

Algoritmos y estructuras de datos

Estrategia incremental

CEIS

Escuela Colombiana de Ingeniería

2022-2

Agenda

1 Problema de ordenamiento

Formulación

Diseño

Análisis

Código

2 Diseño

3 Análisis

4 Aspectos finales

Ejercicios

Agenda

1 Problema de ordenamiento

Formulación

Diseño

Análisis

Código

2 Diseño

3 Análisis

4 Aspectos finales

Ejercicios

Formulación

- ▶ **Entrada:** Una secuencia de n números $\langle a_1, a_2, \dots, a_n \rangle$
- ▶ **Salida:** Una permutación $\langle a'_1, a'_2, \dots, a'_n \rangle$ tal que
 $a'_1 \leq a'_2 \leq \dots \leq a'_n$

Algoritmo Correcto: Una algoritmo se dice correcto si para cada instancia de la entrada,

entrega una salida correcta.

Agenda

1 Problema de ordenamiento

Formulación

Diseño

Análisis

Código

2 Diseño

3 Análisis

4 Aspectos finales

Ejercicios

Insertion Sort



Funciona de manera similar a como una persona organiza una mano de cartas.

Insertion Sort

5	2	4	6	1	3
---	---	---	---	---	---

¿Qué condición se cumple siempre?

Insertion Sort

5	2	4	6	1	3
---	---	---	---	---	---

(a)

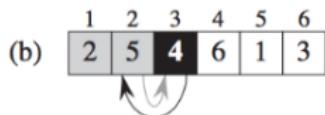
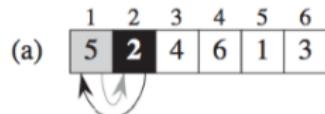
1	2	3	4	5	6
5	2	4	6	1	3



¿Qué condición se cumple siempre?

Insertion Sort

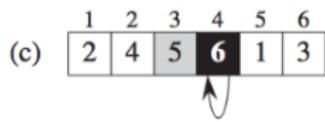
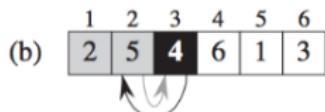
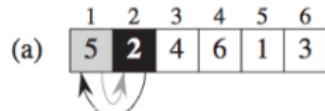
5	2	4	6	1	3
---	---	---	---	---	---



¿Qué condición se cumple siempre?

Insertion Sort

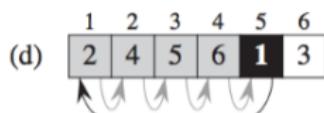
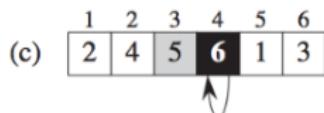
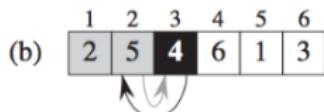
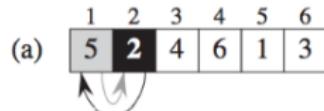
5	2	4	6	1	3
---	---	---	---	---	---



¿Qué condición se cumple siempre?

Insertion Sort

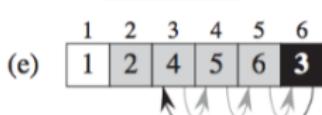
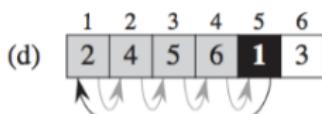
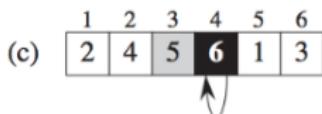
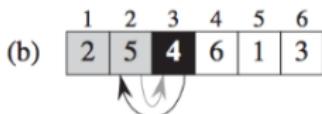
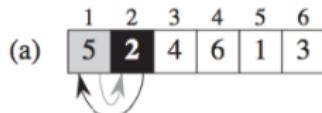
5	2	4	6	1	3
---	---	---	---	---	---



¿Qué condición se cumple siempre?

Insertion Sort

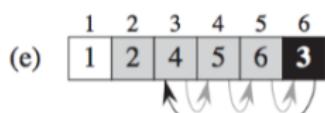
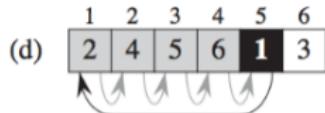
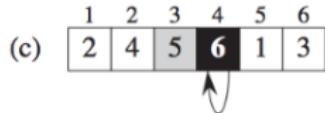
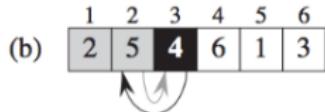
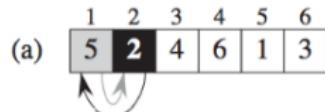
5	2	4	6	1	3
---	---	---	---	---	---



¿Qué condición se cumple siempre?

