Estrategia incremental y notación asintótica

Alejandro Anzola Ávila

2024-2

Algoritmos y Estructuras de Datos – Grupo 4

Quiz

1. Dada una secuencia de entrada de n números $\langle a_1, a_2, \ldots, a_n \rangle$, un algoritmo de ordenamiento produce …
2. Describa las tres características que debe cumplir un invariante de ciclo
   - Inicialización (*Initialization*)
   - Estabilidad (*Maintenance*)
   - Terminación (*Termination*)
3. Describa con sus palabras que es la notación asintótica, y describa Θ, O, Ω.

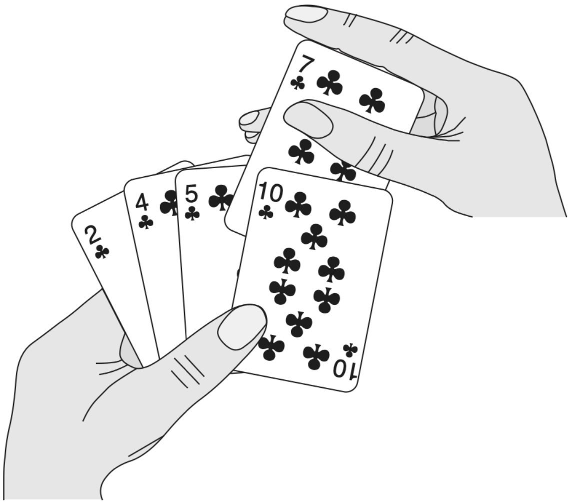
Problema de ordenamiento

Formulación

Ordenamiento

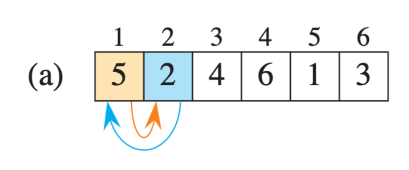
Entrada Una secuencia de n números $\langle a_1, a_2, \ldots, a_n \rangle$

Salida Una permutación $\langle a'_1, a'_2, \ldots, a'_n \rangle$, tal que $a'_1 \leq a'_2 \leq \cdots \leq a'_n$



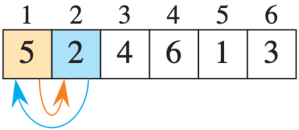
Similar a como una persona organiza una mano de cartas.

¿Qué condición se cumple siempre?

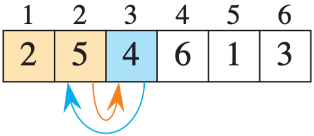


¿Qué condición se cumple siempre?

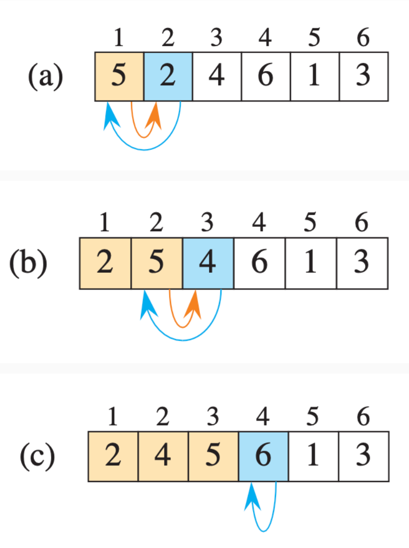
(a)



