

Algoritmos de Ordenamiento

¿Qué son?

Entrada: array A de tipo T , longitud n

Salida: array A' de tipo T , longitud n que cumpla:

- **Ordenado:** $\forall i, j. i < j \implies A[i] \leq_T A[j]$
- **Permutación:** A y A' tienen los mismos elementos (y en iguales cantidades)

La comparación \leq_T es arbitraria, pero tiene que cumplir algunas propiedades sensatas:

- $a \leq_T a$
- si $a \leq_T b$ y $b \leq_T c$, entonces $a \leq_T c$

Ejemplo

$$[4, 2, 3, 1] \rightarrow [1, 2, 3, 4]$$

Ejemplo

$$[4, 2, 3, 1] \rightarrow [1, 2, 3, 4]$$

$$[1, 2, 3, 4] \rightarrow [1, 2, 3, 4]$$

Ejemplo

$$[4, 2, 3, 1] \rightarrow [1, 2, 3, 4]$$

$$[1, 2, 3, 4] \rightarrow [1, 2, 3, 4]$$

$$[1, 3, 2, 1, 4] \rightarrow [1, 1, 2, 3, 4]$$

Ejemplo

$$[4, 2, 3, 1] \rightarrow [1, 2, 3, 4]$$

$$[1, 2, 3, 4] \rightarrow [1, 2, 3, 4]$$

$$[1, 3, 2, 1, 4] \rightarrow [1, 1, 2, 3, 4]$$

- **Pregunta:** ¿cómo chequear si dos arrays son permutaciones uno del otro?

¿Para qué ordenar?

- Podemos buscar en $O(\lg n)$.
- Es parte de muchos otros algoritmos.

```
busqbin(A[N], lo, hi, v)
  if hi < lo  $\Rightarrow$ 
    return -1; // no está
  medio = (lo + hi) / 2
  if A[medio] = v  $\Rightarrow$ 
    return medio
  else if A[medio] < v  $\Rightarrow$ 
    return busqbin(A, medio+1, hi, v)
  else
    return busqbin(A, lo, medio-1, v)
```

Estilos

Hay MUCHOS algoritmos de ordenamiento, de distintos estilos.

- **Basados en comparación:** sólo podemos chequear con \leq_T

Estilos

Hay MUCHOS algoritmos de ordenamiento, de distintos estilos.

- **Basados en comparación:** sólo podemos chequear con \leq_T
- **Especializados al tipo** (e.g. enteros)

Ordenamiento por Comparación

Burbuja / Bubble Sort

- Muy malo... pero sirve para empezar a charlar

Burbuja / Bubble Sort

- Muy malo... pero sirve para empezar a charlar
- Idea: recorremos todo el arreglo dando vuelta “inversiones” adyacentes

Burbuja / Bubble Sort

- Muy malo... pero sirve para empezar a charlar
- Idea: recorremos todo el arreglo dando vuelta “inversiones” adyacentes
- Si en una vuelta no hicimos nada, terminamos (ya está ordenado)

Burbuja / Bubble Sort

- Muy malo... pero sirve para empezar a charlar
- Idea: recorremos todo el arreglo dando vuelta “inversiones” adyacentes
- Si en una vuelta no hicimos nada, terminamos (ya está ordenado)

```
bubble(A[N])  
  hiceAlgo = true  
  while hiceAlgo ⇒ (  
    hiceAlgo = false  
    for i in 1..n-1 ⇒  
      if A[i] > A[i+1] ⇒  
        A[i] ↔ A[i+1]  
        hiceAlgo = true  
  )
```

Burbuja / Bubble Sort

- Muy malo... pero sirve para empezar a charlar
- Idea: recorremos todo el arreglo dando vuelta “inversiones” adyacentes
- Si en una vuelta no hicimos nada, terminamos (ya está ordenado)

```
bubble(A[N])  
    hiceAlgo = true  
    while hiceAlgo ⇒ (  
        hiceAlgo = false  
        for i in 1..n-1 ⇒  
            if A[i] > A[i+1] ⇒  
                A[i] ↔ A[i+1]  
                hiceAlgo = true  
    )
```

[4,2,3,1]