Insertion Sort

5	2	4	6	1	3
---	---	---	---	---	---

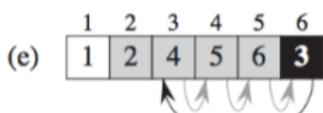
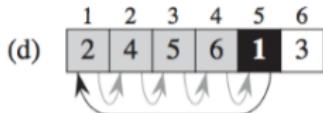
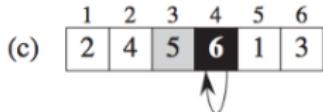
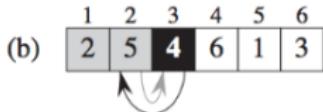
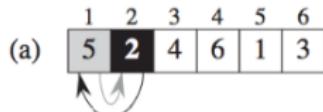


¿Qué condición se cumple siempre?

1	2	3	4	5	6
---	---	---	---	---	---

Insertion Sort

5	2	4	6	1	3
---	---	---	---	---	---



1	2	3	4	5	6
---	---	---	---	---	---

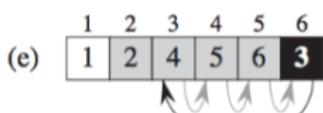
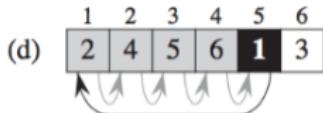
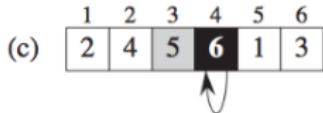
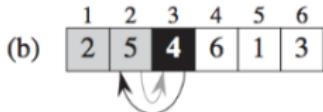
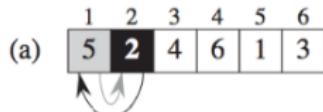
¿Qué condición se cumple siempre?

Los elementos antes del **actual**

- son los originales
- pero ahora están ordenados

Insertion Sort

5	2	4	6	1	3
---	---	---	---	---	---



1	2	3	4	5	6
---	---	---	---	---	---

¿Qué condición se cumple siempre?

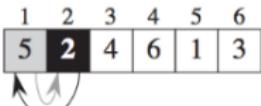
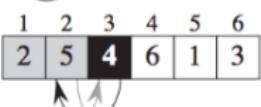
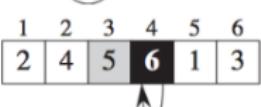
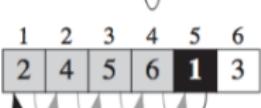
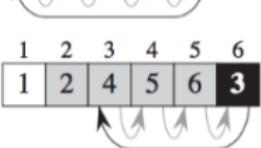
Los elementos antes del **actual**

- son los originales
- pero ahora están ordenados

Este es el INVARIANTE

InsertionSort

5	2	4	6	1	3
---	---	---	---	---	---

- (a) 
- (b) 
- (c) 
- (d) 
- (e) 

INSERTION-SORT(A)

```
1  for  $j = 2$  to  $A.length$ 
2      key =  $A[j]$ 
3      // Insert  $A[j]$  into the sorted sequence  $A[1..j - 1]$ .
4       $i = j - 1$ 
5      while  $i > 0$  and  $A[i] > key$ 
6           $A[i + 1] = A[i]$ 
7           $i = i - 1$ 
8       $A[i + 1] = key$ 
```

1	2	3	4	5	6
---	---	---	---	---	---

Invariante

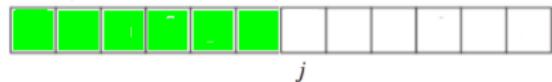


- ▶ Los invariantes del ciclo ayudan a probar que un algoritmo es correcto. Se deben demostrar tres cosas sobre un invariante de ciclo:
 - ▶ Iniciación: El invariante es verdadero antes de la primera iteración.
 - ▶ Estabilidad: Si es verdadero antes de una iteración del ciclo, debe ser verdadero después de la iteración.
 - ▶ Terminación: Cuando el ciclo termina, el invariante es una propiedad importante que ayuda a demostrar que el algoritmo correcto.

