ANÁLISIS DE ALGORITMOS

ESTRUCTURAS DE DATOS
y ALGORITMOS
LCC - TUPW

Análisis de algoritmos
Criterios

Corrección(resultados correctos) :¿da solución al problema en un número finito de pasos?

- **Simplicidad**: facilita su verificación, el estudio de su eficiencia y su mantenimiento.

- **Eficiencia:** *cantidad de recursos, principalmente memoria y tiempo, que necesita para ejecutarse.*

Análisis de algoritmos
Eficiencia

El estudio de la eficiencia de algoritmos:

- permite **medir el costo**, en tiempo, memoria u otro recurso, que consume un algoritmo para encontrar la solución a un problema y
- ofrece la posibilidad de **comparar distintos algoritmos** que resuelven un mismo problema

Análisis de algoritmos
Tiempo de ejecución

El tiempo de ejecución de un algoritmo va a depender de diversos factores:

- **los datos de entrada**
- **la calidad** del código generado por el compilador para crear el programa objeto
- la **naturaleza y rapidez de las instrucciones de máquina del procesador** concreto que ejecute el programa, y
- **la complejidad intrínseca del algoritmo.**

Análisis de algoritmos
Complejidad Tiempo de Ejecución

Analizar la complejidad de un algoritmo y caracterizar su costo

Hay dos estudios posibles :

- Uno que ofrece una **medida *empírica***(a posteriori), consistente en medir el tiempo de ejecución del algoritmo para unos valores de entrada dados y en un ordenador concreto.

- Otro que proporciona una **medida *teórica*** (a priori), que consiste en determinar ***matemáticamente*** el tiempo de ejecución del algoritmo como ***función del tamaño de los datos de entrada***.

Tiempo de ejecución

Reglas para el cálculo de unidades de tiempo de instrucciones

1. Las declaraciones no consumen tiempo.

2. Sentencias simples: 1 ut

3. Expresiones aritméticas, o relacionales : 1 ut

4. Ciclos incondicionales:

T = (tiempo del cuerpo * número iteraciones) +

(tiempo de inicialización, testeos e incremento de variable de control)

5. Ciclos condicionales: Si la cantidad de iteraciones varía en función del valor de la variable de control, el cálculo del tiempo se expresa como una sumatoria.

6. Ciclos incondicionales anidados: Tiempo de ejecución del bloque por el producto de los tamaños de todos los ciclos.

Tiempo de ejecución

Reglas para el cálculo de unidades de tiempo de instrucciones

7. Sentencias alternativas:

Selección doble

Si (condición) →
Entonces S1 → t1
Sino S2 t2
Finsi

➡ T= tiempo de condición + máximo (t1, t2)

Alternativa Múltiple

T= tiempo de condición + máximo (t1, t2,…,tk)

Tiempo de ejecución
Ejemplo

```
Buscar( A[1..n];c)
  j←1
  Mientras (A[j]<c) y (j<n) hacer
    j ← j+1
  Fin Mientras
  Si  A[j]=c
    entonces Retornar Éxito
    sino       Retornar Fracaso
  Fin Si
Fin Buscar
```

Eficiencia de algoritmos
Ejemplo Tiempo de ejecución

Buscar(A[1..n];c)

j←1 — 1 ut - Operación Elemental

Mientras (A[j]<c) y (j<n) hacer — 4 ut : 2 comp., 1 and, 1 acceso a vector

j ← j+1 — 2 ut : 1 incremento y 1 asignación

Fin Mientras

Si A[j]=c — 2 ut : 1 acceso a vector y 1 comparación

entonces Retornar Éxito — 1 ut

sino Retornar Fracaso — 1 ut

Fin Si

Fin Buscar

t(n) - tiempo de ejecución de un algoritmo : número de operaciones ejecutadas por un ordenador idealizado para una entrada de tamaño n

Complejidad de algoritmos
Tiempo de ejecución

Principio de Invarianza

Dado un algoritmo, y dos implementaciones I_1 e I_2, que tardan $t_1(n)$ y $t_2(n)$ segundos para resolver un caso de tamaño n, entonces siempre existen constantes positivas c y *d* tales que:

$$t_1(n) \leq c t_2(n) \text{ y } t_2(n) \leq d\, t_1(n)$$

$t_1(n)=6n+2$ y $t_2(n)=5(6n+2)$) la segunda implementación consume 5 veces mas de tiempo.

Por este principio, todas las implementaciones de un mismo algoritmo tienen las mismas características, aunque la constante multiplicativa pueda cambiar de una implementación a otra

Complejidad de algoritmos
Tiempo de ejecución

El ***tiempo de ejecución de un algoritmo*** va a ser una ***función que mida el número de operaciones elementales*** que realiza el algoritmo ***para un tamaño de entrada dado***.

Suelen estudiarse tres casos:

- *caso mejor:* mínimo valor de t(n) para entradas de tamaño n.
- *caso peor:* máximo valor de t(n) para entradas de tamaño n.
- *caso medio:* valor medio del tiempo de ejecución de todas las entradas de tamaño n. (Aho, Hopcroft y Ullman)

La obtención de los tiempos correspondientes a los tres casos también requiere del análisis de valores posibles de los n datos.

