

Algoritmos de ordenamiento

- Complejidad temporal
- Ordenamiento por selección, burbujeo e inserción

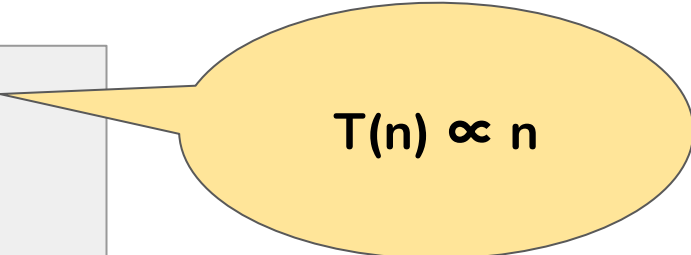


Complejidad temporal

Recapitulando ...

Búsqueda lineal (con corte)

Para todo **i** entre 0 y n (no inclusive):
si **lista[i]** es igual a **target** entonces:
 devolver **i**
si **lista[i]** es mayor a **target** entonces:
 devolver **-1**
devolver **-1**


$$T(n) \propto n$$

Recapitulando ...

Búsqueda binaria

izquierda $\leftarrow 0$

derecha $\leftarrow n - 1$

mientras izquierda **sea menor o igual a** derecha:

 medio $\leftarrow \lfloor (\text{izquierda} + \text{derecha}) / 2 \rfloor$

si lista[medio] **es igual a** target:

devolver medio

si target **es mayor a** lista[medio]:

 izquierda $\leftarrow \text{medio} + 1$

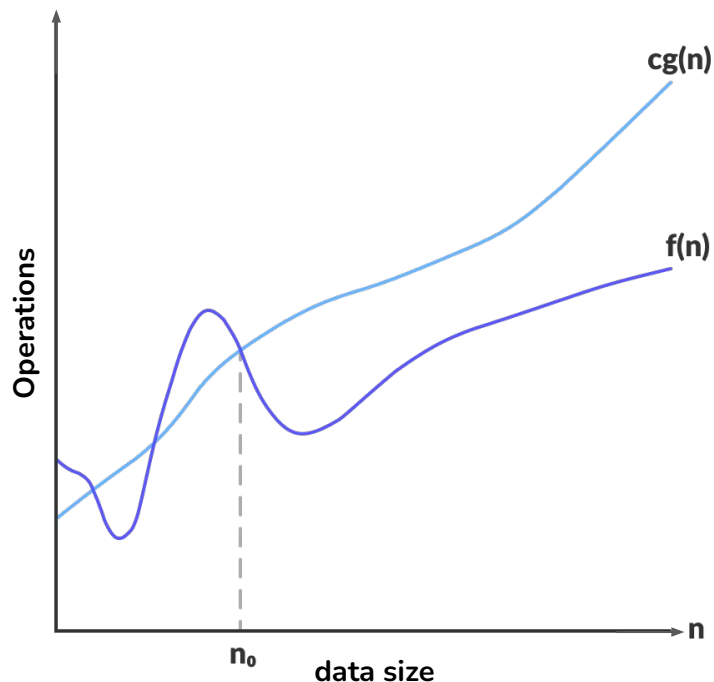
sino:

 derecha $\leftarrow \text{medio} - 1$

devolver -1


$$T(n) \propto \log n$$

Big-O Notation



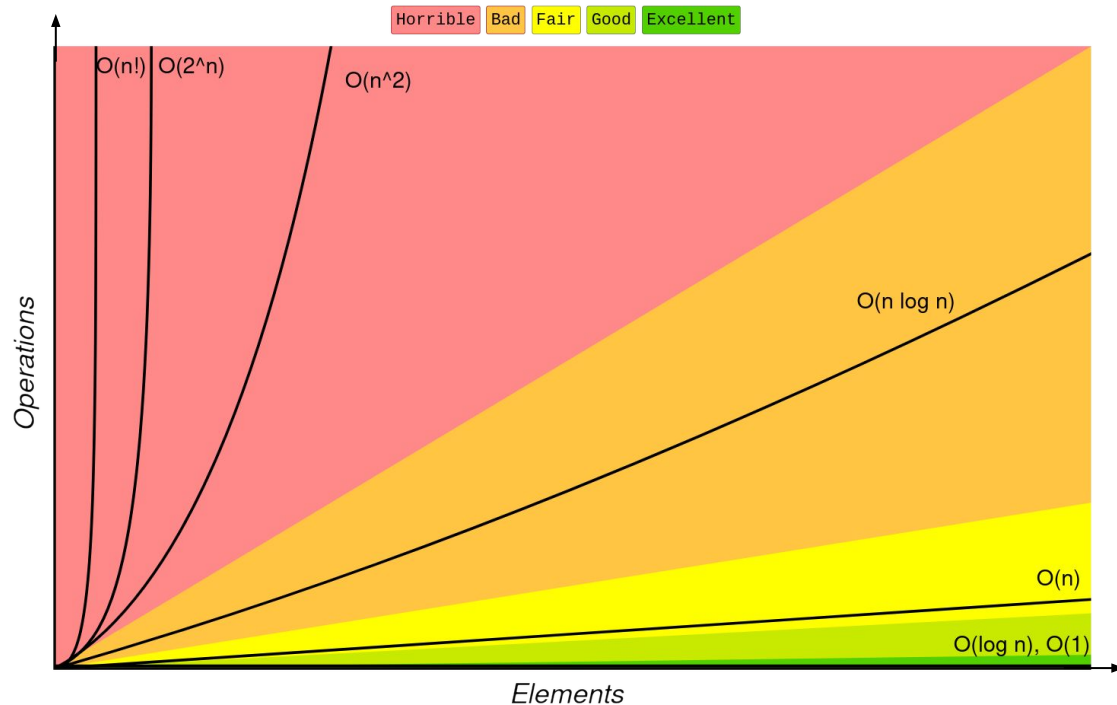
- Nuestro **$T(n)$** es **$f(n)$**
- Si existe un **n_0** a partir del cual **$f(n)$** siempre es **menor** (o igual) a **$c \cdot g(n)$** , se dice que:
 $f(n)$ pertenece al conjunto $O(g(n))$

$$T(n) \in O(g(n)) \Leftrightarrow \exists n_0 / T(n) < c \cdot g(n) \quad \forall n > n_0$$