InsertionSort

INSERTION-SORT(A)

```
1  for  $j = 2$  to  $A.length$ 
2       $key = A[j]$ 
3          // Insert  $A[j]$  into the sorted sequence  $A[1..j - 1]$ .
4       $i = j - 1$ 
5      while  $i > 0$  and  $A[i] > key$ 
6           $A[i + 1] = A[i]$ 
7           $i = i - 1$ 
8       $A[i + 1] = key$ 
```



Se formula el siguiente invariante:

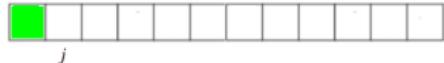
$A[1 \dots j - 1]$ consta de los elementos originalmente en $A[1 \dots j - 1]$, pero ordenados ascendenteamente.

InsertionSort

INSERTION-SORT(A)

```
1  for  $j = 2$  to  $A.length$ 
2      key =  $A[j]$ 
3          // Insert  $A[j]$  into the sorted sequence  $A[1..j - 1]$ .
4       $i = j - 1$ 
5      while  $i > 0$  and  $A[i] > key$ 
6           $A[i + 1] = A[i]$ 
7           $i = i - 1$ 
8       $A[i + 1] = key$ 
```

Iniciación



- ▶ Se debe demostrar que el invariante es cierto antes de la primera iteración, es decir, cuando $j = 2$.
- ▶ El subarreglo $A[1 .. j - 1]$ consiste únicamente del elemento $A[1]$, que es el elemento original en $A[1]$
- ▶ Adicionalmente, $A[1]$ está ordenado ascendente trivial (solo hay un elemento)

InsertionSort

INSERTION-SORT(A)

```
1  for  $j = 2$  to  $A.length$ 
2    key =  $A[j]$ 
3    // Insert  $A[j]$  into the sorted sequence  $A[1 \dots j - 1]$ .
4     $i = j - 1$ 
5    while  $i > 0$  and  $A[i] > key$ 
6       $A[i + 1] = A[i]$ 
7       $i = i - 1$ 
8     $A[i + 1] = key$ 
```

Estabilidad



- ▶ Se debe demostrar la estabilidad del invariante.
- ▶ Como se supone que el invariante vale antes de la ejecución del ciclo, entonces se sabe que los elementos originales están en las posiciones $1, \dots, j-1$ ordenados ascendenteamente en el subarreglo $A[1 \dots j-1]$.
- ▶ El ciclo en la línea 5 mueve $A[j-1], A[j-2], A[j-3]$, etc. una posición a la derecha hasta que encuentra la posición adecuada para $A[j]$ y lo inserta allí (línea 8).
- ▶ Entonces, los elementos originalmente en $1, \dots, j$ están en $A[1 \dots j]$. Además, la inserción de $A[j]$ preserva el orden ascendente de los elementos que al inicio de la iteración estaban en $A[1 \dots j-1]$; entonces $A[1 \dots j]$ está ordenado ascendenteamente.

InsertionSort

INSERTION-SORT(A)

```
1  for  $j = 2$  to  $A.length$ 
2       $key = A[j]$ 
3          // Insert  $A[j]$  into the sorted sequence  $A[1..j - 1]$ .
4       $i = j - 1$ 
5      while  $i > 0$  and  $A[i] > key$ 
6           $A[i + 1] = A[i]$ 
7           $i = i - 1$ 
8       $A[i + 1] = key$ 
```

Terminación



- ▶ El ciclo termina cuando la guarda $j = 2$ to $A.length$ es falsa, es decir, cuando $j > A.length$. Además, por la propiedad de estabilidad se puede suponer que el invariante es cierto.
- ▶ En el cuerpo del ciclo, el valor de j aumenta exactamente en 1; entonces, al finalizar el ciclo, debe ser cierto que $j = n + 1$.
- ▶ Al reemplazar $j = n + 1$ en el invariante, se obtiene que el arreglo $A[1 .. (n - 1) + 1]$ (es decir, $A[1 .. n]$) está ordenado ascendentemente y contiene los valores originalmente en $A[1 .. n]$.
- ▶ Como el subarreglo $A[1 .. n]$ es en realidad la totalidad de A , por el argumento anterior, el algoritmo Insertion-Sort ordena el arreglo A ascendentemente.

Agenda

1 Problema de ordenamiento

Formulación

Diseño

Análisis

Código

2 Diseño

3 Análisis

4 Aspectos finales

Ejercicios

Análisis

- ▶ El análisis de algoritmos trata de identificar los recursos que el algoritmo requiere o utilizará. Ejemplos de estos recursos son la cantidad de memoria, hardware o tiempo de ejecución.
- ▶ Utilizaremos en el curso un modelo tecnológico llamado RAM (Random Access Machine). Este modelo tiene varias características:
 - ▶ Un solo procesador
 - ▶ Instrucciones se ejecutan de manera secuencial
 - ▶ Las operaciones booleanas y aritméticas se presumen eficientes y su costo operativo es constante.