Complejidad de algoritmos
Tiempo de ejecución : Búsqueda Secuencial **Caso Mejor**

A

30	**45**	**72**	**88**	**93**

c = 30 o c < 30

Buscar(A[1..n];c)

J←1 — 1 ut

Mientras (A[j]<c) y (j<n) hacer — 2 ut :1 acc. y 1 comp. (cortocircuito)

j ← j+1

Fin Mientras

Si A[j]=c — 2 ut

entonces Retornar Éxito — 1 ut (c=30) o

sino Retornar Fracaso — 1 ut (c<30)

Fin Si

Fin Buscar

$t(n) = 6$ ut

Complejidad de algoritmos Tiempo de ejecución: Búsqueda Secuencial **Caso Peor**

30	**45**	**72**	**88**	**93**

c = 93 o c > 93

```
Buscar( A[1..n];c)
  J←1
  Mientras (A[j]<c) y (j<n) hacer
    j ← j+1
  Fin Mientras
  Si A[j]=c
    entonces Retornar Éxito
    sino Retornar Fracaso
  Fin Si
Fin Buscar
```

1 ut
4 (n-1) +2(c=93) o 4(c=100) ut
2 (n-1)

2 ut
1 ut(c=93) o
1 ut(c>93)

$$1 + \sum_{1}^{n-1} (4+2)+(4o2)+2+1$$

$$t(n)=6n+2$$

Tiempo de ejecución
Caso medio

Caso Medio: Es el tiempo medio esperado sobre todas las posibles entradas de tamaño N.
En este caso se considera una distribución de probabilidad sobre las entrada

A

30	45	72	88	93

c = 30 o c=45 o c=72 o c=88 o c=93 o c no se encuentra en A.

Todas las componentes del arreglo (A[i], 1<=i<=n), tienen la misma probabilidad (1/n), de contener el elemento c.

$$1\frac{1}{n}+2\frac{1}{n}+3\frac{1}{n}+..+n\frac{1}{n}=\frac{1}{n}\sum_{i=1}^{n}i=\frac{1}{n}\frac{n(n+1)}{2}=\frac{n+1}{2}$$

El costo (complejidad) del caso medio se determina mediante el número medio de elementos del conjunto analizado o número medio de veces que se ejecuta el ciclo.

Tiempo de ejecución

Caso medio

Número medio de veces
que se ejecuta el algoritmo
((n+1)/2)

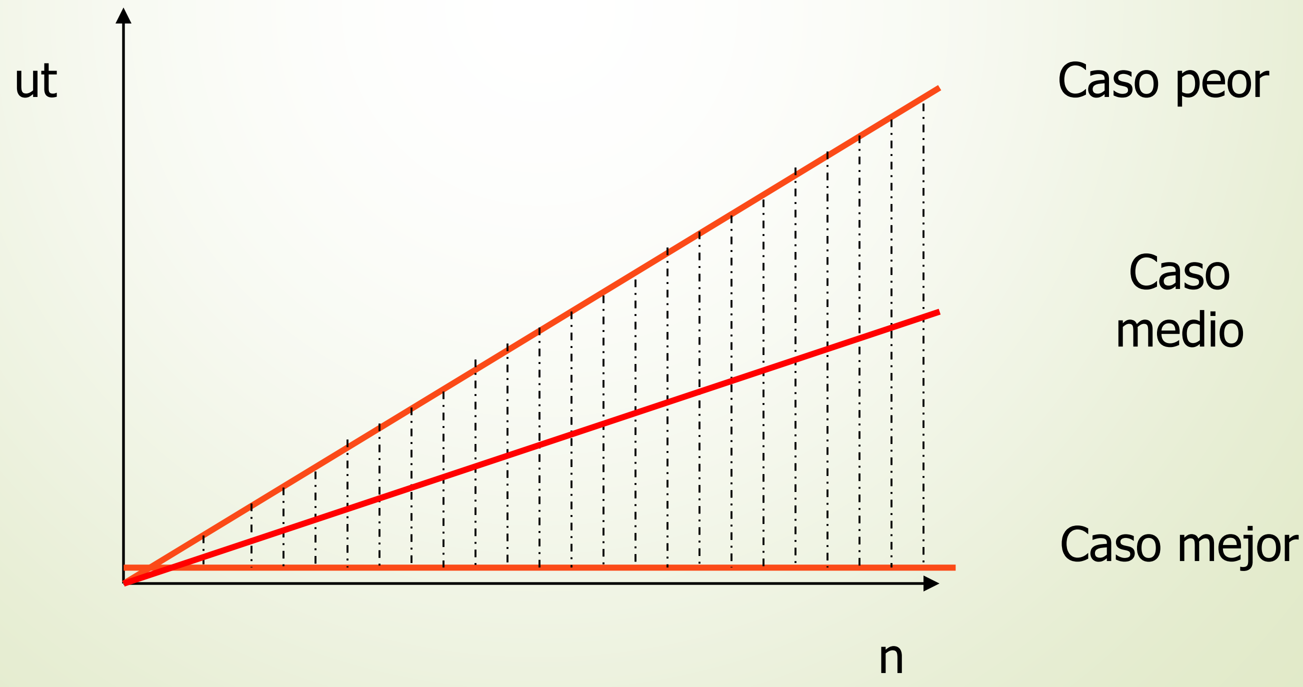
Buscar(A[1..n];c)	
j←1	1 ut
Mientras (A[j]<c) y (j<n) hacer	((n+1) / 2) 4 ut
j ← j+1	((n+1) / 2) 2 ut
Fin Mientras	
Si A[j]=c	2 ut
entonces Retornar Éxito	1 ut o
sino Retornar Fracaso	1 ut
Fin Si	----------------------------------
Fin Buscar	