Big-O Notation

Principales órdenes de complejidad

Orden	Nombre
$O(1)$	constante
$O(\log n)$	logarítmica
$O(n)$	lineal
$O(n \log n)$	casi lineal
$O(n^2)$	cuadrática
$O(n^3)$	cúbica
$O(a^n)$	exponencial
$O(n!)$	factorial



$$O(n!) > O(2^n) > O(n^3) > O(n^2) > O(n \log n) > O(n) > O(\log n) > O(1)$$

Propiedades:

Sean $T_1(n) \in O(g_1(n))$ y $T_2(n) \in O(g_2(n))$,

a y b constantes:

- i. $a \in O(1)$
- ii. $T_1(n) + b \in O(g_1(n))$
- iii. $aT_1(n) \in O(g_1(n))$
- iv. $T_1(n) + T_2(n) \in O(\max[g_1(n), g_2(n)])$
- v. $T_1(n) T_2(n) \in O(g_1(n)g_2(n))$

Ejemplos:

Cantidad de operaciones

1. $n+5 \in O(n)$
2. $4n+1 \in O(n)$
3. $\log n + 20 \in O(\log n)$
4. $5 \log n + 8n \in O(n)$
5. $3n^3 + 2n^2 + 9 \in O(n^3)$
6. $2n \log n + n + 6000 \in O(n \log n)$
7. $(n^2+n)(n+1) \in O(n^2n) = O(n^3)$

¿Qué complejidad tiene cada ejemplo? (n tamaño de la entrada)

a)

```
if n % 2 == 0:
    print('par')
else:
    print('impar')
```

c)

```
for i in range(n):
    for j in range(n):
        if i % 2 == 0:
            print(i)
            print(j)
    print(i + j)
```

f)

```
i = 1
while i < n:
    i *= 2
    print(i)
```

b)

```
for i in range(n):
    if i % 2 == 0:
        j += 2
    print(i, j)
```

d)

```
for i in range(n):
    if i % 2 == 0:
        print(i)
    for i in range(n):
        for k in range(n):
            print(k)
```

g)

```
for i in range(n):
    j = 1
    while j < n:
        j *= 2
    print(j)
```


¿Qué complejidad tiene cada ejemplo? (n tamaño de la entrada)

a)

```
if n % 2 == 0:
    print('par')
else:
    print('impar')
```

c)

```
for i in range(n):
    for j in range(n):
        if i % 2 == 0:
            print(i)
            print(j)
    print(i + j)
```

f)

```
i = 1
while i < n:
    i *= 2
    print(i)
```

b)

```
for i in range(n):
    if i % 2 == 0:
        j += 2
    print(i, j)
```

d)

```
for i in range(n):
    if i % 2 == 0:
        print(i)
    for i in range(n):
        for k in range(n):
            print(k)
```

g)

```
for i in range(n):
    j = 1
    while j < n:
        j *= 2
    print(j)
```

?

¿Qué complejidad tiene cada ejemplo? (n tamaño de la entrada)

a)

```
if n % 2 == 0:
    print('par')
else:
    print('impar')
```

c)

```
for i in range(n):
    for j in range(n):
        if i % 2 == 0:
            print(i)
            print(j)
    print(i + j)
```

f)

```
i = 1
while i < n:
    i *= 2
    print(i)
```

b)

```
for i in range(n):
    if i % 2 == 0:
        j += 2
    print(i, j)
```

d)

```
for i in range(n):
    if i % 2 == 0:
        print(i)
    for i in range(n):
        for k in range(n):
            print(k)
```

g)

```
for i in range(n):
    j = 1
    while j < n:
        j *= 2
    print(j)
```

$O(1)$

¿Qué complejidad tiene cada ejemplo? (n tamaño de la entrada)

a)

```
if n % 2 == 0:
    print('par')
else:
    print('impar')
```

c)

```
for i in range(n):
    for j in range(n):
        if i % 2 == 0:
            print(i)
            print(j)
    print(i + j)
```

f)

```
i = 1
while i < n:
    i *= 2
    print(i)
```

b)

```
for i in range(n):
    if i % 2 == 0:
        j += 2
    print(i, j)
```

d)

```
for i in range(n):
    if i % 2 == 0:
        print(i)
    for j in range(n):
        for k in range(n):
            print(k)
```

g)

```
for i in range(n):
    j = 1
    while j < n:
        j *= 2
        print(j)
```

?

¿Qué complejidad tiene cada ejemplo? (n tamaño de la entrada)

a)

```
if n % 2 == 0:
    print('par')
else:
    print('impar')
```

c)

```
for i in range(n):
    for j in range(n):
        if i % 2 == 0:
            print(i)
            print(j)
    print(i + j)
```

f)

```
i = 1
while i < n:
    i *= 2
    print(i)
```

b)

```
for i in range(n):
    if i % 2 == 0:
        j += 2
    print(i, j)
```

d)

```
for i in range(n):
    if i % 2 == 0:
        print(i)
    for j in range(n):
        for k in range(n):
            print(k)
```

g)

```
for i in range(n):
    j = 1
    while j < n:
        j *= 2
        print(j)
```

$O(n)$

¿Qué complejidad tiene cada ejemplo? (n tamaño de la entrada)

a)

```
if n % 2 == 0:
    print('par')
else:
    print('impar')
```

c)

```
for i in range(n):
    for j in range(n):
        if i % 2 == 0:
            print(i)
            print(j)
    print(i + j)
```

f)

```
i = 1
while i < n:
```

?

b)

```
for i in range(n):
    if i % 2 == 0:
        j += 2
    print(i, j)
```

d)

```
for i in range(n):
    if i % 2 == 0:
        print(i)
    for i in range(n):
        for k in range(n):
            print(k)
```

g)

```
for i in range(n):
    j = 1
    while j < n:
        j *= 2
    print(j)
```