Análisis



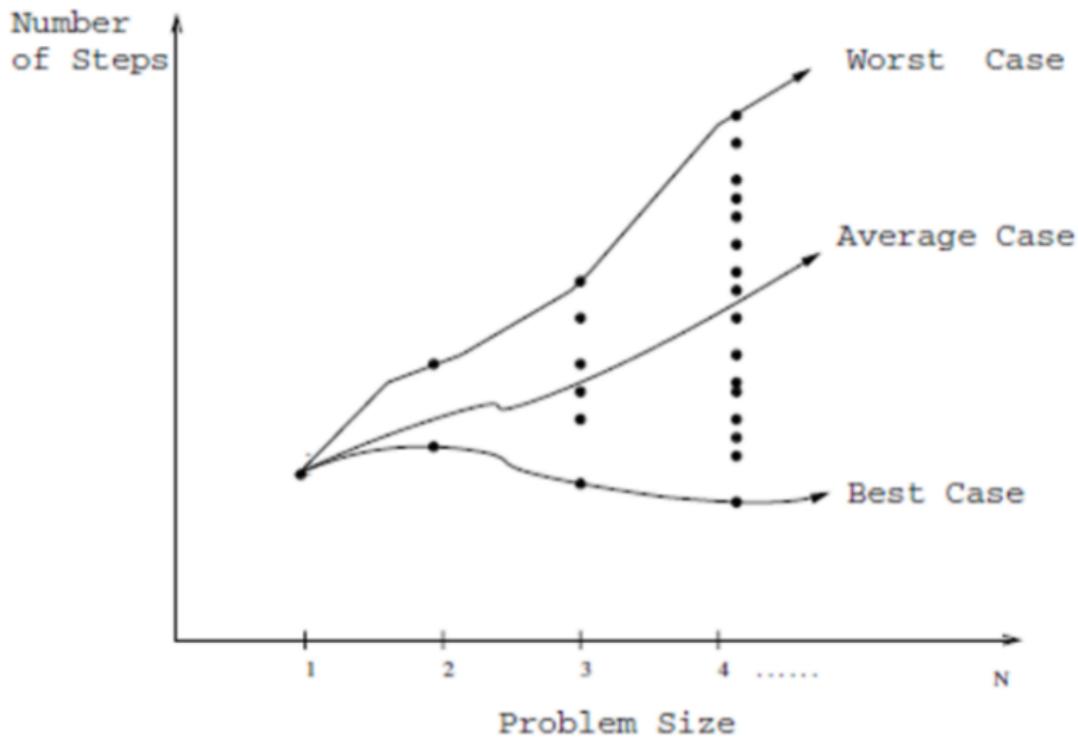
- ▶ Generalmente hablamos del tiempo de ejecución como una función del tamaño de la entrada.
- ▶ **Tamaño de la entrada:** Es dependiente del problema y es una medida de la cantidad de elementos en la entrada.
- ▶ **Tiempo de ejecución:** Número de operaciones primitivas o pasos que son ejecutados para resolver el problema.

Análisis

- ▶ Estamos interesados en encontrar el tiempo de ejecución mas largo.
- I
- ▶ Un límite superior para el tiempo de ejecución del algoritmo con cualquier entrada.
- ▶ Generalmente el caso promedio es igual al peor caso.
- ▶ Muchas veces el peor caso se ejecuta frecuentemente.



Análisis



Análisis

INSERTION-SORT(A)

```
1  for  $j = 2$  to  $A.length$ 
2       $key = A[j]$ 
3      // Insert  $A[j]$  into the sorted
        sequence  $A[1..j - 1]$ .
4       $i = j - 1$ 
5      while  $i > 0$  and  $A[i] > key$ 
6           $A[i + 1] = A[i]$ 
7           $i = i - 1$ 
8       $A[i + 1] = key$ 
```

	<i>cost</i>	<i>times</i>
c_1	n	
c_2	$n - 1$	
	0	$n - 1$
c_4	$n - 1$	
c_5	$\sum_{j=2}^n t_j$	
c_6	$\sum_{j=2}^n (t_j - 1)$	
c_7	$\sum_{j=2}^n (t_j - 1)$	
c_8	$n - 1$	

Análisis

INSERTION-SORT(A)

```
1  for  $j = 2$  to  $A.length$ 
2       $key = A[j]$ 
3      // Insert  $A[j]$  into the sorted
        sequence  $A[1..j - 1]$ .
4       $i = j - 1$ 
5      while  $i > 0$  and  $A[i] > key$ 
6           $A[i + 1] = A[i]$ 
7           $i = i - 1$ 
8       $A[i + 1] = key$ 
```

