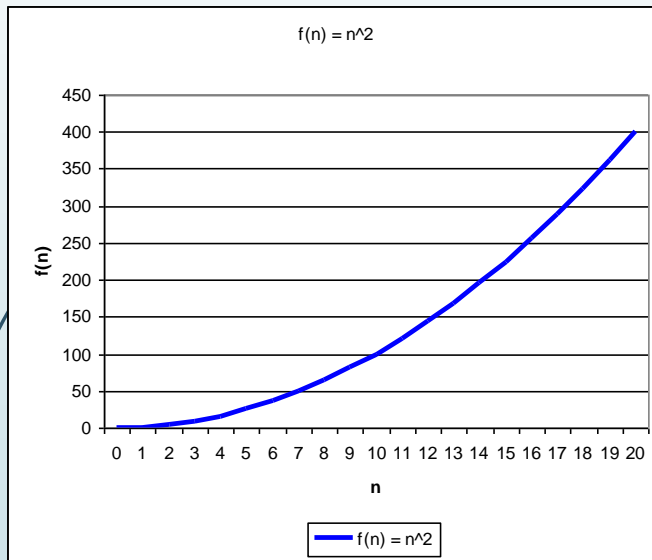
2^n , n^3 , n^2 , $n\log(n)$, $\log(n)$, n

¿Cuál es la gráfica de cada una de ellas?

Complejidad



Complejidad



Complejidad

Un problema se denomina **Tratable** si existe un algoritmo de complejidad polinomial para resolverlo. En caso contrario se denomina **Intratable**.

Esta clasificación es importante porque, cuando el tamaño del problema aumenta, los algoritmos de complejidad polinomial dejan de ser utilizables de manera gradual.

Complejidad

Los algoritmos para resolver problemas intratables, explotan de un momento a otro, volviéndose completamente incapaces para llegar a una respuesta al problema planteado.

El caso limite de los problemas **Intratables** son los problemas **Indecibles**, son problemas para los cuales no existen algoritmos que los resuelvan.

Complejidad	<i>20</i>	<i>50</i>	<i>100</i>	<i>200</i>	<i>500</i>	<i>1000</i>
$1000n$	0.02 s.	0.05 s.	0.1 s.	0.2 s.	0.5 s.	1 s.
$1000n \cdot \log(n)$	0.09 s.	0.3 s.	0.6 s.	1.5 s.	4.5 s.	10 s.
$100n^2$	0.04 s.	0.25 s.	1 s.	4 s.	25 s.	2 m.
$10n^3$	0.02 s.	1 s.	10 s.	1 m.	21 m.	2.7 h.
$n^{\log(n)}$	0.4 s.	1.1 h.	220 días	125 Siglos	XXX	XXX
$2^{n/3}$	0.001 s.	0.1 s.	2.7 h.	$3 \cdot 10^4$ Siglos	XXX	XXX
2^n	1 s.	35 Años	$3 \cdot 10^4$ Siglos	XXX	XXX	XXX
3^n	58 m.	$2 \cdot 10^9$ siglos		XXX	XXX	XXX

Notación Asintótica

Operación elemental: es aquella operación cuyo tiempo de ejecución se puede acotar superiormente por una constante que solamente dependerá de la implementación particular usada: de la maquina, del lenguaje, del compilador, etc.

Ejemplo de estas operaciones son: suma, resta, multiplicación, asignación, acceso a arreglos, etc. Aunque en rigor el tiempo de una multiplicación no es el mismo que el tiempo de la suma, pero difieren en una constante multiplicativa.

Notación Asintótica

La eficiencia de un algoritmo se mide mediante las operaciones elementales, más específicamente del número de operaciones elementales que se deben ejecutar.

Análisis del Peor Caso: se define como el máximo costo (operaciones elementales) de aplicar el algoritmo a un problema de tamaño n .

Este análisis se suele aplicar para casos extremos en los que interesa saber cuanto, como máximo, va a costar la aplicación del algoritmo.