¿Qué complejidad tiene cada ejemplo? (n tamaño de la entrada)

a)

```
if n % 2 == 0:
    print('par')
else:
    print('impar')
```

c)

```
for i in range(n):
    for j in range(n):
        if i % 2 == 0:
            print(i)
            print(j)
    print(i + j)
```

f)

```
i = 1
while i < n:
```

$O(n^2)$

b)

```
for i in range(n):
    if i % 2 == 0:
        j += 2
    print(i, j)
```

d)

```
for i in range(n):
    if i % 2 == 0:
        print(i)
    for i in range(n):
        for k in range(n):
            print(k)
```

g)

```
for i in range(n):
    j = 1
    while j < n:
        j *= 2
    print(j)
```

¿Qué complejidad tiene cada ejemplo? (n tamaño de la entrada)

a)

```
if n % 2 == 0:
    print('par')
else:
    print('impar')
```

c)

```
for i in range(n):
    for j in range(n):
        if i % 2 == 0:
            print(i)
            print(j)
    print(i + j)
```

f)

```
i = 1
while i < n:
    i *= 2
    print(i)
```

b)

```
for i in range(n):
    if i % 2 == 0:
        j += 2
    print(i, j)
```

d)

```
for i in range(n):
    if i % 2 == 0:
        print(i)
    for i in range(n):
        for k in range(n):
            print(k)
```

g)

```
for i in range(n):
    j = 1
    while i < n:
        print(j)
```

?

¿Qué complejidad tiene cada ejemplo? (n tamaño de la entrada)

a)

```
if n % 2 == 0:
    print('par')
else:
    print('impar')
```

c)

```
for i in range(n):
    for j in range(n):
        if i % 2 == 0:
            print(i)
            print(j)
    print(i + j)
```

f)

```
i = 1
while i < n:
    i *= 2
    print(i)
```

b)

```
for i in range(n):
    if i % 2 == 0:
        j += 2
    print(i, j)
```

d)

```
for i in range(n):
    if i % 2 == 0:
        print(i)
    for i in range(n):
        for k in range(n):
            print(k)
```

g)

```
for i in range(n):
    j = 1
    while i < n:
        print(j)
```

$O(n^2)$

¿Qué complejidad tiene cada ejemplo? (n tamaño de la entrada)

a)

```
if n % 2 == 0:
    print('par')
else:
    print('impar')
```

c)

```
for i in range(n):
    for j in range(n):
```

f)

```
i = 1
while i < n:
    i *= 2
    print(i)
```

b)

```
for i in range(n):
    if i % 2 == 0:
        j += 2
    print(i, j)
```

g)

```
if i % 2 == 0:
    print(i)
for i in range(n):
    for k in range(n):
        print(k)
```

```
for i in range(n):
    j = 1
    while j < n:
        j *= 2
    print(j)
```

?

¿Qué complejidad tiene cada ejemplo? (n tamaño de la entrada)

a)

```
if n % 2 == 0:
    print('par')
else:
    print('impar')
```

c)

```
for i in range(n):
    for j in range(n):
```

f)

```
i = 1
while i < n:
    i *= 2
    print(i)
```

b)

```
for i in range(n):
    if i % 2 == 0:
        j += 2
    print(i, j)
```

g)

```
if i % 2 == 0:
    print(i)
for i in range(n):
    for k in range(n):
        print(k)
```

h)

```
for i in range(n):
    j = 1
    while j < n:
        j *= 2
    print(j)
```

Suponer $n=2^K$ potencia de 2

print(i): 2, 4, 8, 16 ..., $n = 2^1, 2^2, 2^3, \dots, 2^K$

Cantidad de iteraciones $K = \log_2(n)$

$O(\log(n))$

¿Qué complejidad tiene cada ejemplo? (n tamaño de la entrada)

a)

```
if n % 2 == 0:
    print('par')
else:
    print('impar')
```

c)

```
for i in range(n):
    for j in range(n):
        if i % 2 == 0:
            print(i)
            print(j)
    print(i + j)
```

f)

```
i = 1
while i < n:
    i *= 2
    print(i)
```

b)

```
for i in range(n):
    if i % 2 == 0:
        j += 2
    print(i, j)
```

?

for k in range(n):
 print(k)

g)

```
for i in range(n):
    j = 1
    while j < n:
        j *= 2
    print(j)
```

¿Qué complejidad tiene cada ejemplo?

a)

```
if n % 2 == 0:
    print('par')
else:
    print('impar')
```

c)

```
for i in range(n):
    for j in range(n):
        if i % 2 == 0:
            print(i)
            print(j)
    print(i + j)
```

f)

```
i = 1
while i < n:
    i *= 2
    print(i)
```

b)

```
for i in range(n):
    if i % 2 == 0:
        j += 2
    print(i, j)
```

Suponer $n=2^K$ potencia de 2

print(i): 2, 4, 8, 16, ..., $n = 2^1, 2^2, 2^3, \dots, 2^K$

Cantidad de iteraciones $n \cdot K = n \cdot \log_2(n)$

$O(n \log(n))$

```
for k in range(n):
    print(k)
```

e)

```
for i in range(n):
    j = 1
    while j < n:
        j *= 2
    print(j)
```



<https://wiki.python.org/moin/TimeComplexity>

Algoritmos de Ordenamiento

Existen muchas situaciones muy diversas en las que necesitamos implementar el ordenamiento de ciertas estructuras de datos

- Ordenar palabras alfabeticamente
- Ordenar equipos de acuerdo al puntaje
- Ordenar registros de acuerdo a la fecha
- Ordenar listas para realizar búsquedas
- Etc...