	<i>cost</i>	<i>times</i>
1	c_1	n
2	c_2	$n - 1$
3	0	$n - 1$
4	c_4	$n - 1$
5	c_5	$\sum_{j=2}^n t_j$
6	c_6	$\sum_{j=2}^n (t_j - 1)$
7	c_7	$\sum_{j=2}^n (t_j - 1)$
8	c_8	$n - 1$

$T(n)$

$$= c_1 n + c_2(n - 1) + c_3(n - 1) + c_4 \sum_{j=1}^{n-1} t_j + c_5 \sum_{j=1}^{n-1} (t_j - 1) + c_6 \sum_{j=1}^{n-1} (t_j - 1) + c_7(n - 1)$$

InsertionSort: $T(n)$

$T(n)$

$$= c_1 n + c_2(n - 1) + c_3(n - 1) + c_4 \sum_{j=1}^{n-1} t_j + c_5 \sum_{j=1}^{n-1} (t_j - 1) + c_6 \sum_{j=1}^{n-1} (t_j - 1) + c_7(n - 1)$$



InsertionSort: $T(n)$

$$T(n) = c_1n + c_2(n-1) + c_4(n-1) + c_5 \sum_{j=2}^n t_j + c_6 \sum_{j=2}^n (t_j - 1) + c_7 \sum_{j=2}^n (t_j - 1) + c_8(n-1)$$

Mejor $t_j = 1; j = 2, 3, \dots, n$

$$\begin{aligned} T(n) &= c_1n + c_2(n-1) + c_3(n-1) + c_4(n-1) + c_7(n-1) \\ &= (c_1 + c_2 + c_3 + c_4 + c_7)n + (c_2 + c_3 + c_4 + c_7) \end{aligned}$$

an + b *Función lineal de n*

Peor

InsertionSort: $T(n)$

$T(n)$

$$= c_1n + c_2(n - 1) + c_3(n - 1) + c_4 \sum_{j=1}^{n-1} t_j + c_5 \sum_{j=1}^{n-1} (t_j - 1) + c_6 \sum_{j=1}^{n-1} (t_j - 1) + c_7(n - 1)$$

Mejor

$t_j = 1; j = 2, 3, \dots, n$

$$\begin{aligned} T(n) &= c_1n + c_2(n - 1) + c_3(n - 1) + c_4(n - 1) + c_7(n - 1) \\ &= (c_1 + c_2 + c_3 + c_4 + c_7)n + (c_2 + c_3 + c_4 + c_7) \end{aligned}$$

$an + b$ Función lineal de n

Pior

$t_j = j; j = 2, 3, \dots, n$

$$\begin{aligned} T(n) &= c_1n + c_2(n - 1) + c_4(n - 1) + c_5\left(\frac{n(n+1)}{2} - 1\right) + c_6\left(\frac{n(n-1)}{2}\right) + c_7\left(\frac{n(n-1)}{2}\right) + c_8(n - 1) \\ &= \left(\frac{c_5}{2} + \frac{c_6}{2} + \frac{c_7}{2}\right)n^2 + \left(c_1 + c_2 + c_4 + \frac{c_5}{2} - \frac{c_6}{2} - \frac{c_7}{2} + c_8\right)n - (c_2 + c_4 + c_5 + c_8) \end{aligned}$$

$an^2 + bn + c$ Función cuadrática de n

Agenda

1 Problema de ordenamiento

Formulación

Diseño

Análisis

Código

2 Diseño

3 Análisis

4 Aspectos finales

Ejercicios

Código

```
def insertion_sort(A):
    for j in range(2, len(A)):
        key = A[j]
        i = j - 1

        while i > 0 and A[i] > key:
            A[i + 1] = A[i]
            i = i - 1

        A[i + 1] = key
```

Agenda

1 Problema de ordenamiento

Formulación

Diseño

Análisis

Código

2 Diseño

3 Análisis

4 Aspectos finales

Ejercicios

Invariante



- ▶ Los invariantes del ciclo ayudan a probar que un algoritmo es correcto. Se deben demostrar tres cosas sobre un invariante de ciclo:
 - ▶ Iniciación: El invariante es verdadero antes de la primera iteración.
 - ▶ Estabilidad: Si es verdadero antes de una iteración del ciclo, debe ser verdadero después de la iteración.
 - ▶ Terminación: Cuando el ciclo termina, el invariante es una propiedad importante que ayuda a demostrar que el algoritmo correcto.