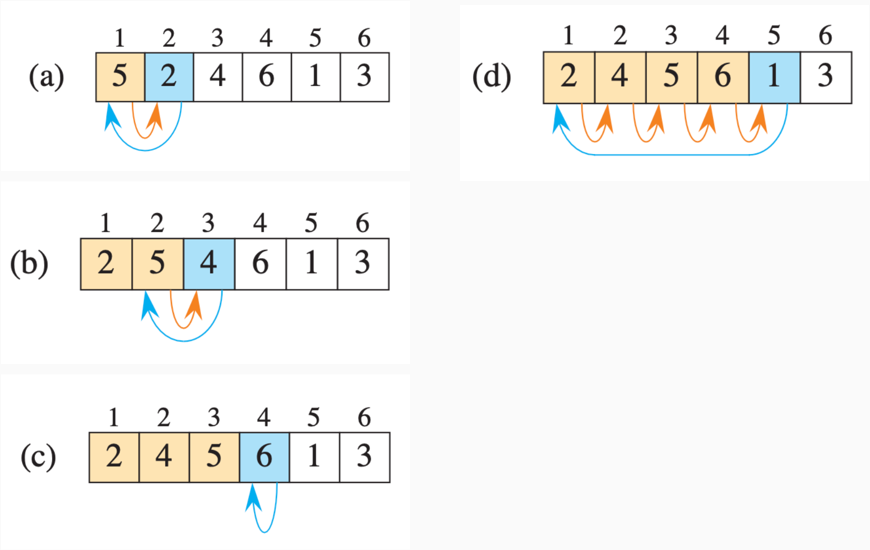
(b)



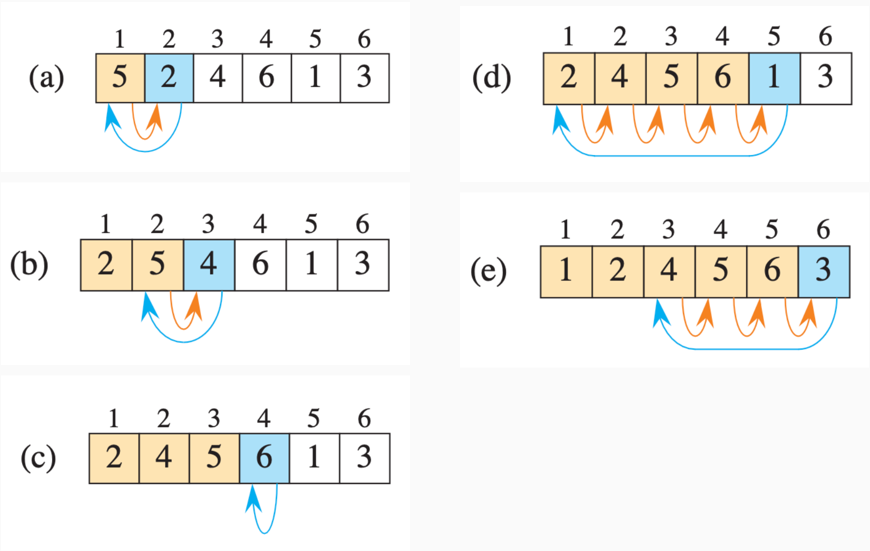
¿Qué condición se cumple siempre?



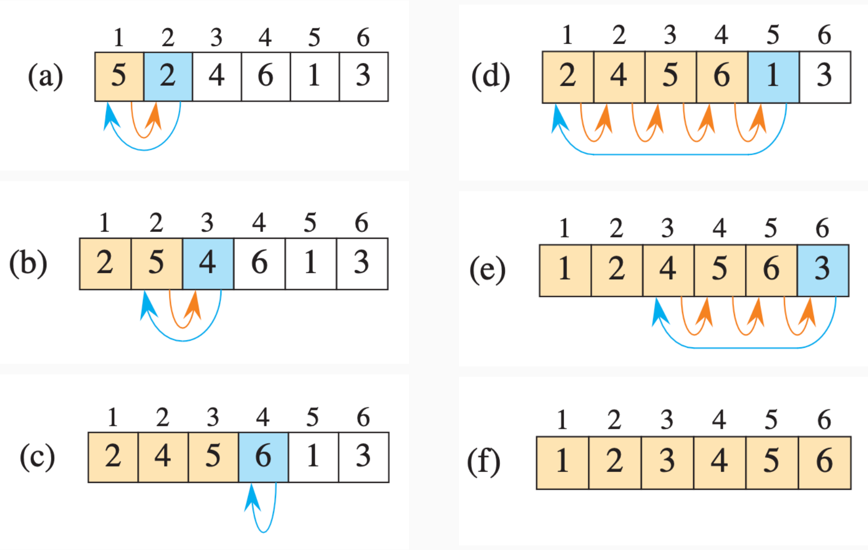
¿Qué condición se cumple siempre?