Burbuja / Bubble Sort

- Muy malo... pero sirve para empezar a charlar
- Idea: recorremos todo el arreglo dando vuelta “inversiones” adyacentes
- Si en una vuelta no hicimos nada, terminamos (ya está ordenado)

```
bubble(A[N])  
  hiceAlgo = true  
  while hiceAlgo ⇒ (  
    hiceAlgo = false  
    for i in 1..n-1 ⇒  
      if A[i] > A[i+1] ⇒  
        A[i] ↔ A[i+1]  
        hiceAlgo = true  
  )
```

[4,2,3,1] → [2,4,3,1]

Burbuja / Bubble Sort

- Muy malo... pero sirve para empezar a charlar
- Idea: recorremos todo el arreglo dando vuelta “inversiones” adyacentes
- Si en una vuelta no hicimos nada, terminamos (ya está ordenado)

```
bubble(A[N])  
  hiceAlgo = true  
  while hiceAlgo ⇒ (  
    hiceAlgo = false  
    for i in 1..n-1 ⇒  
      if A[i] > A[i+1] ⇒  
        A[i] ↔ A[i+1]  
        hiceAlgo = true  
  )
```

$[4,2,3,1] \rightarrow [2,4,3,1] \rightarrow [2,3,4,1]$

Burbuja / Bubble Sort

- Muy malo... pero sirve para empezar a charlar
- Idea: recorremos todo el arreglo dando vuelta “inversiones” adyacentes
- Si en una vuelta no hicimos nada, terminamos (ya está ordenado)

```
bubble(A[N])  
  hiceAlgo = true  
  while hiceAlgo ⇒ (  
    hiceAlgo = false  
    for i in 1..n-1 ⇒  
      if A[i] > A[i+1] ⇒  
        A[i] ↔ A[i+1]  
        hiceAlgo = true  
  )
```

\Rightarrow $[4,2,3,1] \rightarrow [2,4,3,1] \rightarrow [2,3,4,1]$
 $\Rightarrow [2,3,1,4]$

Burbuja / Bubble Sort

- Muy malo... pero sirve para empezar a charlar
- Idea: recorremos todo el arreglo dando vuelta “inversiones” adyacentes
- Si en una vuelta no hicimos nada, terminamos (ya está ordenado)

```
bubble(A[N])  
  hiceAlgo = true  
  while hiceAlgo ⇒ (  
    hiceAlgo = false  
    for i in 1..n-1 ⇒  
      if A[i] > A[i+1] ⇒  
        A[i] ↔ A[i+1]  
        hiceAlgo = true  
  )
```

$\Rightarrow [4,2,3,1] \rightarrow [2,4,3,1] \rightarrow [2,3,4,1]$
 $\Rightarrow [2,3,1,4] \rightarrow [2,3,1,4]$

Burbuja / Bubble Sort

- Muy malo... pero sirve para empezar a charlar
- Idea: recorremos todo el arreglo dando vuelta “inversiones” adyacentes
- Si en una vuelta no hicimos nada, terminamos (ya está ordenado)

```
bubble(A[N])  
  hiceAlgo = true  
  while hiceAlgo ⇒ (  
    hiceAlgo = false  
    for i in 1..n-1 ⇒  
      if A[i] > A[i+1] ⇒  
        A[i] ↔ A[i+1]  
        hiceAlgo = true  
  )
```

\Rightarrow $[4,2,3,1] \rightarrow [2,4,3,1] \rightarrow [2,3,4,1]$
 \Rightarrow $[2,3,1,4] \rightarrow [2,3,1,4] \rightarrow [2,1,3,4]$

Burbuja / Bubble Sort

- Muy malo... pero sirve para empezar a charlar
- Idea: recorremos todo el arreglo dando vuelta “inversiones” adyacentes
- Si en una vuelta no hicimos nada, terminamos (ya está ordenado)

```
bubble(A[N])  
  hiceAlgo = true  
  while hiceAlgo ⇒ (  
    hiceAlgo = false  
    for i in 1..n-1 ⇒  
      if A[i] > A[i+1] ⇒  
        A[i] ↔ A[i+1]  
        hiceAlgo = true  
  )
```