Notación Asintótica

- Algunas reglas básicas para realizar dicho conteo:
 - Operaciones básicas (+, -, *, =, ...): Una unidad de tiempo, o alguna constante.
 - Operaciones de entrada / salida: Otra unidad de tiempo, o una constante diferente.
 - Ciclos Para (for): Se pueden expresar como la sumatoria, con los límites del ciclo.
 - Si y Case: Estudiar lo que puede ocurrir. Mejor caso y peor caso según la condición.
 - Llamadas a procedimientos: Una constante de la llamada más el tiempo del procedimiento.

Notación Asintótica

La **notación asintótica** permite realizar simplificaciones aun cuando estemos interesados en medir algo más tangible que el tiempo de ejecución, tal como es el número de veces que se ejecuta una instrucción dentro del programa.

La notación asintótica trata acerca del comportamiento de funciones en el límite, es decir, para valores grandes de su parámetro. Esto hace que los valores pequeños de las entradas no sean interesantes.

Dicho de otra manera, estamos interesados en las tasas de crecimientos en lugar de los valores concretos.

Notación Asintótica

La notación O (o grande) o cota superior es la encargada de dar una cota para el peor caso y determinar las acotaciones superiores lo más exactamente posible para valores crecientes de la entrada.

Por lo tanto se puede asegurar que conociendo la cota superior, ningún tiempo empleado en resolver el problema dado será de un orden superior al de la cota. Se conoce como el orden del peor caso.

La notación asintótica clasifica las funciones de tiempo de los algoritmos para que puedan ser comparadas.

Notación Asintótica

Definición.

Sean $f(n)$ y $g(n)$ dos funciones arbitrarias tales que:

$f(n), g(n): \mathbb{Z}^+ \rightarrow \mathbb{Z}^+$, se dice que $f(n)$ es “*O grande*” de $g(n)$ y se escribe:

$f(n) = O(g(n))$, si existen constantes positivas c y n_0 tales que

$$f(n) \leq cg(n) \quad , \quad \forall n \geq n_0 .$$

Decir que $f(n) = O(g(n))$ supone que $cg(n)$ es una cota superior del tiempo de ejecución del algoritmo.

Notación Asintótica

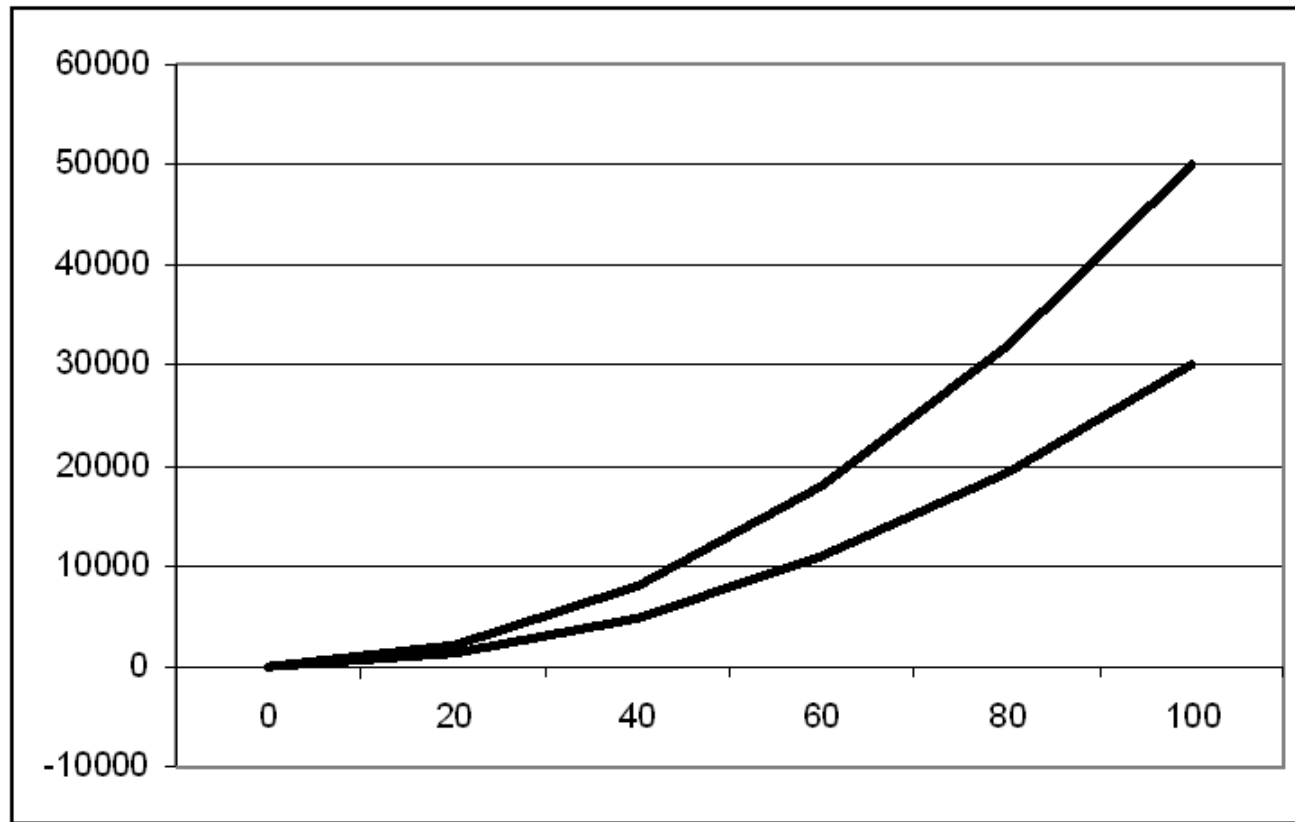
Ejemplo: Comprobar que $T(n) = 3n^2 + 2n - 5$ es de orden de $O(n^2)$.