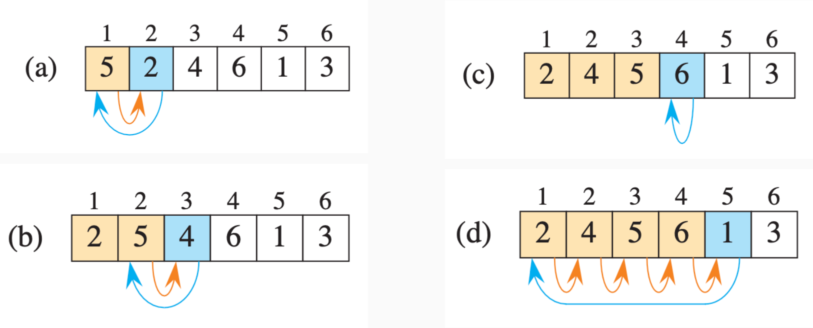
¿Qué condición se cumple siempre?



¿Qué condición se cumple siempre?

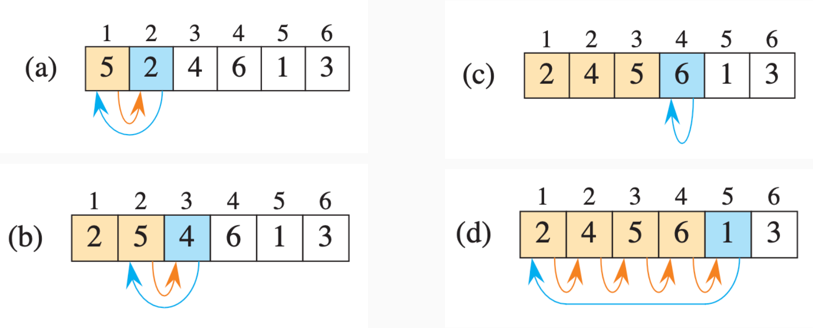


Invariante



Los elementos antes del **actual**:

- Son los originales
- Pero ahora están ordenados



Los elementos antes del **actual**:

- Son los originales
- Pero ahora están ordenados

Este es el invariante de ciclo.

Características de un invariante

Los *invariantes de ciclo* nos ayudan a demostrar que un algoritmo es correcto.

Se deben demostrar *tres* cosas sobre un invariante de ciclo:

1. **Iniciación**: Es verdadero antes de la primera iteración.
2. **Estabilidad**: Es verdadero
   - Antes de una iteración del ciclo
   - Antes de empezar el siguiente ciclo.
3. **Terminación**: El ciclo termina, el invariante nos da una propiedad útil para demostrar que el algoritmo es *correcto*.

```
INSERTION-SORT(A, n)
for i = 2 to n
    key = A[i]
    // Insert A[i] into the sorted subarray A[1 : i − 1].
    j = i − 1
    while j > 0 and A[j] > key
        A[j + 1] = A[j]
        j = j − 1
    A[j + 1] = key
```

i: Posición actual

j:

```
INSERTION-SORT(A, n)
for i = 2 to n
    key = A[i]
    // Insert A[i] into the sorted subarray A[1 : i - 1].
    j = i - 1
    while j > 0 and A[j] > key
        A[j + 1] = A[j]
        j = j - 1
    A[j + 1] = key
```

i: Posición actual

j: Posición del elemento a comparar y ordenar

Notación de Cormen

Los arreglos empiezan por 1, en Python por 0.