$[4,2,3,1] \rightarrow [2,4,3,1] \rightarrow [2,3,4,1]$
 $\Rightarrow [2,3,1,4] \rightarrow [2,3,1,4] \rightarrow [2,1,3,4]$
 $\Rightarrow [2,1,3,4]$

Burbuja / Bubble Sort

- Muy malo... pero sirve para empezar a charlar
- Idea: recorremos todo el arreglo dando vuelta “inversiones” adyacentes
- Si en una vuelta no hicimos nada, terminamos (ya está ordenado)

```
bubble(A[N])  
  hiceAlgo = true  
  while hiceAlgo ⇒ (  
    hiceAlgo = false  
    for i in 1..n-1 ⇒  
      if A[i] > A[i+1] ⇒  
        A[i] ↔ A[i+1]  
        hiceAlgo = true  
  )
```

$[4,2,3,1] \rightarrow [2,4,3,1] \rightarrow [2,3,4,1]$
 $\Rightarrow [2,3,1,4] \rightarrow [2,3,1,4] \rightarrow [2,1,3,4]$
 $\Rightarrow [2,1,3,4] \rightarrow [1,2,3,4]$

Burbuja / Bubble Sort

- Muy malo... pero sirve para empezar a charlar
- Idea: recorremos todo el arreglo dando vuelta “inversiones” adyacentes
- Si en una vuelta no hicimos nada, terminamos (ya está ordenado)

```
bubble(A[N])  
  hiceAlgo = true  
  while hiceAlgo ⇒ (  
    hiceAlgo = false  
    for i in 1..n-1 ⇒  
      if A[i] > A[i+1] ⇒  
        A[i] ↔ A[i+1]  
        hiceAlgo = true  
  )
```

$[4,2,3,1] \rightarrow [2,4,3,1] \rightarrow [2,3,4,1]$
 $\Rightarrow [2,3,1,4] \rightarrow [2,3,1,4] \rightarrow [2,1,3,4]$
 $\Rightarrow [2,1,3,4] \rightarrow [1,2,3,4] \rightarrow [1,2,3,4]$

Burbuja / Bubble Sort

- Muy malo... pero sirve para empezar a charlar
- Idea: recorremos todo el arreglo dando vuelta “inversiones” adyacentes
- Si en una vuelta no hicimos nada, terminamos (ya está ordenado)

```
bubble(A[N])  
  hiceAlgo = true  
  while hiceAlgo ⇒ (  
    hiceAlgo = false  
    for i in 1..n-1 ⇒  
      if A[i] > A[i+1] ⇒  
        A[i] ↔ A[i+1]  
        hiceAlgo = true  
  )
```

$[4,2,3,1] \rightarrow [2,4,3,1] \rightarrow [2,3,4,1]$
 $\Rightarrow [2,3,1,4] \rightarrow [2,3,1,4] \rightarrow [2,1,3,4]$
 $\Rightarrow [2,1,3,4] \rightarrow [1,2,3,4] \rightarrow [1,2,3,4]$
 $\Rightarrow [1,2,3,4]$

Burbuja / Bubble Sort

- Muy malo... pero sirve para empezar a charlar
- Idea: recorremos todo el arreglo dando vuelta “inversiones” adyacentes
- Si en una vuelta no hicimos nada, terminamos (ya está ordenado)

```
bubble(A[N])  
  hiceAlgo = true  
  while hiceAlgo ⇒ (  
    hiceAlgo = false  
    for i in 1..n-1 ⇒  
      if A[i] > A[i+1] ⇒  
        A[i] ↔ A[i+1]  
        hiceAlgo = true  
  )
```

	⇒	[4,2,3,1]	→	[2,4,3,1]	→	[2,3,4,1]
	⇒	[2,3,1,4]	→	[2,3,1,4]	→	[2,1,3,4]
	⇒	[2,1,3,4]	→	[1,2,3,4]	→	[1,2,3,4]
	⇒	[1,2,3,4]	→	[1,2,3,4]		