Para ello debemos probar que $3n^2 + 2n - 5 \leq cn^2$.

Solución: $3n^2 + 2n - 5 \leq 5n^2$, entonces diremos que $T(n)$ es de $O(n^2)$.

Si graficamos ambas curvas podemos ver que efectivamente $5n^2$ acota superiormente a la función $3n^2 + 2n - 5$.

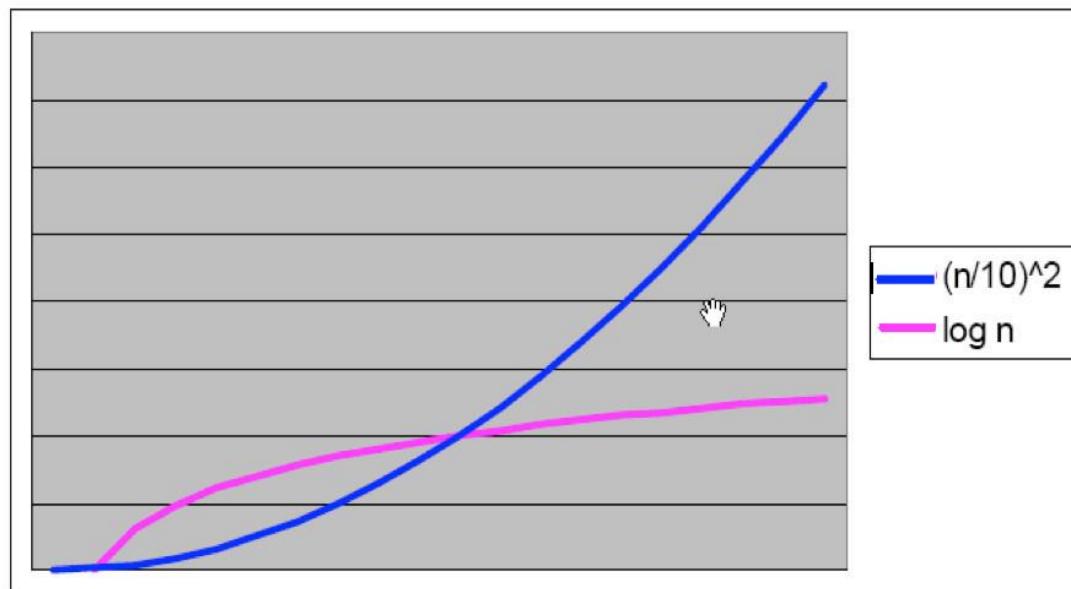
Notación Asintótica



Notación Asintótica

Esto se debe a que cuando n crece, el término que contiene la mayor potencia de n domina a los otros términos en ecuaciones como la ya mencionadas.