aka 1-origin indexing ; 0-origin indexing

Iniciación

```
INSERTION-SORT(A, n)
for i = 2 to n
    key = A[i]
    // Insert A[i] into the sorted subarray A[1 : i − 1].
    j = i − 1
    while j > 0 and A[j] > key
        A[j + 1] = A[j]
        j = j − 1
    A[j + 1] = key
```

Invariante

P_0:

Iniciación

```
INSERTION-SORT(A, n)
for i = 2 to n
    key = A[i]
    // Insert A[i] into the sorted subarray A[1 : i − 1].
    j = i − 1
    while j > 0 and A[j] > key
        A[j + 1] = A[j]
        j = j − 1
    A[j + 1] = key
```

Invariante

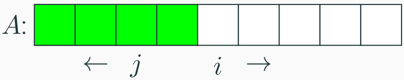
P_0: El subarreglo $A[1 : i - 1]$ esta ordenado

Estabilidad

```
INSERTION-SORT(A, n)
for i = 2 to n
    key = A[i]
    // Insert A[i] into the sorted subarray A[1 : i − 1].
    j = i − 1
    while j > 0 and A[j] > key
        A[j + 1] = A[j]
        j = j − 1
    A[j + 1] = key
```

Invariante

P_0: El subarreglo $A[1 : i - 1]$ esta ordenado

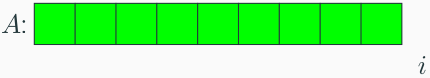


Terminación

```
INSERTION-SORT(A, n)
for i = 2 to n
    key = A[i]
    // Insert A[i] into the sorted subarray A[1 : i − 1].
    j = i − 1
    while j > 0 and A[j] > key
        A[j + 1] = A[j]
        j = j − 1
    A[j + 1] = key
```

Invariante

P_0: El subarreglo $A[1:i-1]$ esta ordenado



Estrategia incremental

Consiste en solucionar un problema de manera progresiva.

> *"INSERTION SORT usa el método incremental: para cada elemento $A[i]$, insertarlo en su lugar apropiado en el subarreglo $A[1:i]$, teniendo el subarreglo $A[1:i-1]$ ya ordenado."*
>
> *— Traducido de Cormen*

Análisis

Recursos

En algoritmos, queremos saber como se hace uso de los recursos.

Recursos

En algoritmos, queremos saber como se hace uso de los recursos.

Usualmente nos interesan:

- Tiempo de ejecución
- Espacio utilizado

Tiempo de ejecución

Usualmente estamos interesados en el tiempo de ejecución T.
Este en función del tamaño de nuestra entrada n.

$$T(n)$$

Tiempo de ejecución de INSERTION SORT

INSERTION-SORT(A, n)	*cost*	*times*
for $i = 2$ **to** n	c_1	n
$\quad key = A[i]$	c_2	$n - 1$
$\quad$ // Insert $A[i]$ into the sorted subarray $A[1:i-1]$.	0	$n - 1$
$\quad j = i - 1$	c_4	$n - 1$
$\quad$ **while** $j > 0$ and $A[j] > key$	c_5	$\sum_{i=2}^{n} t_i$
$\qquad A[j+1] = A[j]$	c_6	$\sum_{i=2}^{n} (t_i - 1)$
$\qquad j = j - 1$	c_7	$\sum_{i=2}^{n} (t_i - 1)$
$\quad A[j+1] = key$	c_8	$n - 1$