Burbuja / Bubble Sort

- Muy malo... pero sirve para empezar a charlar
- Idea: recorremos todo el arreglo dando vuelta “inversiones” adyacentes
- Si en una vuelta no hicimos nada, terminamos (ya está ordenado)

```
bubble(A[N])  
  hiceAlgo = true  
  while hiceAlgo ⇒ (  
    hiceAlgo = false  
    for i in 1..n-1 ⇒  
      if A[i] > A[i+1] ⇒  
        A[i] ↔ A[i+1]  
        hiceAlgo = true  
  )
```

	⇒	[4,2,3,1]	→	[2,4,3,1]	→	[2,3,4,1]
	⇒	[2,3,1,4]	→	[2,3,1,4]	→	[2,1,3,4]
	⇒	[2,1,3,4]	→	[1,2,3,4]	→	[1,2,3,4]
	⇒	[1,2,3,4]	→	[1,2,3,4]	→	[1,2,3,4]

Burbuja / Bubble Sort

- Muy malo... pero sirve para empezar a charlar
- Idea: recorremos todo el arreglo dando vuelta “inversiones” adyacentes
- Si en una vuelta no hicimos nada, terminamos (ya está ordenado)

```
bubble(A[N])  
  hiceAlgo = true  
  while hiceAlgo ⇒ (  
    hiceAlgo = false  
    for i in 1..n-1 ⇒  
      if A[i] > A[i+1] ⇒  
        A[i] ↔ A[i+1]  
        hiceAlgo = true  
  )
```

⇒ [4,2,3,1] → [2,4,3,1] → [2,3,4,1]
⇒ [2,3,1,4] → [2,3,1,4] → [2,1,3,4]
⇒ [2,1,3,4] → [1,2,3,4] → [1,2,3,4]
⇒ [1,2,3,4] → [1,2,3,4] → [1,2,3,4]

- Primera pasada: el máximo queda al final.

Burbuja / Bubble Sort

- Muy malo... pero sirve para empezar a charlar
- Idea: recorremos todo el arreglo dando vuelta “inversiones” adyacentes
- Si en una vuelta no hicimos nada, terminamos (ya está ordenado)

```
bubble(A[N])  
  hiceAlgo = true  
  while hiceAlgo ⇒ (  
    hiceAlgo = false  
    for i in 1..n-1 ⇒  
      if A[i] > A[i+1] ⇒  
        A[i] ↔ A[i+1]  
        hiceAlgo = true  
  )
```

⇒ [4,2,3,1] → [2,4,3,1] → [2,3,4,1]
⇒ [2,3,1,4] → [2,3,1,4] → [2,1,3,4]
⇒ [2,1,3,4] → [1,2,3,4] → [1,2,3,4]
⇒ [1,2,3,4] → [1,2,3,4] → [1,2,3,4]

- Primera pasada: el máximo queda al final.
- Máximo de pasadas:

Burbuja / Bubble Sort

- Muy malo... pero sirve para empezar a charlar
- Idea: recorremos todo el arreglo dando vuelta “inversiones” adyacentes
- Si en una vuelta no hicimos nada, terminamos (ya está ordenado)

```
bubble(A[N])  
  hiceAlgo = true  
  while hiceAlgo ⇒ (  
    hiceAlgo = false  
    for i in 1..n-1 ⇒  
      if A[i] > A[i+1] ⇒  
        A[i] ↔ A[i+1]  
        hiceAlgo = true  
  )
```

⇒ [4,2,3,1] → [2,4,3,1] → [2,3,4,1]
⇒ [2,3,1,4] → [2,3,1,4] → [2,1,3,4]
⇒ [2,1,3,4] → [1,2,3,4] → [1,2,3,4]
⇒ [1,2,3,4] → [1,2,3,4] → [1,2,3,4]

- Primera pasada: el máximo queda al final.
- Máximo de pasadas: n

Burbuja / Bubble Sort

- Muy malo... pero sirve para empezar a charlar
- Idea: recorremos todo el arreglo dando vuelta “inversiones” adyacentes
- Si en una vuelta no hicimos nada, terminamos (ya está ordenado)

```
bubble(A[N])  
  hiceAlgo = true  
  while hiceAlgo ⇒ (  
    hiceAlgo = false  
    for i in 1..n-1 ⇒  
      if A[i] > A[i+1] ⇒  
        A[i] ↔ A[i+1]  
        hiceAlgo = true  
  )
```