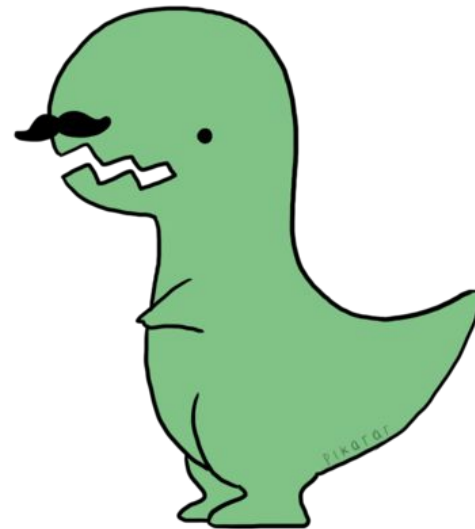
Volviendo a los algoritmos de búsqueda ...

Búsqueda lineal

- Recibe una lista de largo n
- Complejidad algorítmica **$O(n)$**

Búsqueda binaria

- Recibe una lista “ordenada” de largo n
- Complejidad algorítmica **$O(\log n)$**



Volviendo a los algoritmos de búsqueda ...

Búsqueda lineal

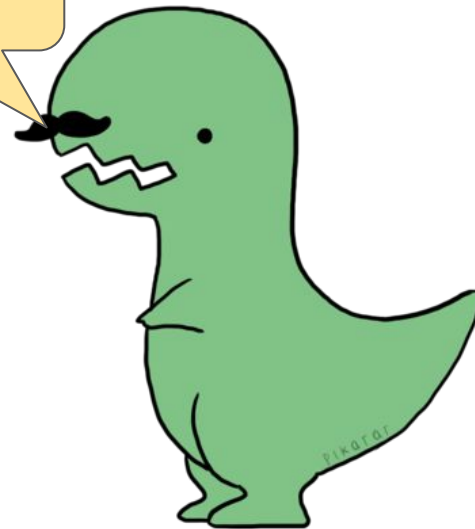
- Recibe una lista de largo n
- Complejidad algo

Se necesita una lista "ordenada"

Mejora apreciablemente la complejidad temporal

Búsqueda binaria

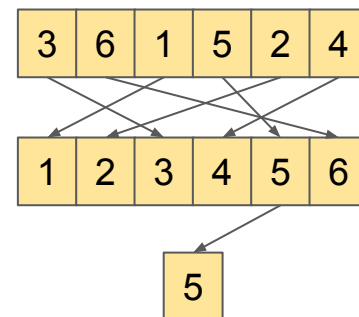
- Recibe una lista "ordenada" de largo n
- Complejidad algorítmica **$O(\log n)$**



Volviendo a los algoritmos de búsqueda ...

Si queremos una búsqueda binaria

- Primero hay que ordenar la lista: ¿ $O(\text{sort})$?
- Luego hay que buscar en la lista ordenada: $O(\log n)$
- Ordenamiento+búsqueda binaria: ¿ $O(\text{sort})$? + $O(\log n)$



Pero ...

- Para ordenar una lista hay que pasar al menos una vez por cada elemento!
- Por lo tanto, solo el ordenamiento tendrá complejidad $\geq O(n)!!$

Volviendo a los algoritmos de búsqueda ...

Busqueda lineal

- Complejidad $O(n)!!$

¿Cuándo conviene entonces hacer una búsqueda binaria?

Busqueda lineal

- Complejidad $O(n)!!$

¿Cuándo conviene entonces hacer una búsqueda binaria?

- Cuando necesitemos hacer más de una búsqueda sobre la misma lista
- La ordenamos una sola vez, luego buscamos cada vez que se necesite
- Mientras más búsquedas se haga sobre la misma lista, más se diluye el costo que agrega el ordenamiento

Ordenamiento Bogo

Ordenamiento Bogo

También conocido como Bogo Sort, Stupid Sort, Slow Sort, Permutation Sort, Shotgun Sort.

Ejemplo: ordenar una baraja de cartas

1. Tirar las cartas al aire
2. Levantarlas
3. ¿Están ordenadas?
4. Repetir si no están ordenadas



Ordenamiento Bogo

Algoritmo (pseudocódigo)

```
repetir ciclo:  
  revisar lista completa  
  si lista está ordenada:  
    retornar lista  
  sino:  
    mezclar lista
```



- Mejor caso (la lista ya estaba ordenada), solo requiere recorrer n elementos para verificar: $O(n)$
- Peor caso: nunca se comprueba una lista ordenada: complejidad indefinida

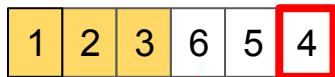
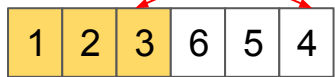
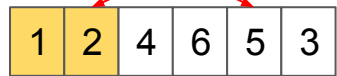
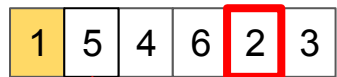
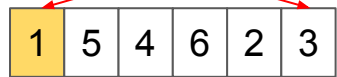
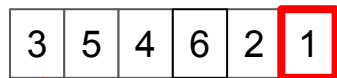
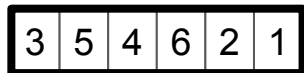
Ordenamiento por selección

Ordenamiento por selección

1. Buscar el mínimo elemento de la lista desordenada
2. Intercambiar mínimo con el primero de la lista desordenada
3. Definir nueva lista excluyendo elementos ya ordenados
4. Repetir procedimiento hasta que la lista completa esté ordenada

Ordenamiento por selección

Lista original
desordenada



a

Mínimo elemento de la
sublista no ordenada



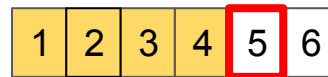
Intercambio de elementos
entre el mínimo y el
primer elemento de la
sublista no ordenada