Tiempo de ejecución de INSERTION SORT

INSERTION-SORT(A, n)	*cost*	*times*
for $i = 2$ **to** n	c_1	n
$\quad key = A[i]$	c_2	$n-1$
$\quad$ // Insert $A[i]$ into the sorted subarray $A[1:i-1]$.	0	$n-1$
$\quad j = i - 1$	c_4	$n-1$
$\quad$ **while** $j > 0$ and $A[j] > key$	c_5	$\sum_{i=2}^{n} t_i$
$\quad\quad A[j+1] = A[j]$	c_6	$\sum_{i=2}^{n} (t_i - 1)$
$\quad\quad j = j - 1$	c_7	$\sum_{i=2}^{n} (t_i - 1)$
$\quad A[j+1] = key$	c_8	$n-1$

$$
\begin{aligned}
T(n) = & c_1 n + c_2(n-1) + c_4(n-1) + c_5 \sum_{i=2}^{n} t_i + c_6 \sum_{i=2}^{n} (t_i - 1) \\
& + c_7 \sum_{i=2}^{n} (t_i - 1) + c_8(n-1)
\end{aligned}
$$

Mejor caso

$t_i = 1$; $i = 2, 3, 4, \ldots$

$$
\begin{aligned}
T(n) =& c_1 n + c_2(n-1) + c_4(n-1) + c_5 \sum_{i=2}^{n} t_i + c_6 \sum_{i=2}^{n} (t_i - 1) \\
& + c_7 \sum_{i=2}^{n} (t_i - 1) + c_8(n-1) \\
=& \\
=& \\
=&
\end{aligned}
$$

Mejor caso

$t_i = 1 \; ; i = 2, 3, 4, \ldots$

$$\begin{aligned} T(n) =& c_1 n + c_2(n-1) + c_4(n-1) + c_5 \sum_{i=2}^{n} t_i + c_6 \sum_{i=2}^{n} (t_i - 1) \\ & + c_7 \sum_{i=2}^{n} (t_i - 1) + c_8(n-1) \\ =& c_1 n + c_2(n-1) + c_4(n-1) + c_5(n-1) + c_8(n-1) \\ =& \\ =& \end{aligned}$$

Mejor caso

$t_i = 1 \; ; i = 2, 3, 4, \ldots$

$$
\begin{aligned}
T(n) =& c_1 n + c_2(n-1) + c_4(n-1) + c_5 \sum_{i=2}^{n} t_i + c_6 \sum_{i=2}^{n} (t_i - 1) \\
& + c_7 \sum_{i=2}^{n} (t_i - 1) + c_8(n-1) \\
=& c_1 n + c_2(n-1) + c_4(n-1) + c_5(n-1) + c_8(n-1) \\
=& (c_1 + c_2 + c_4 + c_5 + c_8) n - (c_2 + c_4 + c_5 + c_8) \\
=& an - b
\end{aligned}
$$

Peor caso

$t_i = i \; ; i = 2, 3, 4, \ldots$

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

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

$$\sum_{i=2}^{n} (i-1) = \sum_{i=1}^{n-1} i = \frac{n(n-1)}{2}$$

Peor caso

$t_i = i \; ; i = 2, 3, 4, \ldots$

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

$$\sum_{i=2}^{n} i = \sum_{i=1}^{n} i - 1 = \frac{n(n+1)}{2} - 1 \qquad \sum_{i=2}^{n} (i-1) = \sum_{i=1}^{n-1} i = \frac{n(n-1)}{2}$$

Mejor caso

$t_i = 1\ ;\ i = 2, 3, 4, \ldots$

$$T(n) = an - b$$

Peor caso

$t_i = i\ ;\ i = 2, 3, 4, \ldots$

$$T(n) = an^2 + bn - c$$

La formula de tiempo de ejecución $T(n)$ de INSERTION SORT:

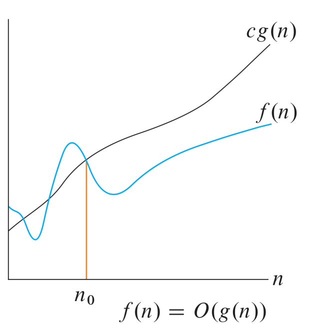
$$T(n) = an^2 + bn - c$$

- Es compleja
- Comparar con otro algoritmo es difícil

Nos interesa una forma sencilla y versátil de distinguir si un algoritmo es mejor que otro.