Consideremos las funciones $f(n) = \log(n)$ \wedge $g(n) = (n/4)^2$



Que puede decir de las funciones.

Notación Asintótica

La igualdad $f(n) = O(g(n))$ funciona solo en una dirección.

Dado que el termino $O(g(n))$ realmente especifica un conjunto infinito de funciones, esta sentencia dice que $f(n)$ pertenece al conjunto de funciones que pueden ser acotadas superiormente por un múltiplo fijo de $g(n)$, cuando n es suficientemente grande.

La notación $O(g(n)) = f(n)$ no debe ser usada nunca.

Como estamos tratando con cotas superiores asintóticas, si $f(n) = O(g(n))$ y $h(n)$ es una función cuyos valores son mayores que los de $g(n)$, cuando n es suficientemente grande, entonces $f(n) = O(h(n))$.

Notación Asintótica

Resumen

Cuando se dice que $f(n) = O(g(n))$, se está garantizando que la función $f(n)$ crece a una velocidad no mayor que $g(n)$.

Propiedades de $O()$

Para cualquier par de funciones $f(n) \wedge g(n)$ se verifican las siguientes propiedades:

$$cO(f(n)) \text{ es } O(f(n))$$

$$O(f(n) + g(n)) \text{ es } \max(O(f(n)), O(g(n)))$$

$$O(f(n)) + O(g(n)) \text{ es } O(f(n) + g(n))$$

$$O(f(n))O(g(n)) \text{ es } O(f(n)g(n))$$

$$O(O(f(n))) \text{ es } O(f(n))$$

Funciones de complejidad algorítmica más usuales ordenadas de mayor a menor eficiencia son:

- $O(1)$: Complejidad constante
- $O(\log(n))$: Complejidad logarítmica, suele aparecer en algoritmos con iteración o recursión no estructurada (búsqueda binaria).
- $O(n)$: Complejidad lineal, es en general una complejidad buena y bastante usual. Suele aparecer, en la evaluación de ciclos simples cuando la complejidad de las operaciones interiores es constante o en algoritmos de recursión estructurada.
- $O(n \log(n))$: Aparece en algoritmos con recursión no estructurada (por ejemplo Quick sort) y se considera una complejidad buena.
- $O(n^2)$: Complejidad cuadrática, aparece en ciclos o recursiones doblemente anidadas.
- $O(n^3)$: Complejidad cubica, aparece en ciclos o recursiones triples, para un valor grande de n empieza a crecer en exceso.
- $O(n^k)$: Complejidad polinómica ($n > 3$), si k crece la complejidad del programa es bastante mala.
- $O(2^n)$: Complejidad exponencial, debe evitarse en la medida de lo posible, puede aparecer en rutinas recursivas que contengan dos o más llamadas internas. En problemas donde aparece esta complejidad suele hablarse de “*explosión combinatoria*”.

Notación Asintótica

Ejercicios

1. Comparar los siguientes pares de funciones usando notación asintótica.

$$f(n)$$

$$10^{-3} n^4$$

$$n^2$$

$$\log(n)$$

$$2^{n^2}$$

$$100n + \log_{10}(n)$$

$$\log(n)$$

$$n^{\log(n)}$$

$$g(n)$$

$$10^3 n^3$$

$$n \log(n)$$

$$\log(\log(n))$$

$$2^{2^n}$$

$$n + (\log(n))^2$$

$$\log(n^2)$$

$$n$$



Análisis

Complejidad de asignaciones y expresiones simples: El tiempo de ejecución de toda instrucción de asignación simple, de la evaluación de una expresión formada por términos simples o de toda constante es **$O(1)$** .