⇒ [4,2,3,1] → [2,4,3,1] → [2,3,4,1]
⇒ [2,3,1,4] → [2,3,1,4] → [2,1,3,4]
⇒ [2,1,3,4] → [1,2,3,4] → [1,2,3,4]
⇒ [1,2,3,4] → [1,2,3,4] → [1,2,3,4]

- Primera pasada: el máximo queda al final.
- Máximo de pasadas: n
- Tiempo de ejecución:

Burbuja / Bubble Sort

- Muy malo... pero sirve para empezar a charlar
- Idea: recorremos todo el arreglo dando vuelta “inversiones” adyacentes
- Si en una vuelta no hicimos nada, terminamos (ya está ordenado)

```
bubble(A[N])  
  hiceAlgo = true  
  while hiceAlgo ⇒ (  
    hiceAlgo = false  
    for i in 1..n-1 ⇒  
      if A[i] > A[i+1] ⇒  
        A[i] ↔ A[i+1]  
        hiceAlgo = true  
  )
```

⇒ [4,2,3,1] → [2,4,3,1] → [2,3,4,1]
⇒ [2,3,1,4] → [2,3,1,4] → [2,1,3,4]
⇒ [2,1,3,4] → [1,2,3,4] → [1,2,3,4]
⇒ [1,2,3,4] → [1,2,3,4] → [1,2,3,4]

- Primera pasada: el máximo queda al final.
- Máximo de pasadas: n
- Tiempo de ejecución: $O(n^2)$

Burbuja / Bubble Sort

- Muy malo... pero sirve para empezar a charlar
- Idea: recorremos todo el arreglo dando vuelta “inversiones” adyacentes
- Si en una vuelta no hicimos nada, terminamos (ya está ordenado)

```
bubble(A[N])  
  hiceAlgo = true  
  while hiceAlgo ⇒ (  
    hiceAlgo = false  
    for i in 1..n-1 ⇒  
      if A[i] > A[i+1] ⇒  
        A[i] ↔ A[i+1]  
        hiceAlgo = true  
  )
```

$[4, 2, 3, 1] \rightarrow [2, 4, 3, 1] \rightarrow [2, 3, 4, 1]$
 $\Rightarrow [2, 3, 1, 4] \rightarrow [2, 3, 1, 4] \rightarrow [2, 1, 3, 4]$
 $\Rightarrow [2, 1, 3, 4] \rightarrow [1, 2, 3, 4] \rightarrow [1, 2, 3, 4]$
 $\Rightarrow [1, 2, 3, 4] \rightarrow [1, 2, 3, 4] \rightarrow [1, 2, 3, 4]$

- Primera pasada: el máximo queda al final.
- Máximo de pasadas: n
- Tiempo de ejecución: $O(n^2)$
- Memoria:

Burbuja / Bubble Sort

- Muy malo... pero sirve para empezar a charlar
- Idea: recorremos todo el arreglo dando vuelta “inversiones” adyacentes
- Si en una vuelta no hicimos nada, terminamos (ya está ordenado)

```
bubble(A[N])  
  hiceAlgo = true  
  while hiceAlgo ⇒ (  
    hiceAlgo = false  
    for i in 1..n-1 ⇒  
      if A[i] > A[i+1] ⇒  
        A[i] ↔ A[i+1]  
        hiceAlgo = true  
  )
```

⇒ [4,2,3,1] → [2,4,3,1] → [2,3,4,1]
⇒ [2,3,1,4] → [2,3,1,4] → [2,1,3,4]
⇒ [2,1,3,4] → [1,2,3,4] → [1,2,3,4]
⇒ [1,2,3,4] → [1,2,3,4] → [1,2,3,4]

- Primera pasada: el máximo queda al final.
- Máximo de pasadas: n
- Tiempo de ejecución: $O(n^2)$
- Memoria: $O(1)$

Inserción / Insertion Sort

- Razonable para arreglos pequeños ($N \leq 50$).
- Idea: vamos llevando un prefijo ordenado, y agregando elementos de a uno
- Hacemos eso $N - 1$ veces