Notación O (Big Oh)

Nos especifica un limite superior de una función $f(n)$.



$$O(g(n)) = \{f(n) : \text{existe una constante positiva } c \text{ y } n_0 \text{ tal que } 0 \leq f(n) \leq cg(n) \text{ para cualquier } n \geq n_0\}$$

$$O(g(n)) = \{f(n) : \text{existe una constante positiva } c \text{ y } n_0 \text{ tal que}$$
$$0 \leq f(n) \leq cg(n) \text{ para cualquier } n \geq n_0\}$$

Según esta definición se puede decir que para un valor de n suficientemente grande:

$f(n)$	$\in O(g(n))$	Si/No	c
n	$\in O(n)$		

$$O(g(n)) = \{f(n) : \text{existe una constante positiva } c \text{ y } n_0 \text{ tal que}$$
$$0 \leq f(n) \leq cg(n) \text{ para cualquier } n \geq n_0\}$$

Según esta definición se puede decir que para un valor de n suficientemente grande:

$f(n)$	$\in O(g(n))$	Si/No	c
n	$\in O(n)$	Si	$c \geq 1$
n^2	$\in O(n)$		

$$O(g(n)) = \{f(n) : \text{existe una constante positiva } c \text{ y } n_0 \text{ tal que}$$
$$0 \leq f(n) \leq cg(n) \text{ para cualquier } n \geq n_0\}$$

Según esta definición se puede decir que para un valor de n suficientemente grande:

$f(n)$	$\in O(g(n))$	Si/No	c
n	$\in O(n)$	Si	$c \geq 1$
n^2	$\in O(n)$	No	No existe c que cumpla esta condición
n	$\in O(n^2)$		

$$O(g(n)) = \{f(n) : \text{existe una constante positiva } c \text{ y } n_0 \text{ tal que}$$
$$0 \leq f(n) \leq cg(n) \text{ para cualquier } n \geq n_0\}$$

Según esta definición se puede decir que para un valor de n suficientemente grande:

$f(n)$	$\in O(g(n))$	Si/No	c
n	$\in O(n)$	Si	$c \geq 1$
n^2	$\in O(n)$	No	No existe c que cumpla esta condición
n	$\in O(n^2)$	Si	$c \geq 1$

Todos estos casos tienen $n_0 = 1$.

Abuso de notación i

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

$$\Updownarrow$$

$$f(n) \in O(g(n))$$

Abuso de notación ii

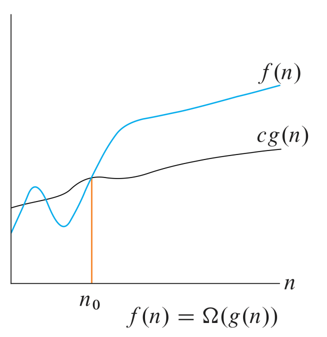
$$2n^2 + O(n)$$
$$\Updownarrow$$
$$2n^2 + f(n),\, f(n) \in O(n)$$

Big Oh para Insertion Sort

$$\begin{aligned} T(n) &= an^2 + bn + c \\ &= O(an^2) + O(bn) + O(c) \\ &= O(n^2) + O(n) + O(1) \\ &= O(n^2) \end{aligned}$$

Notación Ω

Nos especifica un limite inferior de una función $f(n)$.

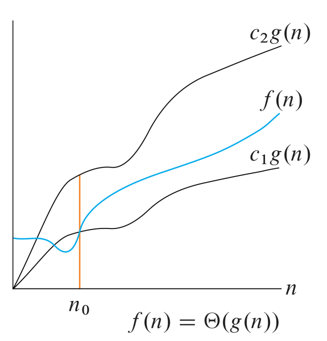


$\Omega(g(n)) = \{f(n) :$ existe una constante positiva c y n_0 tal que

$0 \leq cg(n) \leq f(n)$ para cualquier $n \geq n_0\}$

Notación Θ

Nos especifica un limite estricto de una función $f(n)$.