$$1+\frac{(n+1)}{2}(4+2)+2+1=\frac{(n+1)}{2}6+4=(n+1)3+4$$

t(n) = 3n+7

Complejidad de algoritmos
Tiempo de ejecución
Búsqueda Secuencial

- ***caso mejor***: $t(n) = 6$ ⇒ $\mathbf{t(n) = c0}$
- ***caso peor***: $t(n)=6n+2$ ⇒ $\mathbf{t(n) = c1 \cdot n+c2}$



Complejidad de Algoritmos

Medidas Asintóticas

Una medida asintótica es un **conjunto de funciones que muestran un comportamiento similar cuando los argumentos toman valores muy grandes (∞)**.

En análisis de algoritmos una **cota ajustada asintótica** es una función que sirve de cota de otra función, tanto superior como inferior, cuando el argumento tiende a infinito.

Las **notaciones asintóticas** son lenguajes **que** nos permitan analizar el tiempo de ejecución de un **algoritmo** identificando su comportamiento si el tamaño de entrada para el **algoritmo** aumenta. Esto también se conoce como la **tasa de crecimiento** de un **algoritmo**.

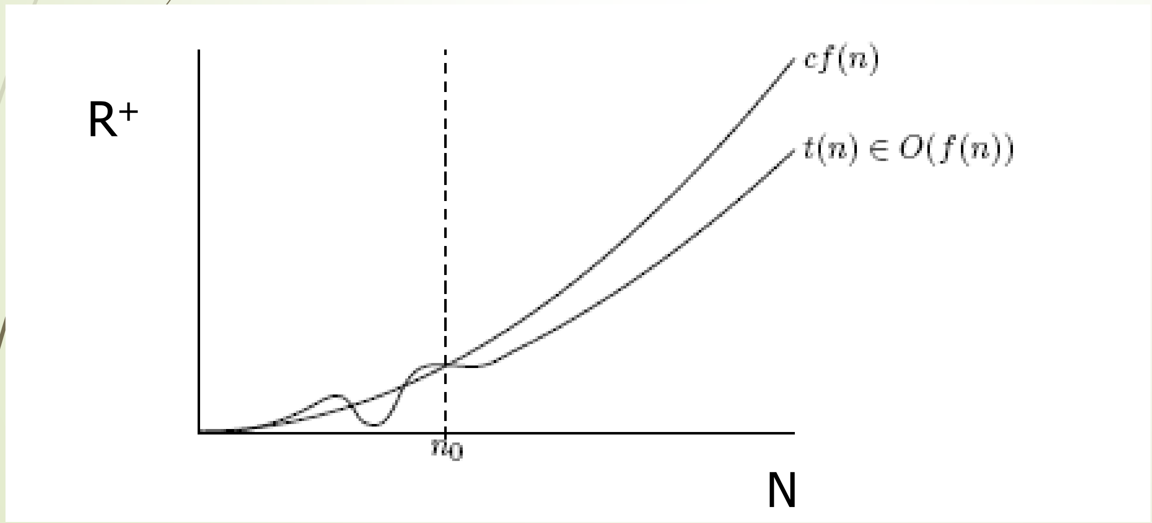
Complejidad de Algoritmos

Medidas Asintóticas

Sea $f: N \rightarrow R^+$,

t(n) es O(f(n))

si $\exists c \in R^+, \exists n_0 \in N, \forall\, n \geq n0 : t(n) \leq cf(n)$



$t(n) = 6n+2$,
$f(n) = n$
$C=7$ y $n0=2$,
$6n+2 \leq 7n$

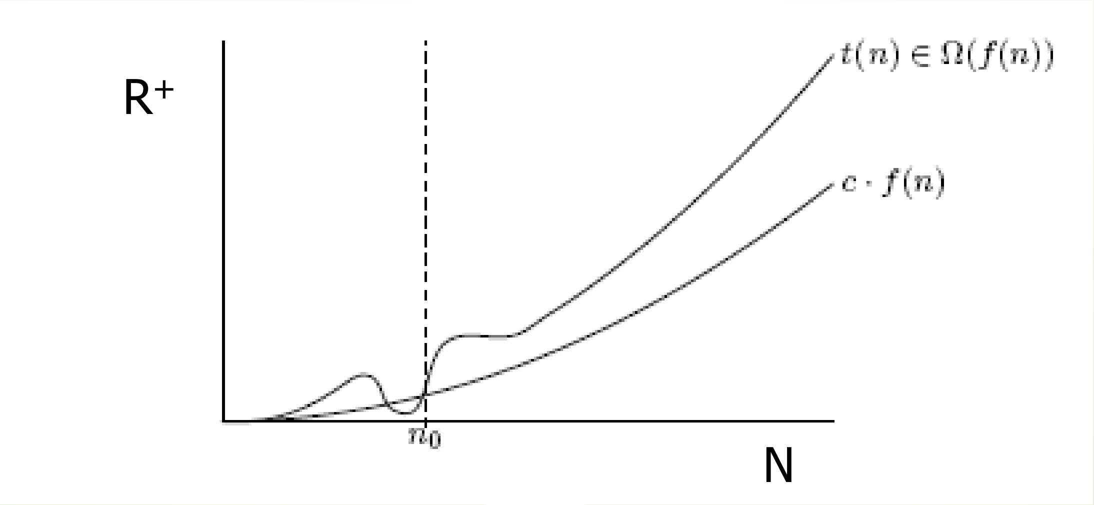
La notación asintótica sirve para indicar la velocidad de crecimiento de la función del tiempo de ejecución de un algoritmo