Queda en el
mismo lugar

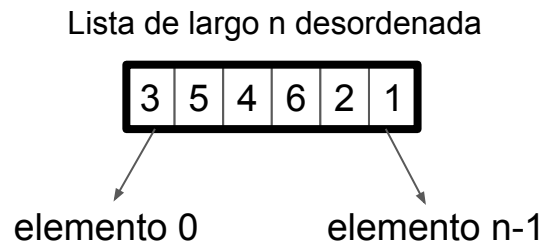
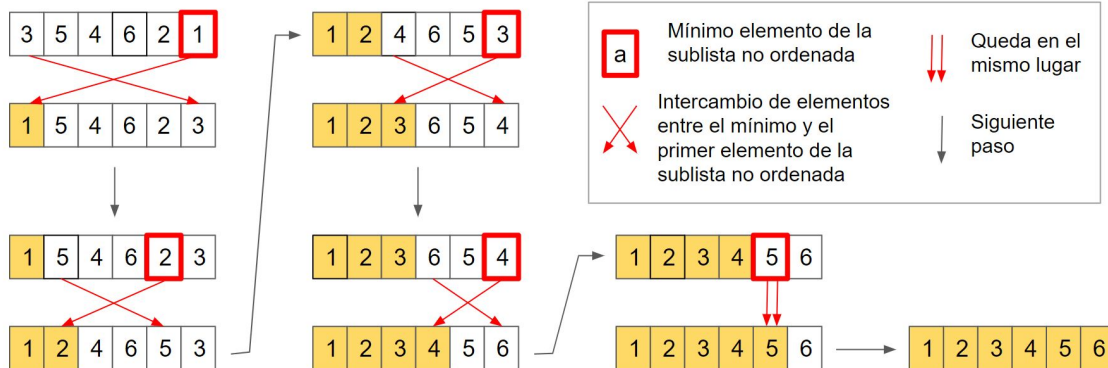


Siguiente
paso



Ordenamiento por selección

para j entre $[0, n-2]$
para k entre $[j, n-1]$
*chequear y guardar **mínimo** de sublista*
***intercambiar** mínimo con inicio lista sin ordenar*



Ordenamiento por selección

Lista de largo n

para j entre $[0, n-2]$

para k entre $[j, n-1]$

chequear y guardar **mínimo** de sublista

intercambiar mínimo con inicio lista sin ordenar

Primer ciclo: $n-1$ iteraciones
 $j = 0, 1, 2, \dots, n-2$

Ordenamiento por selección

Lista de largo n

para j entre $[0, n-2]$

para k entre $[j, n-1]$

chequear y guardar índice de sublista

intercambiar mínimo

ordenar

Iteraciones

$j = 0, 1, 2, \dots, n-2$

$k = 0, 1, 2, \dots, n-1 \rightarrow n$ iteraciones

$k = 1, 2, \dots, n-1 \rightarrow n-1$ iteraciones

\vdots

$k = n-3, n-2, n-1 \rightarrow 3$ iteraciones

$k = n-2, n-1 \rightarrow 2$ iteraciones

Iteraciones totales:

$$2 + 3 + 4 + \dots + n$$

Ordenamiento por selección

Lista de largo n

para j entre $[0, n-2]$

para k entre $[j, n-1]$

chequear y guardar **mínimo** de sublista

intercambiar mínimo con inicio lista sin ordenar

c operaciones fijas

d operaciones fijas

Ordenamiento por selección

Lista de largo n

para j entre [0, n-2]
 para k entre [j, n-1]
 *chequear y guardar **mínimo** de sublista*
 ***intercambiar** mínimo con inicio lista sin ordenar*

Cantidad de operaciones totales:

$$c * (2 + 3 + \dots + n) + d * (n-1)$$

Propiedad (suma de gauss):

$$1 + 2 + 3 + 4 + \dots + N = N*(N+1)/2$$

Complejidad temporal

$$T(n) = c * (2 + \dots + n-1 + n) + d * (n-1) = c * (n*(n+1) / 2 - 1) + d * (n-1)$$

$$T(n) = c/2 * n^2 + (d - c/2) * n - d \rightarrow O(n^2)$$

Ordenamiento por burbujeo

Ordenamiento por burbujeo

1. Comenzar comparando dos elementos desde la izquierda
2. Si el de la izquierda es mayor, intercambiar elementos
3. Incrementar una posición y continuar hasta recorrer toda la lista
4. Definir nueva sublista excluyendo el último elemento de la sublista anterior
5. Repetir todo el proceso hasta que la nueva sublista sea de largo 1

Ordenamiento por burbujeo

Lista original
desordenada

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

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

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

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

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

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

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

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

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

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

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

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

a	b
---	---

Cuando $a \leq b$, quedan ambos en la misma posición

a	b
---	---

Cuando $a > b$, se intercambian (siempre se lleva el más alto hacia la derecha)

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

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

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

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

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

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

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

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

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

Ordenamiento por burbujeo

para j entre $[1, n-1]$
 para k entre $[1, n-j]$
 si $der < izq$
 Intercambiar izq y der

3 5 4 6 2 1

3 5 4 6 2 1

3 4 5 6 2 1

3 4 5 6 2 1

3 4 5 2 6 1

3 4 5 2 1 6

3 4 5 2 1 6

3 4 5 2 1 6

3 4 5 2 1 6

3 4 2 5 1 6

3 4 2 1 5 6

3 4 2 1 5 6

3 4 2 1 5 6

3 2 4 1 5 6

3 2 1 4 5 6

3 2 1 4 5 6

2 3 1 4 5 6

2 1 3 4 5 6

2 1 3 4 5 6

1 2 3 4 5 6

a b

Cuando $a \leq b$, quedan ambos en la misma posición

a b

Cuando $a > b$, se intercambian (siempre se lleva el más alto hacia la derecha)

Lista de largo n desordenada

3 5 4 6 2 1

elemento 0

elemento $n-1$

Ordenamiento por burbujeo

Lista de largo n

```
para  $j$  entre  $[1, n-1]$   
  para  $k$  entre  $[1, n-j]$   
    si  $der < izq$   
      Intercambiar  $izq$  y  $der$ 
```