$$\Theta(g(n)) = \{f(n) : \text{existen constantes positivas } c_1, c_2 \text{ y } n_0 \text{ tales que}$$
$$0 \leq c_1 g(n) \leq f(n) \leq c_2 g(n) \text{ para cualquier } n \geq n_0\}$$

Tasas de crecimiento

n $f(n)$	$\lg n$	n	$n \lg n$	n^2	2^n	$n!$
10	0.003 μs	0.01 μs	0.033 μs	0.1 μs	1 μs	3.63 ms
20	0.004 μs	0.02 μs	0.086 μs	0.4 μs	1 ms	77.1 years
30	0.005 μs	0.03 μs	0.147 μs	0.9 μs	1 sec	8.4×10^{15} yrs
40	0.005 μs	0.04 μs	0.213 μs	1.6 μs	18.3 min	
50	0.006 μs	0.05 μs	0.282 μs	2.5 μs	13 days	
100	0.007 μs	0.1 μs	0.644 μs	10 μs	4×10^{13} yrs	
1,000	0.010 μs	1.00 μs	9.966 μs	1 ms		
10,000	0.013 μs	10 μs	130 μs	100 ms		
100,000	0.017 μs	0.10 ms	1.67 ms	10 sec		
1,000,000	0.020 μs	1 ms	19.93 ms	16.7 min		
10,000,000	0.023 μs	0.01 sec	0.23 sec	1.16 days		
100,000,000	0.027 μs	0.10 sec	2.66 sec	115.7 days		
1,000,000,000	0.030 μs	1 sec	29.90 sec	31.7 years		

Ejercicios

Ejercicios

1. Ordenar las siguientes funciones de menor a mayor orden
   a. n
   b. $n - n^3 + 7n^5$
   c. $n^2 + \log n$
   d. n^3
   e. 2^n
   f. $\log n$
   g. n^2
   h. $(\log n)^2$
   i. $n \log n$
   j. $\sqrt{n}$
   k. 2^{n-1}
   l. $n!$
   m. $\ln n$
   n. e^n
   ñ. $\log \log n$
   o. $n^{1+\epsilon}, 0 < \epsilon < 1$
2. Establezca una invariante de ciclo, y use sus propiedades de iniciación, estabilidad y terminación para mostrar que el siguiente algoritmo retorna la suma de los n números de $A[1 : n]$.

```
SUM-ARRAY(A, n)
sum = 0
for i = 1 to n
    sum = sum + A[i]
return sum
```

3. Implementar el algoritmo de INSERTION SORT para ordenar en orden descendente en vez de ascendente y establezca una nueva invariante.
4. ¿Es $2^{n+1} = O(2^n)$? ¿Es $2^{2n} = O(2^n)$?
5. ¿Es $2^{n+1} = \Omega(2^n)$? ¿Es $2^{n+1} = \Theta(2^n)$?

Refuerzo: Recursión
Modele una función f recursiva que

1. Computa la suma $f(n) = 1 + 2 + \cdots + n$
2. Hallé el elemento mínimo en la secuencia $\langle a_1, \ldots, a_n \rangle$
3. Halle la suma de la secuencia $\langle a_1, \ldots, a_n \rangle$
4. Determine si una secuencia de elementos es un palíndromo, de serlo debe retornar 1, de lo contrario 0.
5. Determine cuales son los divisores de un número. La salida sería una lista.
6. Halle la suma de los divisores de un número n.
7. $\star$ Determine si un número n es perfecto. Retorne 1 si lo es, de lo contrario 0. Un número perfecto es un número que es igual a la suma de sus divisores. Los primeros tres números perfectos son:
   $$6 = 1 + 2 + 3$$
   $$28 = 1 + 2 + 4 + 7 + 14$$
   $$496 = 1 + 2 + 4 + 8 + 16 + 31 + 62 + 124 + 248$$