Complejidad de Algoritmos
Medidas Asintóticas

Sea $f: N \rightarrow R^{+}$,

t(n) es Ω(f(n))

si $\exists c \in R^{+}, \exists n_0 \in N, \forall\, n \geq n0 : t(n) \geq cf(n)$



Complejidad de Algoritmos

Medidas Asintóticas

Propósito: caracterizar el costo de un algoritmo mediante funciones simples, que acoten superior e inferiormente el costo de toda instancia, para **n** suficientemente grandes.

Se definen familias de cotas, *clases de equivalencia*, que corresponden a las funciones que crecen de la misma forma

Complejidad de Algoritmos

Notación O

Cota Superior – Notación O

$$O(f)=\{t: N \rightarrow R^+ \ / \ \exists c \in R^+, \exists n0 \in N, \forall \ n \geq n0 : t(n) \leq cf(n)\}$$

Dada una función $f{:}N \rightarrow R^+$, llamamos **orden de *f*** al conjunto de todas las funciones de N en R^+ **acotadas superiormente** por un múltiplo real positivo de f para valores de n suficientemente grandes.

$t_1(n) = 6n+2 \in O(f(n))$

$t_2(n) = 3n+7 \in O(f(n))$

$f(n) = n$

$t_1(n) = 6n+2 \in O(n)$

$t_2(n) = 3n+7 \in O(n)$

Complejidad de Algoritmos

Notación O

Observaciones:

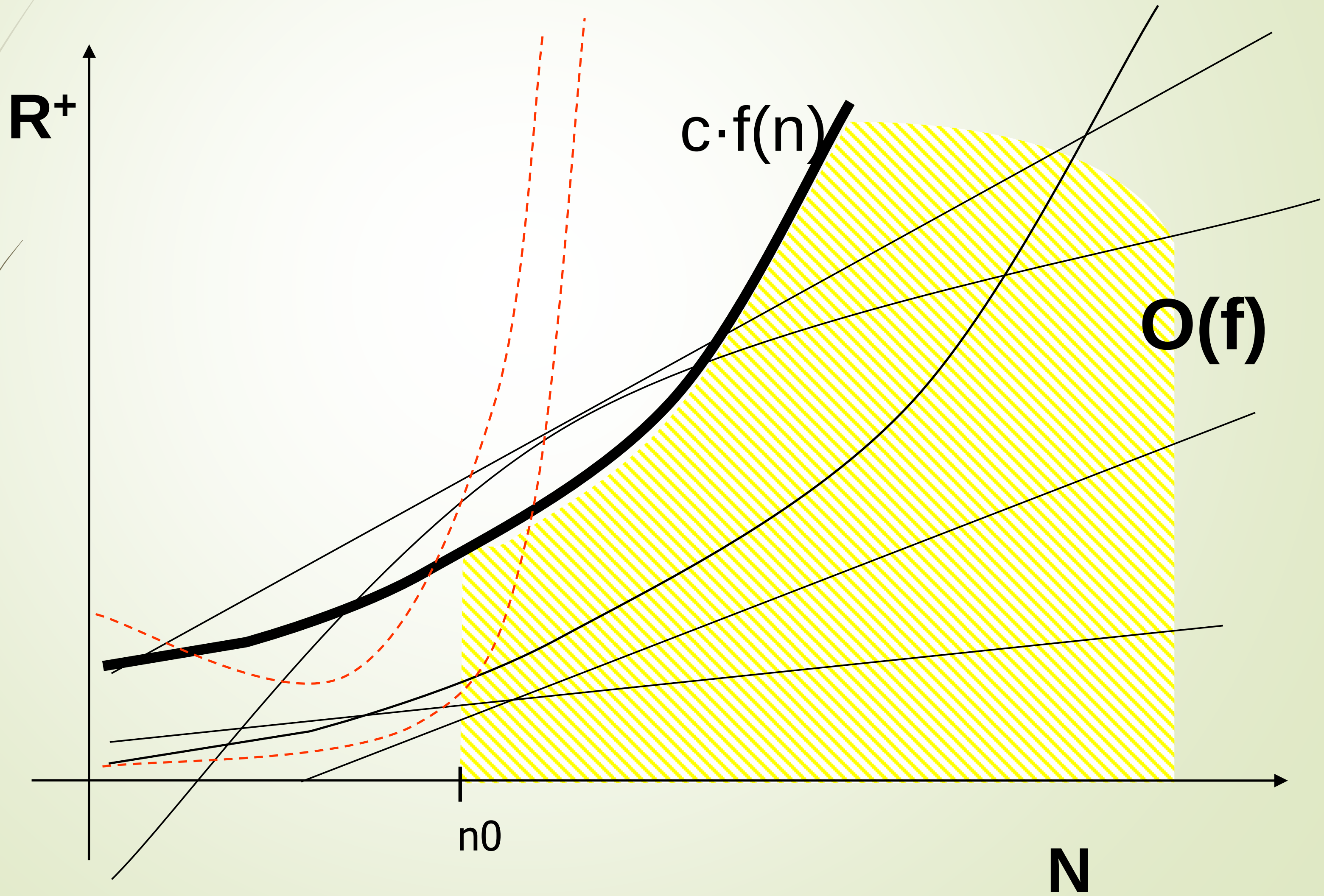
•**O(f)** es un **conjunto de funciones**, no una función.