Secuencia de Instrucciones: El tiempo de una secuencia de instrucciones es igual a la suma de sus tiempos de ejecución respectivos. Para una secuencia de dos instrucciones I_1 e I_2 tenemos que su tiempo de ejecución viene dado por la suma de los tiempos de ejecución de I_1 e I_2 , es decir:

$$T(I_1 ; I_2) = T(I_1) + T(I_2)$$

Aplicando la regla de la suma su orden es:

$$O(T(I_1 ; I_2)) = O(T(I_1) + T(I_2))$$

$$O(T(I_1 ; I_2)) = \max(O(T(I_1)), O(T(I_2)))$$

Instrucciones Condicionales: El tiempo de ejecución requerido por una instrucción condicional **IF – THEN**, es el necesario para evaluar la condición, mas el requerido para el conjunto de instrucciones que se ejecutan cuando se cumple la condición.

$$T(\text{IF} - \text{THEN}) = T(\text{condición}) + T(\text{rama THEN})$$

El tiempo para una instrucción condicional del tipo **IF – THEN – ELSE** es el resultante de evaluar la condición mas el máximo entre los requeridos para ejecutar el conjunto de instrucciones de las ramas **THEN** y **ELSE**.

$$T(\text{IF} - \text{THEN} - \text{ELSE}) = T(\text{condición}) + \max(T(\text{rama THEN}), T(\text{rama ELSE}))$$

Si aplicamos la regla de la suma tenemos que:

$$O(T(\text{IF-THEN-ELSE})) = O(T(\text{condición})) + \max(O(T(\text{rama THEN})), O(T(\text{rama ELSE})))$$

Instrucciones de iteración: La complejidad en tiempo de un ciclo **FOR** es el producto del numero de iteraciones por la complejidad de las instrucciones del cuerpo del ciclo. Para ciclos **WHILE**, **LOOP** y **REPEAT** se sigue la regla anterior considerando la estimación del numero de iteraciones para el peor caso posible.

Llamadas a procedimientos: La evaluación de la complejidad de la llamada a un procedimiento esta dada por el tiempo requerido para ejecutar el cuerpo del procedimiento, no se tiene en cuenta el tiempo necesario para efectuar el paso de los argumentos.

Ejemplos:

1) $x = x + 1;$ ($x++$)
Es $O(1)$; Constante.

2) $\text{for}(i=1; i < n; i++)$
 $x=x+1;$

$$O(T(n)) = O\left(\sum_{i=1}^n 1\right)$$

$$O(T(n)) = O(n); \text{ Lineal}$$

```
3) for(i=1; i < n; i++)  
    for(j=1; j < n; j++)  
        for(k=1; k<n; k++)  
            x=x+1;
```

$$O(T(n)) = O\left(\sum_{i=1}^n \sum_{j=1}^n \sum_{k=1}^n 1\right)$$

$$O(T(n)) = O(n^3);$$


```
4) if ((n mod 2) == 0)
    for(i=1 i < n ; i++)
        x=x++;
```

$$\begin{aligned}O(T(n)) &= O(T(\text{condicion}) + O(T(\text{ramaTHEN}))) \\&= O(O(1) + T(\text{FOR})) \\&= O(1) + O\left(\sum_{i=1}^n 1\right) \\&= O(n); \quad \text{Lineal}\end{aligned}$$

5) `i=1;`
 `while(i<n){`
 `x++;`
 `i=i+2;`
 `}`

$$\begin{aligned} O(T(n)) &= \max(O(1), O(T(WHILE))) \\ &= \max(O(1), O(\sum_{i=1}^{\text{int}(n/2)} 1)) \\ &= \max(O(1), O(\text{int}(n/2))) \\ &= \max(O(1), O(n)) \\ &= O(n); \text{ Lineal} \end{aligned}$$

6) for(i=1; i<n; i++)
 for(j=1; j<i; j++)
 x++;

$$\begin{aligned} O(T(n)) &= O\left(\sum_{i=1}^n \sum_{j=1}^i 1\right) \\ &= O\left(\sum_{i=1}^n i\right) \\ &= O\left(\frac{n(n+1)}{2}\right) \\ &= O(n^2) \end{aligned}$$

7) $x=1;$
 $\text{while}(x < n)$
 $x=2*x;$
 $O(T(n)) = \max(O(1), O(T(WHILE)))$

El cálculo del número de iteraciones del ciclo while equivale a calcular el valor de la variable t en el siguiente conjunto de instrucciones:

```
x=1;
t=1;
while(x < n){
    x=2*x;
    t++;
}
```