Primer ciclo: $n-1$ iteraciones
 $j = 1, 2, 3, \dots, n-1$

Ordenamiento por burbujeo

Lista de largo n

para j entre $[1, n-1]$

para k entre $[1, n-j]$

si $der < izq$

Intercambia

Repite el ciclo j veces

$j = 1, 2, 3, \dots, n-1$

$k = 1, 2, \dots, n-2, n-1 \rightarrow n-1$ iteraciones

$k = 1, 2, \dots, n-2 \rightarrow n-2$ iteraciones

\vdots

\vdots

$k = 1, 2 \rightarrow 2$ iteraciones

$k = 1 \rightarrow 1$ iteración

Iteraciones totales:

$$1 + 2 + 3 + \dots + n-1$$

Ordenamiento por burbujeo

Lista de largo n

para j entre $[1, n-1]$

para k entre $[1, n-j]$

si $der < izq$

Intercambiar izq y der

c operaciones fijas

Ordenamiento por burbujeo

Lista de largo n

```
para  $j$  entre  $[1, n-1]$   
  para  $k$  entre  $[1, n-j]$   
    si  $der < izq$   
      Intercambiar  $izq$  y  $der$ 
```

Cantidad de operaciones totales:

$$c * (1 + 2 + 3 + \dots + n-1)$$

Propiedad (suma de gauss):

$$1 + 2 + 3 + 4 + \dots + N = N*(N+1)/2$$

Complejidad temporal

$$T(n) = c * (1 + 2 + \dots + n-2 + n-1) = c * (n-1)*n / 2 = c/2 * n^2 - c/2 * n$$

$$T(n) = c/2 * n^2 - c/2 * n \rightarrow O(n^2)$$

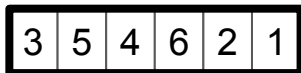
Ordenamiento por inserción

Ordenamiento por inserción

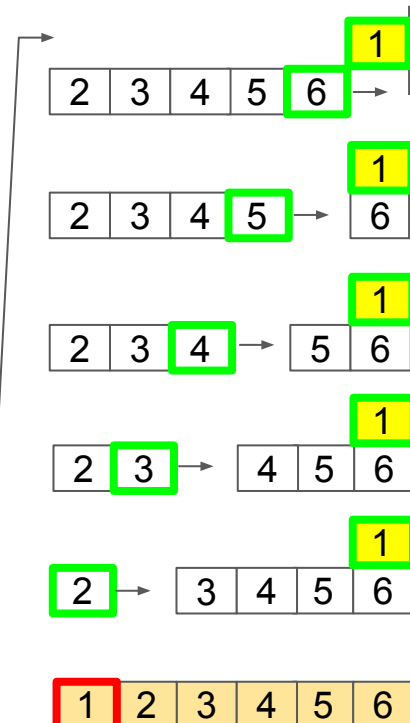
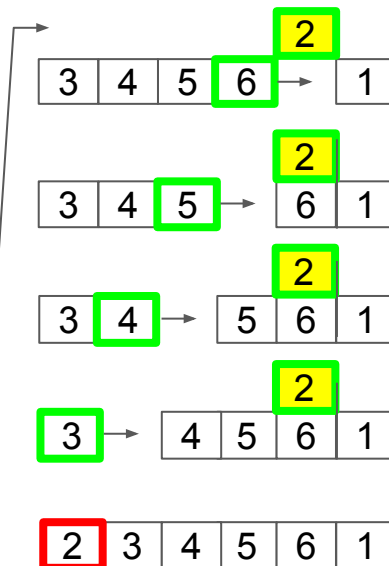
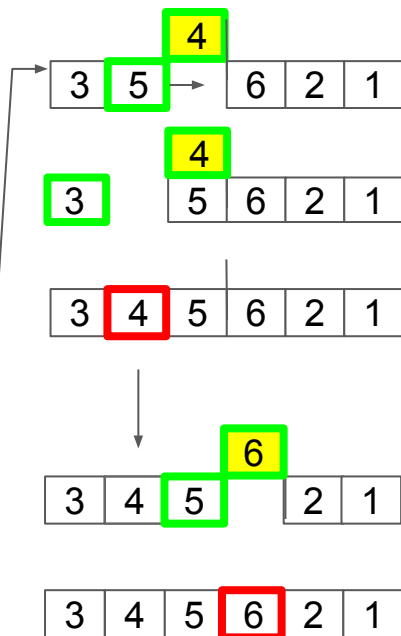
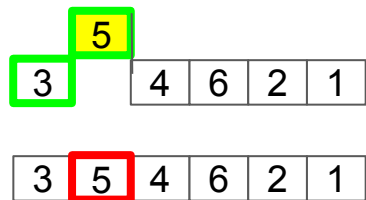
1. Inicialmente tomar la posición 1 como referencia (actual)
2. Comparar el elemento actual con los anteriores (a izquierda)
3. Insertar elemento actual a la derecha del primer elemento menor al actual o en la posición 0 si no se detectó ningún elemento menor
4. Incrementar la posición de referencia del elemento actual
5. Repetir desde paso 2 hasta que la posición de referencia llegue al final

Ordenamiento por inserción

Lista original
desordenada

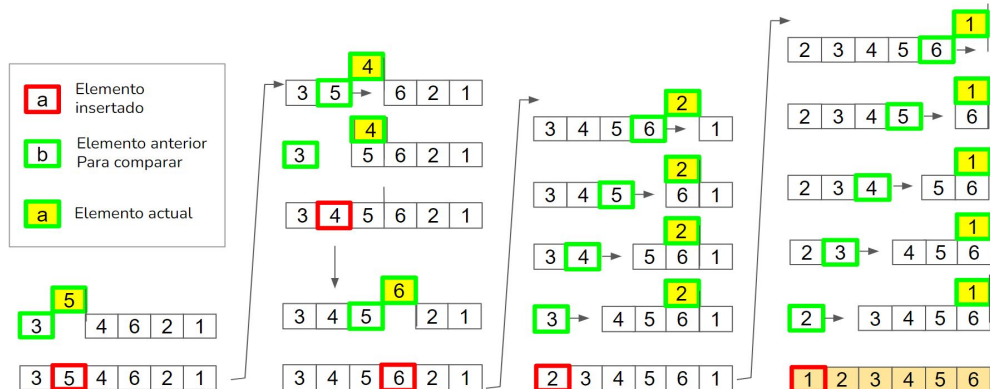


- a** Elemento insertado
- b** Elemento anterior Para comparar
- a** Elemento actual