•"**Valores de n** suficientemente grandes..." pues no importa lo que pase para valores pequeños.

•"La definición es aplicable a **cualquier función de N en R que caracterice** el uso de algún recurso, no sólo las que representan tiempos de ejecución.

Complejidad de Algoritmos

Notación O



Complejidad de Algoritmos

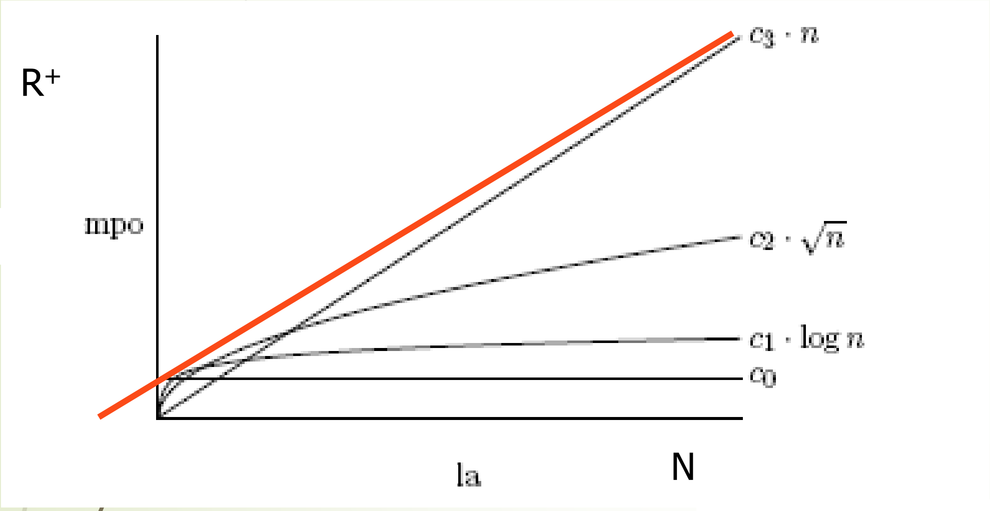
Notación O

Se dice que O(f(n)) define un **"orden de complejidad"**. Como representante de este orden, se escoge a la función- f(n), más sencilla del mismo.
Se tiene:

$O(1)$	orden constante	Orden Sublineal
$O(\log n)$	orden logarítmico	Orden Sublineal
$O(\sqrt{n})$		Orden Sublineal
$O(n)$	orden lineal	Orden Lineal
$O(n \log n)$	orden casi lineal	Orden Superlineal
$O(n^2)$	orden cuadrático	Orden Superlineal
$O(n^3)$	orden cúbico	Orden Superlineal
$O(n^a)$	orden polinómico de grado $a (a > 3)$	Orden Superlineal
$O(a^n)$	orden exponencial $(n > 2)$	Orden Superlineal
$O(n!)$	orden factorial	