En este caso tenemos que:

$$n = 8 \Rightarrow t = 4$$

$$n = 16 \Rightarrow t = 5$$

$$n = 32 \Rightarrow t = 6$$

De donde podemos deducir que:

$$2^{t-1} \geq n; \text{ despejando } t \text{ nos queda}$$

$$\log_2 2^{t-1} = \log_2 n$$

$$(t-1) \log_2 2 = \log_2 n$$

$$t = \log_2 n + 1$$

Por lo tanto la complejidad del ciclo while es:

$$O(T(\text{WHILE}))=O(\log n + 1)$$

$$O(T(\text{WHILE}))=O(\log n)$$

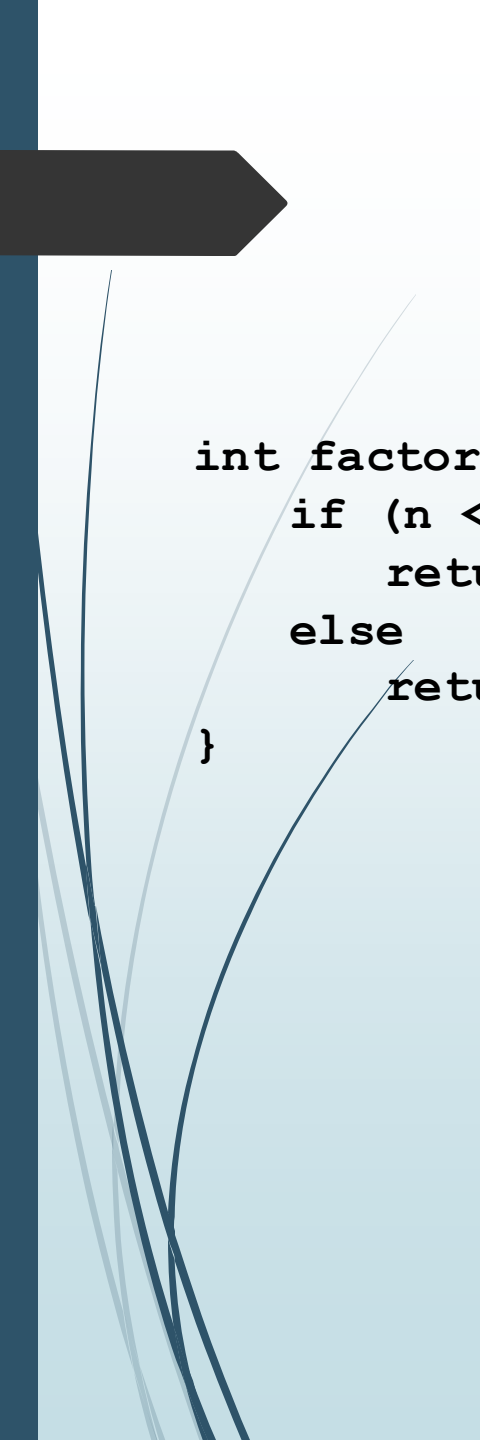
Obteniendo finalmente:

$$O(T(n))=\max(O(1), O(\log n))$$

$$O(T(n))=O(\log n).$$

Que calcula el algoritmo siguiente y cuál es su complejidad.

```
int i, aux, menor, mayor:
if (n<=1){
    return 1;
}
else{
    mayor = 1;
    menor = 1;
    for (i = 2, i <n, i++){
        aux = menor;
        menor = mayor;
        mayor = aux + menor;
    }
    return mayor;
}
```

A dark grey arrow points right from the left edge. Several thin, curved lines in shades of blue and grey sweep from the bottom left towards the center of the slide.

```
int factorial(int n){  
    if (n <= 1)  
        return (1);  
    else  
        return (factorial(n-1) * n);  
}
```