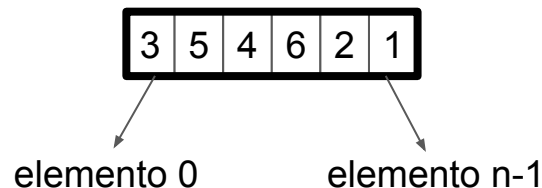


Ordenamiento por inserción

para j entre $[1, n-1]$
 actual \leftarrow elemento j
 $k \leftarrow j$
 mientras $k > 0$ **and** elemento $k-1 >$ *actual*
 guardar elemento $k-1$ en k
 $k \leftarrow k-1$
 guardar *actual* en k



Lista de largo n desordenada



Ordenamiento por inserción

Lista de largo n

```
para  $j$  entre  $[1, n-1]$   
   $actual \leftarrow \text{elemento } j$   
   $k \leftarrow j$   
  mientras  $k > 0$  and  $\text{elemento } k-1 > actual$   
    guardar  $\text{elemento } k-1$  en  $k$   
     $k \leftarrow k-1$   
  guardar  $actual$  en  $k$ 
```

Primer ciclo: $n-1$ iteraciones
 $j = 1, 2, 3, \dots, n-1$

Ordenamiento por inserción

Lista de largo n

para j entre $[1, n-1]$

$actual \leftarrow \text{elemento } j$

$k \leftarrow j$

mientras $k > 0$ **and** $\text{elemento } k-1 > actual$

guardar $\text{elemento } k-1$ en k

$k \leftarrow k-1$

guardar $actual$ en k

Repite el ciclo j veces
 $k = j, j-1, \dots, 3, 2, 1$

$k = 1$	$\rightarrow 1$ iteraciones
$k = 2, 1$	$\rightarrow 2$ iteraciones
$k = 3, 2, 1$	$\rightarrow 3$ iteraciones
\vdots	\vdots
$k = n-1, \dots, 3, 2, 1$	$\rightarrow n-1$ iteraciones

Iteraciones totales:

$1 + 2 + 3 + \dots + n-1$

Ordenamiento por inserción

Lista de largo n

para j entre $[1, n-1]$

$actual \leftarrow \text{elemento } j$

$k \leftarrow j$

mientras $k > 0$ **and** $\text{elemento } k-1 > actual$

guardar $\text{elemento } k-1$ en k

$k \leftarrow k-1$

guardar $actual$ en k

2 operaciones

1 operación

$c * (1 + 2 + 3 + \dots + n-1)$
operaciones

Ordenamiento por inserción

Lista de largo n

```
para  $j$  entre  $[1, n-1]$   
   $actual \leftarrow$  elemento  $j$   
   $k \leftarrow j$   
  mientras  $k > 0$  and elemento  $k-1 > actual$   
    guardar elemento  $k-1$  en  $k$   
     $k \leftarrow k-1$   
  guardar  $actual$  en  $k$ 
```

Cantidad de operaciones totales:

$$c * (1 + 2 + 3 + \dots + n-1) + 3 * (n-1)$$

Propiedad (suma de gauss):

$$1 + 2 + 3 + 4 + \dots + N = N * (N+1) / 2$$

Complejidad temporal

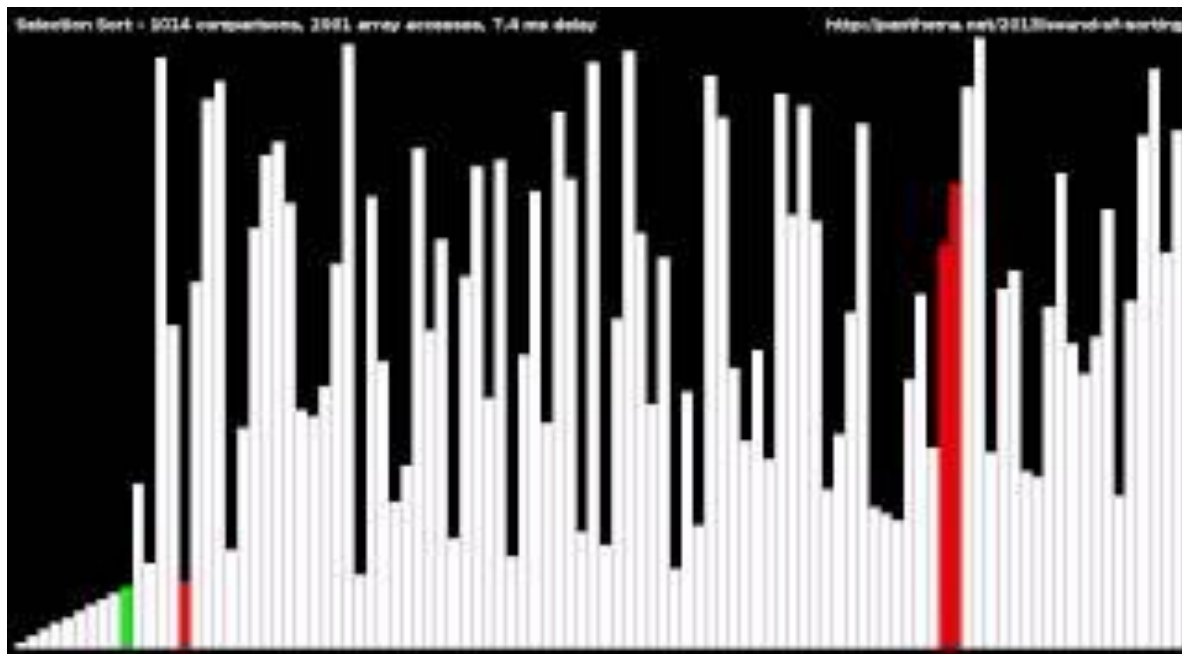
$$T(n) = c * (1 + 2 + \dots + n-2 + n-1) + 3 * (n-1) = c * (n-1) * n / 2 + 3 * (n-1)$$