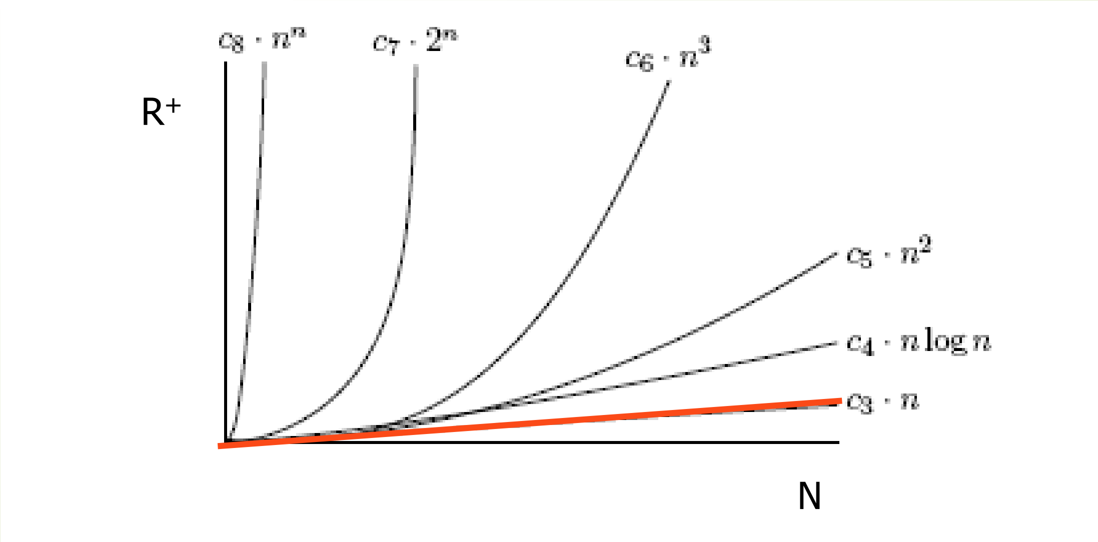
Complejidad de Algoritmos

Órdenes de Complejidad



Crecimiento Lineal y Sublineal

Crecimiento Lineal y Superlineal



Complejidad de Algoritmos

Notación O

Propiedades

- $O(\ f(n)\ ) + k \quad = \quad O(\ f(n)\ )$
- $O(\ k\ f(n)\ ) \quad = \quad O(\ f(n)\ )$
- $O(\ f(n)\ ) * O(\ g(n)\ ) = O(\ f(n) * g(n)\ )$
- $O(\ f(n)\ ) + O(\ g(n)\ ) = \max\ (\ O(\ f(n)\ ), O(\ g(n)\ )\ )$

Complejidad de Algoritmos

Órdenes de Complejidad

n	lg n	n lg n	n^2	n^3	2^n	3^n	n!
1	0	0	1	1	2	3	1
2	1	2	4	8	4	9	2
4	2	8	16	64	16	81	24
8	3	24	64	512	256	6.561	40.320
16	4	64	256	4.096	65.536	43.046.721	20.922.789.888.000
32	5	160	1.024	32.768	4.294.967.296	¿ ?	¿ ?
64	6	384	4.096	262.144	*	¿?	¿ ?
128	7	896	16.384	2.097.152	**	¿?	¿ ?

* el número de instrucciones que puede ejecutar un supercomputador de 1 GFLOP en 500 años.