Análisis de Algoritmos

Para el ejemplo del caso recursivo se plantea el problema en términos recurrentes:

$$n! = \begin{cases} 1 & \text{si } n = 0 \\ n(n-1)! & \text{si } n > 0 \end{cases}$$

lo que nos interesa es determinar el tiempo de ejecución del algoritmo $T(n)$, a partir de la función podemos plantear la siguiente expresión que relaciona el tiempo de ejecución en términos recurrentes:

$$T(n) = \begin{cases} 1 & \text{si } n = 0 \\ T(n-1) + 1 & \text{si } n > 0 \end{cases}$$

Análisis de Algoritmos

Para la solución de este tipo de ecuaciones se plantean 4 métodos:

- a) Sustituir las ecuaciones por su igualdad hasta llegar a cierto $T(n_0)$ conocido. Este método se llama de expansión de recurrencia.
- b) Elegir una función $f(n)$ cota superior y una función $g(n)$ cota inferior del mismo orden y usar la ecuación de recurrencia para probar que $g(n) \leq T(n) \leq f(n) \quad \forall n$, a este método se le llama de acotación.
- c) Suponer una solución $f(n)$ y usar la recurrencia para demostrar que $T(n) \leq f(n)$.
- d) Emplear la solución general para ciertas ecuaciones de recurrencia de tipos comunes.

$$\begin{aligned}
 T(n) &= T(n-1) + 1 \\
 &= (T(n-2) + 1) + 1 \\
 &= T(n-2) + 2 \\
 &= T(T(n-3) + 1) + 2 \\
 &= T(n-3) + 3 \\
 &\vdots
 \end{aligned}$$

$$T(n) = T(n-k) + k \quad \text{para el } k\text{-ésimo término}$$

(En este punto debemos considerar el valor de $T(n)$ conocido)

Si elegimos un $k = n$ tenemos que

$$T(n) = T(0) + n$$

$$T(n) = 1 + n$$

$$O(T(n)) = O(n) \Rightarrow \text{complejidad de orden lineal}$$

```
int recursiva(int n) {  
    if (n <= 1)  
        return (5);  
    else  
        return (recursiva(n-1) + recursiva(n-1));  
}
```

Expresando la función en términos de tiempo de ejecución recursivo tenemos que:

$$T(n) = \begin{cases} 1 & \text{si } n \leq 1 \\ 2T(n-1) + 1 & \text{si } n > 1 \end{cases}$$

y aplicando el método de expansión de recurrencias para resolverlo tenemos que:

$$\begin{aligned}
 T(n) &= 2T(n-1) + 1 \\
 &= 2(2T(n-2) + 1) + 1 \\
 &= 2^2 T(n-2) + 3 \\
 &= 2^3 T(n-3) + 7 \\
 &\vdots \\
 T(n) &= 2^k T(n-k) + 2^k - 1
 \end{aligned}$$

$$\begin{aligned}
 \text{Si } k &= n-1 \\
 2^{n-1} T(1) &+ 2^{n-1} - 1 \\
 2^{n-1} + 2^{n-1} - 1 \\
 2 * 2^{n-1} - 1 \\
 2^n - 1
 \end{aligned}$$

$$O(T(n)) = O(2^n) \Rightarrow$$

Complejidad exponencial

```
int recursiva_1(int n) {  
    if (n <= 1)  
        return (1);  
    else  
        return (2*recursiva_1(n / 2));  
}
```

Expresando la función en términos de tiempo de ejecución recursivo tenemos que:

$$T(n) = \begin{cases} 1 & \text{si } n \leq 1 \\ T(n/2) + 1 & \text{si } n > 1 \end{cases}$$

y aplicando el método de expansión de recurrencias para resolverlo tenemos que:

$$\begin{aligned}T(n) &= T(n/2) + 1 \\&= T(((n/2)/2) + 1) + 1 \\&= T(n/4) + 2 \\&= T(((n/4)/2) + 1) + 2 \\&= T(n/8) + 3 \\&\vdots \\&= T(n/2^k) + k\end{aligned}$$

Si $k = \log_2 n$

$$T(n/2^{\log_2 n}) + \log_2 n$$

$$O(T(n)) = O(\log_2 n) \Rightarrow \text{complejidad de orden logarítmico}$$



Análisis de Algoritmos



FIN