Agenda

1 Problema de ordenamiento

Formulación

Diseño

Análisis

Código

2 Diseño

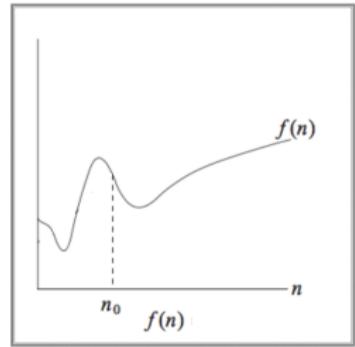
3 Análisis

4 Aspectos finales

Ejercicios

Análisis Asintótico

- Cuando se estudia la eficiencia asintótica de un algoritmo, estamos interesados en saber como el tiempo de ejecución aumenta con una entrada que se incrementa sin límite.
- El orden de crecimiento del tiempo de ejecución de un algoritmo nos da una caracterización sencilla de la eficiencia del algoritmo y nos permite comparar el desempeño relativo de varios algoritmos.
- La notación que se usa para describir el tiempo de ejecución asintótico de un algoritmo esta definido en términos de funciones cuyo dominio es el conjunto de números naturales $N = \{0, 1, 2, \dots\}$.

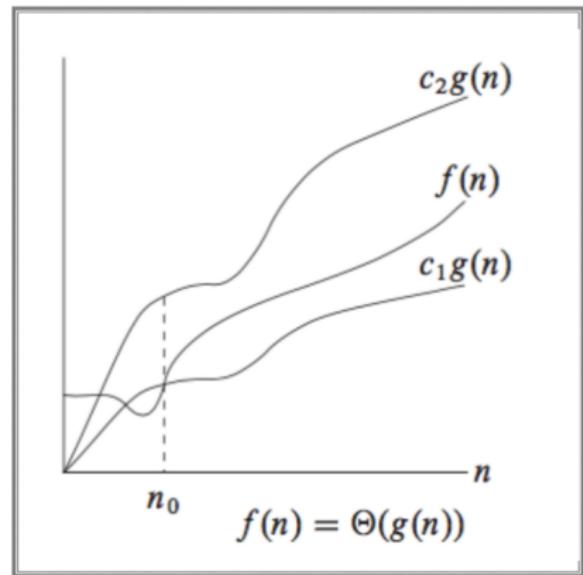


Notación Θ

$$\Theta(g(n)) =$$

$\{f(n) : \text{Existen constantes positivas } c_1, c_2 \text{ y } n_0$
tales que $0 \leq c_1g(n) \leq f(n) \leq c_2g(n)$
para todos los $n \geq n_0\}$

La notación Θ envuelve a la función por arriba
y por abajo.

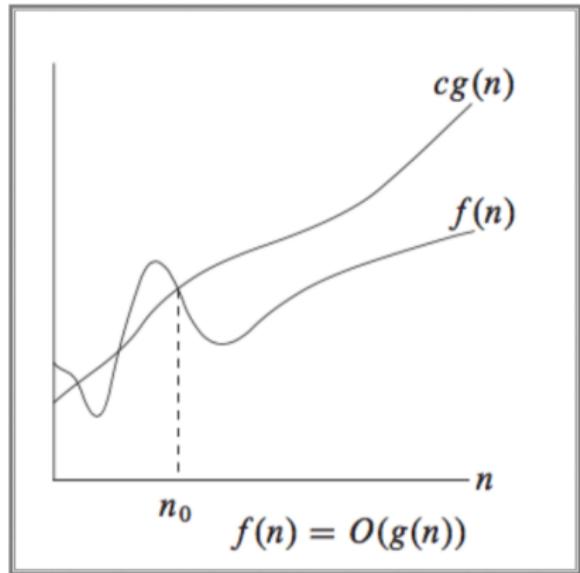


Notación O

$O(g(n)) =$

$\{f(n) : \text{Existen constantes positivas } c \text{ y } n_0$
 $\text{tales que } 0 \leq f(n) \leq cg(n)$
 $\text{para todos los } n \geq n_0\}$

Cuando solo se tienen un límite superior, se usa la notación-O.