** sería 500 billones de veces la edad del universo (20 billones de años) en nanosegundos.

D. Riley en [Ril-87]

Complejidad de Algoritmos

Notación Ω

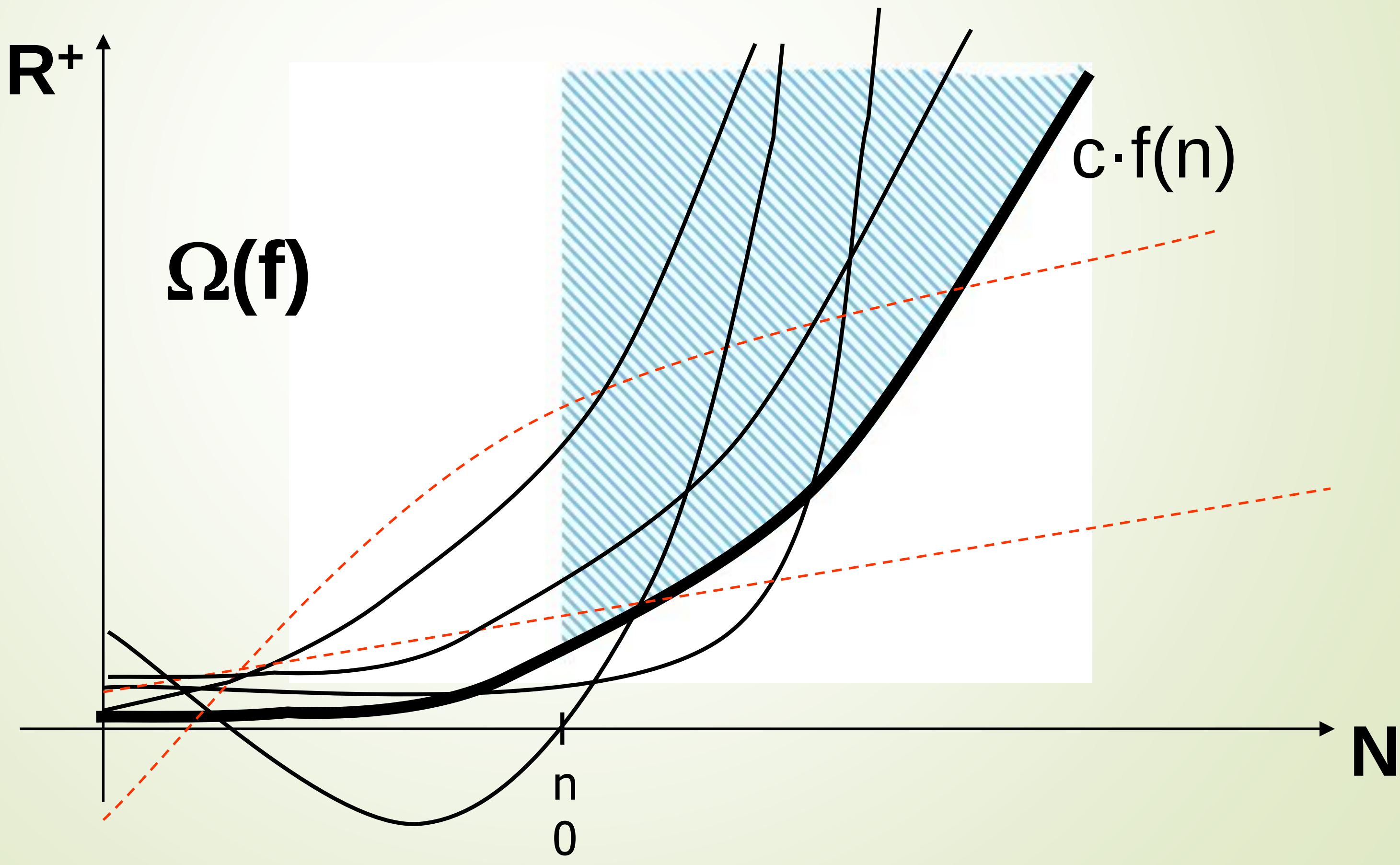
Cota Inferior – Notación Ω

$$\Omega(f)=\{t: N \rightarrow R^{+} \;/\; \exists c \in R^{+}, \exists n0 \in N, \forall\, n \geq n0 : cf(n) \leq t(n)\,\}$$

Dada una función $f\!:\!N \rightarrow R^{+}$, llamamos **omega de *f*** al conjunto de todas las funciones de N en R^{+} **acotadas inferiormente** por un múltiplo real positivo de f para valores de n suficientemente grandes.

Complejidad de Algoritmos

Notación Ω



Eficiencia de algoritmos

¿Si el hardware es
cada vez mas potente y accesible,
siempre es importante
tener en cuenta la eficiencia?

El análisis de eficiencia de algoritmos no es tan importante si, entre otros aspectos:

- El programa va a ejecutarse pocas veces
- El programa va a ejecutarse con pocos datos
- No es crítico el uso del recurso (tiempo por ejemplo)

Eficiencia de algoritmos

¿Porqué es importante

tener en cuenta

la eficiencia de un algoritmo?

Eficiencia de algoritmos
Tiempo de ejecución: Búsqueda Binaria
Caso Peor

```
Binaria (A[0..n-1], n, x)
  li=0
  ls=n-1
  mi=( li+ls) div 2
  Mientras((li<=ls) y (x <> A[mi]))
     Si   ( x < a[mi] )
                 Entonces  ls = mi -1
                 Sino       li = mi+1
     FinSi
     mi =( li+ ls) div 2
  FinMientras
  Si  (li > ls)
     Entonces   Retornar Fracaso
     Sino        Retornar Exito
   Finsi
Fin
```

A

1	1	2	3	4	5	5	5	6	9

n: 10   x: ?

x: 4

x: 1

x: 7

Eficiencia de algoritmos

Tiempo de ejecución: Búsqueda Binaria

Caso Peor

```
Binaria (A[0..n-1], n, x)
  li=0
  ls=n-1
  mi=( li+ls) div 2
  Mientras((li<=ls) y (x <> A[mi]))
     Si   ( x < a[mi] )
                  Entonces  ls = mi -1
                  Sino         li = mi+1
     FinSi
     mi =( li+ ls) div 2
  FinMientras
  Si  (li > ls)
     Entonces   Retornar Fracaso
     Sino          Retornar Exito
   Finsi
Fin
```

Cuantas veces se ejecuta la iteración?

A   n: 10 x: 7

1	1	2	3	4	5	5	5	6	9

1	1	2	3	**4**	5	5	5	6	9

1	1	2	3	4	5	5	5	6	9

1	1	2	3	4	5	5	**5**	6	9

1	1	2	3	4	5	5	5	6	9

1	1	2	3	4	5	5	5	**6**	9

1	1	2	3	4	5	5	5	**6**	9

iteración	li	ls	mi	A[mi]
1	0	9	4	4
2	5	9	7	5
3	8	9	8	6
4	9	9	9	9
	9	8	9	

Eficiencia de algoritmos
Tiempo de ejecución : Búsqueda Binaria
Caso Peor

La iteración se ejecuta hasta que el subarreglo analizado tiene una sola componente !!!

Numero de iteración	Longitud del espacio de búsqueda
1	n
2	n/2= $n/2^1$
3	n/4= $n/2^2$
------------	----------------
k	$n/2^{k-1}=1$

$$\frac{n}{2^{k-1}} = 1 \Rightarrow n = 2^{k-1} \Rightarrow \log_2 n = (k-1)\log_2 2 \Rightarrow \qquad 1 + \log_2 n = k$$

O(log n)

Eficiencia de algoritmos

Búsqueda Secuencial vs Búsqueda Binaria

N	**Cantidad de Ciclos**	
	Búsqueda Secuencial	**Búsqueda Binaria**
1	1	1
8	8	3
128	128	7
1024	1024	10
8192	8192	13

Eficiencia de algoritmos

Es conveniente:

inicialmente Diseñar algoritmos claros

y

luego prestar atención a su Optimización

Como elegir el mejor algoritmo?