$$T(n) = c/2 * n^2 + (3 - c/2) * n - 3 \rightarrow O(n^2)$$

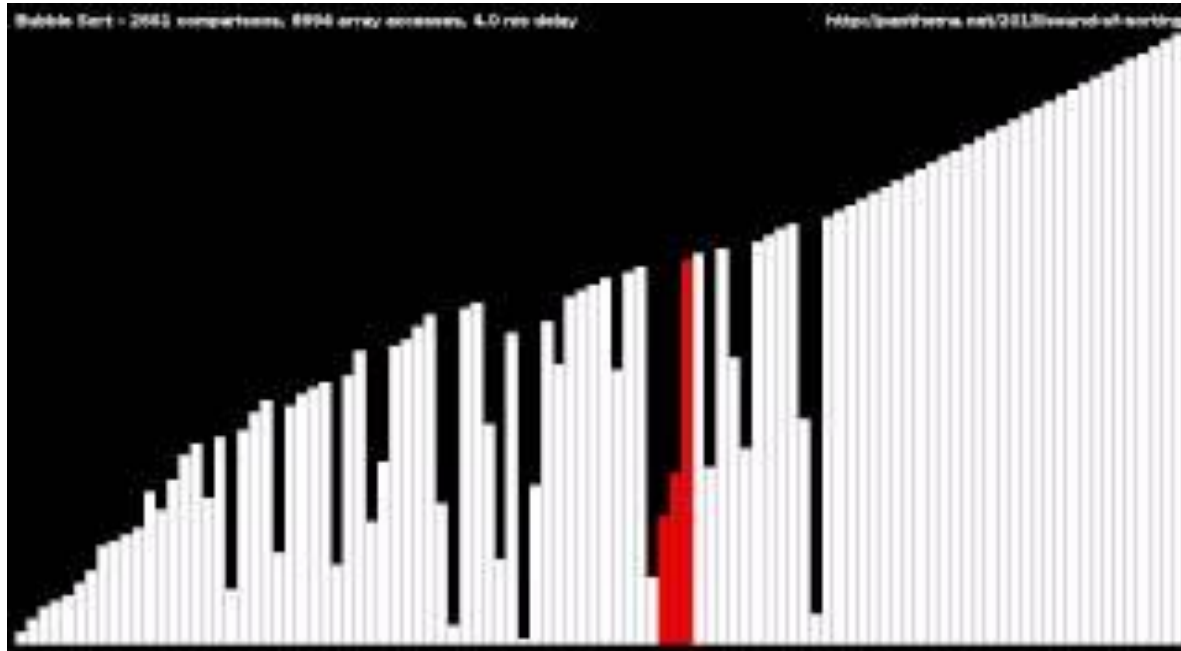
Complejidad temporal (Big-O):

- Búsqueda lineal: $O(n)$
- Búsqueda binaria: $O(\log n)$
- Ordenamiento por burbujeo: $O(n^2)$
- Ordenamiento por selección: $O(n^2)$
- Ordenamiento por inserción: $O(n^2)$

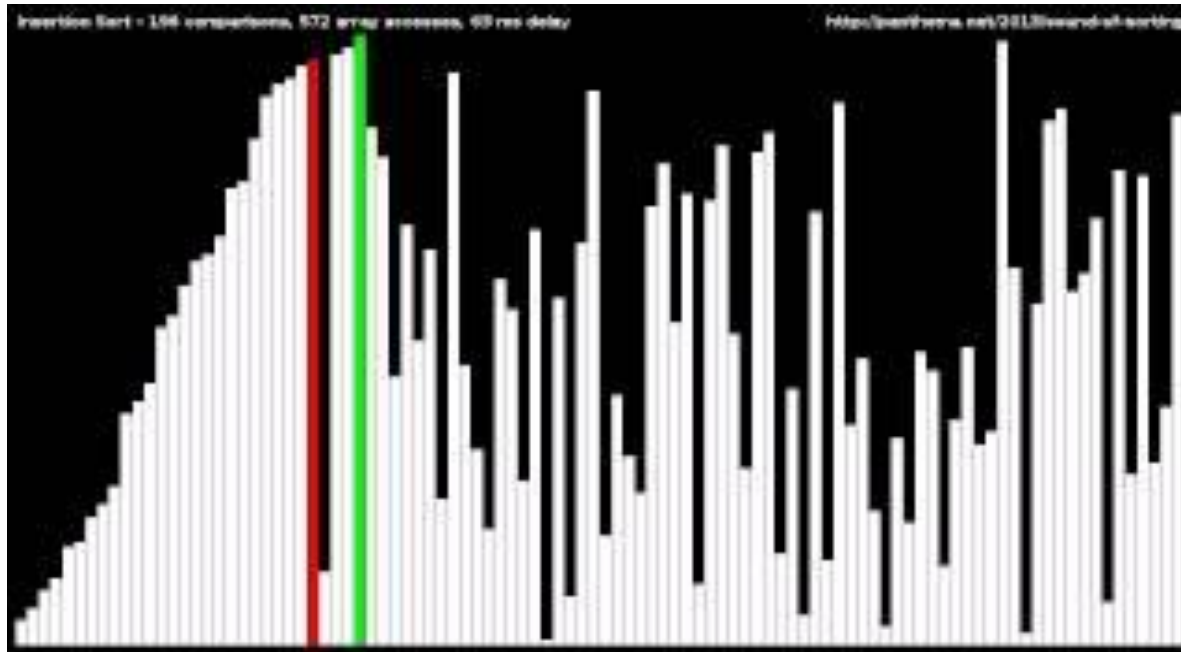
Ordenamiento por selección



Ordenamiento por burbujeo



Ordenamiento por inserción



Problema

Generar una lista de números aleatorios. Luego, implementar el ordenamiento de la lista mediante una función que ordene por burbujeo.