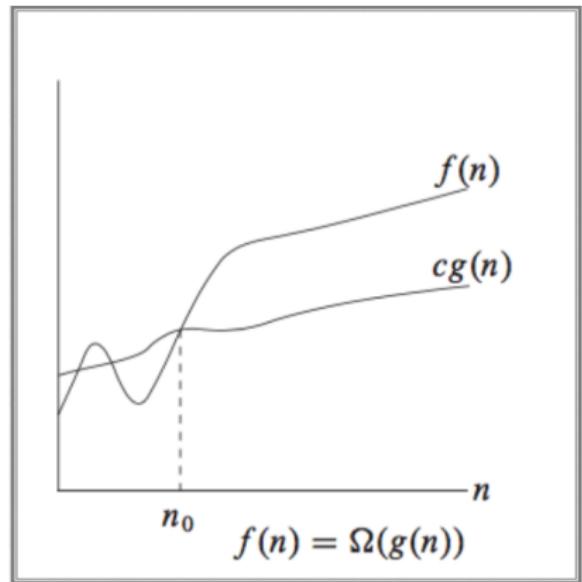


Notación Ω

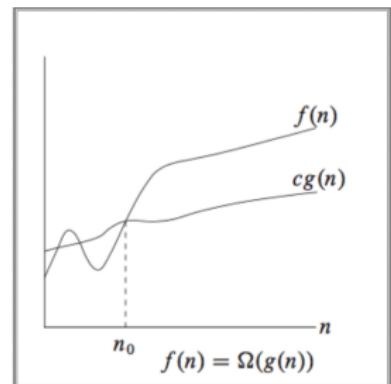
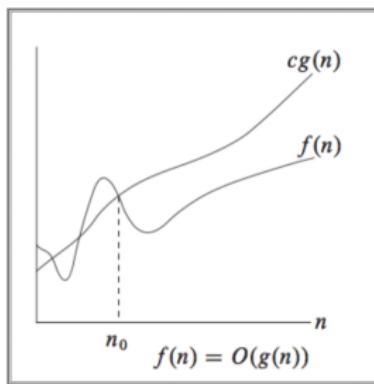
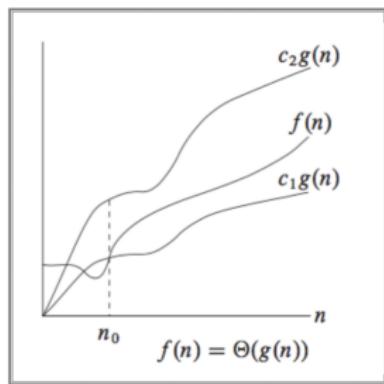
$\Omega(g(n)) =$

$\{f(n) : \text{Existen constantes positivas } c \text{ y } n_0$
tales que $0 \leq cg(n) \leq f(n)$
para todos los $n \geq n_0\}$

Esta notación provee un límite inferior asintótico para la función.