Ejemplo : Ordenamiento

Los algoritmos comunes de ordenamiento pueden dividirse en dos clases según su orden de complejidad:

- **Algoritmos de complejidad cuadrática $0(n^2)$:**

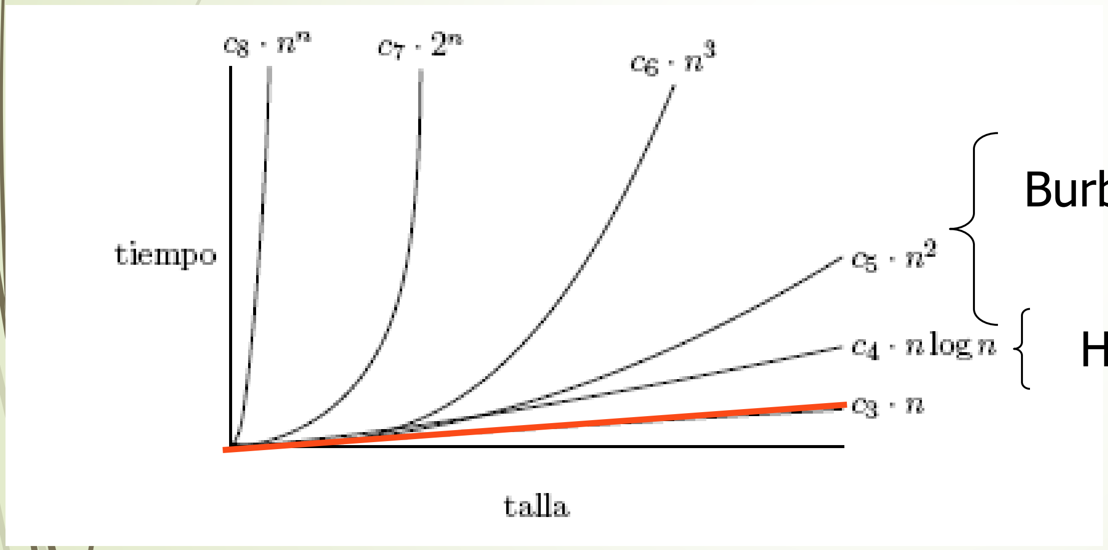
Burbuja, de Inserción, de Selección, Shellsort.

- **Algoritmos complejidad $0(n*log(n))$:**

Heapsort, Mergesort y Quicksort

Como elegir el mejor algoritmo?

Puede ser **Ω(n²)** una cota inferior para el problema de ordenamiento?



Burbuja, de Inserción, de Selección y el Shellsort

Heapsort, Mergesort y Quicksort

NO, a lo sumo la cota inferior es **Ω(n log n)**

¿Como elegir el mejor algoritmo?

P
u
e
d
e

o
c
u
r
r
i
r

- Caso 1:

 En la actualidad la cota inferior de un problema es Ω(n log n) y la complejidad temporal del mejor algoritmo para resolverlo es O(n^2).

- Caso 2:

La cota inferior actual es Ω(n log n) y existe un algoritmo con complejidad temporal O(n log n)

1 – La cota inferior del problema es demasiado baja, por lo que hay que encontrar una cota inferior mas precisa o alta. (Mover la cota inferior hacia arriba)

2- El mejor algoritmo disponible no es bueno, por lo que hay que tratar de encontrar un algoritmo con complejidad temporal mas baja. (Mover hacia abajo la complejidad temporal).

3- Mejorar la cota inferior y tambien mejorar el algoritmo.

¿Como elegir el mejor algoritmo?

Como se sabe que un algoritmo es el óptimo?

Un **algoritmo es el óptimo**, si su complejidad temporal – O- es igual a una cota inferior de este problema - Ω; ya no es posible mejorar mas ni la cota inferior ni el algoritmo.

¿Como elegir el mejor algoritmo? Resumen

- La cota inferior de un problema es la complejidad temporal mínima de todos los algoritmos que pueden aplicarse para resolverlo.
- Si la cota inferior conocida actual es mas baja que la complejidad temporal del mejor algoritmo disponible para resolver el problema, entonces es posible mejorar la cota inferior moviéndola hacia arriba. El algoritmo puede mejorarse moviendo su complejidad temporal hacia abajo.
- Si la cota inferior conocida actual es igual a la complejidad temporal de un algoritmo disponible, entonces ya no es posible mejorar mas ni el algoritmo ni la cota inferior. El algoritmo es un algoritmo óptimo y la cota inferior es la máxima.

Análisis Amortizado

En un análisis amortizado se promedia el tiempo requerido para realizar una secuencia de operaciones. Con este análisis se puede mostrar que el costo promedio de una operación es pequeño aunque una sola operación dentro de la secuencia pueda ser costosa.

El análisis amortizado produce una cota en el tiempo de ejecución de la serie de operaciones (es decir, sigue siendo un análisis del peor de los casos)

Los resultados del análisis amortizado sirven para optimizar el diseño de la estructuras de datos, produciendo entonces estructuras de datos avanzadas

Eficiencia de algoritmos
Referencias

Giles Brassard, Paul Bratley. **Fundamentos de Algoritmia.**

Ralph Grimaldi. **Matemática Discreta y Combinatoria**.

Rosa Guerequeta, Antonio Vallecillo. **Técnicas de Diseño de Algoritmos.** http://www.lcc.uma.es/~av/Libro/indice.html

R.C.T.Lee, S.S.Tseng, R.C.Chang, Y.T. Tsai. **Introducción al diseño y análisis de algoritmos-Un enfoque estratégico**