Notaciones Θ O Ω



Para dos funciones $f(n)$ y $g(n)$, se tiene que

$$f(n) = \Theta(g(n))$$

si y solo si

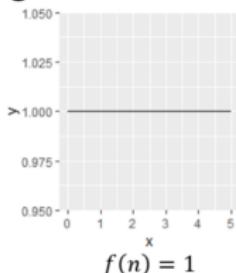
$$f(n) = O(g(n))$$

y

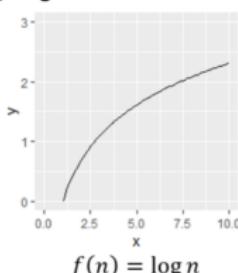
$$f(n) = \Omega(g(n))$$

Orden de crecimiento

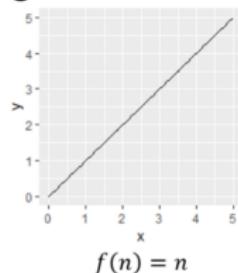
1 Constante



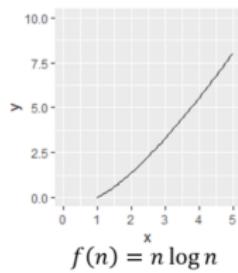
2 Logarítmico



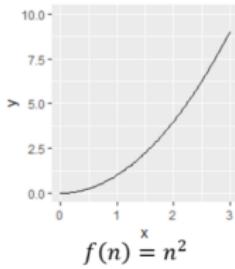
3 Lineal



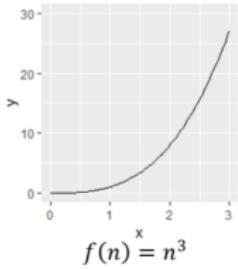
4 Superlineal



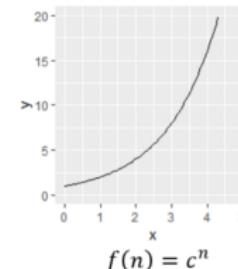
5 Cuadrático



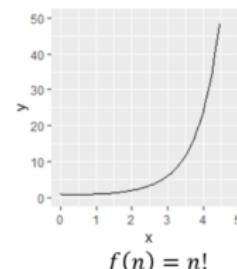
6 Cúbico



7 Exponencial



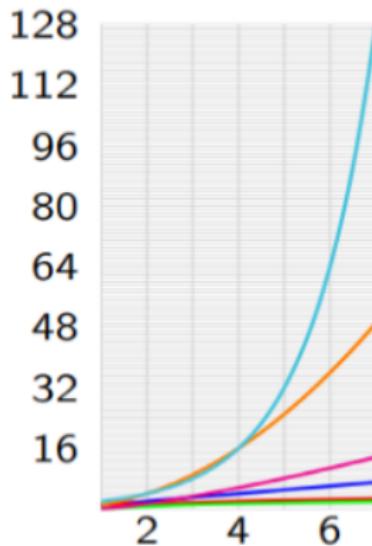
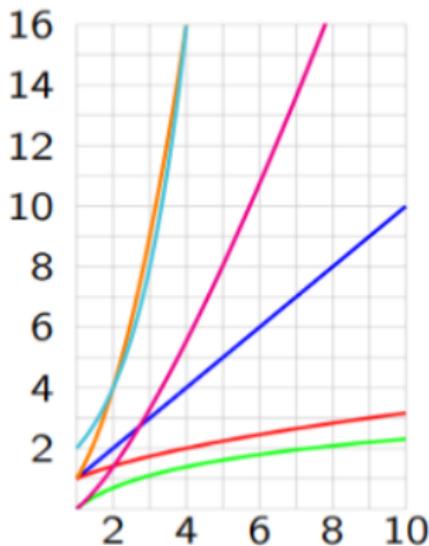
8 Factorial



$$n! \gg 2^n \gg n^3 \gg n^2 \gg n \log n \gg n \gg \log n \gg 1$$

Orden de crecimiento

$$\log n \prec \sqrt{n} \prec n \prec n \log n \prec n^2 \prec 2^n$$



Análisis comparativo

n	$\log n$	n	$n \log n$	n^2	n^3	2^n
8	3	8	24	64	512	256
16	4	16	64	256	4,096	65,536
32	5	32	160	1,024	32,768	4,294,967,296
64	6	64	384	4,096	262,144	1.84×10^{19}
128	7	128	896	16,384	2,097,152	3.40×10^{38}
256	8	256	2,048	65,536	16,777,216	1.15×10^{77}
512	9	512	4,608	262,144	134,217,728	1.34×10^{154}

Running Time (μs)	Maximum Problem Size (n)		
	1 second	1 minute	1 hour
$400n$	2,500	150,000	9,000,000
$2n^2$	707	5,477	42,426
2^n	19	25	31

Agenda

1 Problema de ordenamiento

Formulación

Diseño

Análisis

Código

2 Diseño

3 Análisis

4 Aspectos finales

Ejercicios

Ejercicios

1. Ordenar las siguientes funciones de menor a mayor orden:

$$1. n$$

$$2. n - n^3 + 7n^5$$

$$3. n^2 + \log n$$

$$4. n^3$$

$$5. 2^n$$

$$6. \log n$$

$$7. n^2$$

$$8. (\log n)^2$$

$$9. n \log n$$

$$10. \sqrt{n}$$

$$11. 2^{n-1}$$

$$12. n!$$

$$13. \ln n$$

$$14. e^n$$

$$15. \log \log n$$

$$16. n^{1+\varepsilon}, 0 < \varepsilon < 1$$

2. Para las siguientes funciones, determinar el resultado como una función de n y representar el peor caso de ejecución con notación Big Oh:

```
function mystery(n)
    r := 0
    for i := 1 to n - 1 do
        for j := i + 1 to n do
            for k := 1 to j do
                r := r + 1
    return(r)
```

```
function pesky(n)
    r := 0
    for i := 1 to n do
        for j := 1 to i do
            for k := j to i + j do
                r := r + 1
    return(r)
```

```
function prestiforous(n)
    r := 0
    for i := 1 to n do
        for j := 1 to i do
            for k := j to i + j do
                for l := 1 to i + j - k do
                    r := r + 1
    return(r)
```



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

$$\sum_{j=i+1}^n j = \sum_{j=1}^n j - \sum_{j=1}^i j$$

$$\sum_{x=1}^n x^2 = \frac{1}{6} n(n+1)(2n+1)$$

3. Implementar el algoritmo de *insertion sort* para ordenar en orden descendente en vez de ascendente.