Inserción / Insertion Sort

- Razonable para arreglos pequeños ($N \leq 50$).
- Idea: vamos llevando un prefijo ordenado, y agregando elementos de a uno
- Hacemos eso $N - 1$ veces

```
insercion(A[N])  
  for i in 2..N ⇒  
    for j in i-1..1 ⇒  
      if A[j] > A[j+1] ⇒  
        A[j] ↔ A[j+1]  
      else ⇒ break;
```

Inserción / Insertion Sort

- Razonable para arreglos pequeños ($N \leq 50$).
- Idea: vamos llevando un prefijo ordenado, y agregando elementos de a uno
- Hacemos eso $N - 1$ veces

```
insercion(A[N])  
  for i in 2..N ⇒  
    for j in i-1..1 ⇒  
      if A[j] > A[j+1] ⇒  
        A[j] ↔ A[j+1]  
      else ⇒ break;
```

(ordenado)	$a[i]$...
------------	--------	-----

Inserción / Insertion Sort

- Razonable para arreglos pequeños ($N \leq 50$).
- Idea: vamos llevando un prefijo ordenado, y agregando elementos de a uno
- Hacemos eso $N - 1$ veces

```
insercion(A[N])  
  for i in 2..N ⇒  
    for j in i-1..1 ⇒  
      if A[j] > A[j+1] ⇒  
        A[j] ↔ A[j+1]  
      else ⇒ break;
```

(ordenado)	$a[i]$...
------------	--------	-----

=

$\leq a[i]$	$> a[i]$	$a[i]$...
-------------	----------	--------	-----

Insertión / Insertion Sort

- Razonable para arreglos pequeños ($N \leq 50$).
- Idea: vamos llevando un prefijo ordenado, y agregando elementos de a uno
- Hacemos eso $N - 1$ veces

```
insercion(A[N])  
  for i in 2..N ⇒  
    for j in i-1..1 ⇒  
      if A[j] > A[j+1] ⇒  
        A[j] ↔ A[j+1]  
      else ⇒ break;
```

(ordenado)	$a[i]$...
------------	--------	-----

=

$\leq a[i]$	$> a[i]$	$a[i]$...
-------------	----------	--------	-----

↔

$\leq a[i]$	$a[i]$	$> a[i]$...
-------------	--------	----------	-----

Insertión / Insertion Sort

- Razonable para arreglos pequeños ($N \leq 50$).
- Idea: vamos llevando un prefijo ordenado, y agregando elementos de a uno
- Hacemos eso $N - 1$ veces

insercion(A[N])

[4,2,3,1]

for i in 2..N \Rightarrow

for j in i-1..1 \Rightarrow

if A[j] > A[j+1] \Rightarrow

A[j] \leftrightarrow A[j+1]

else \Rightarrow break;

(ordenado)	$a[i]$...
------------	--------	-----

=

$\leq a[i]$	$> a[i]$	$a[i]$...
-------------	----------	--------	-----

\rightsquigarrow

$\leq a[i]$	$a[i]$	$> a[i]$...
-------------	--------	----------	-----

Insertión / Insertion Sort

- Razonable para arreglos pequeños ($N \leq 50$).
- Idea: vamos llevando un prefijo ordenado, y agregando elementos de a uno
- Hacemos eso $N - 1$ veces

```
insercion(A[N])  
  for i in 2..N ⇒  
    for j in i-1..1 ⇒  
      if A[j] > A[j+1] ⇒  
        A[j] ↔ A[j+1]  
      else ⇒ break;
```

$[4, 2, 3, 1] \rightarrow [2, 4, 3, 1]$

(ordenado)	$a[i]$...
------------	--------	-----

=

$\leq a[i]$	$> a[i]$	$a[i]$...
-------------	----------	--------	-----

↪

$\leq a[i]$	$a[i]$	$> a[i]$...
-------------	--------	----------	-----

Insertión / Insertion Sort

- Razonable para arreglos pequeños ($N \leq 50$).
- Idea: vamos llevando un prefijo ordenado, y agregando elementos de a uno
- Hacemos eso $N - 1$ veces

```
insercion(A[N])  
  for i in 2..N ⇒  
    for j in i-1..1 ⇒  
      if A[j] > A[j+1] ⇒  
        A[j] ↔ A[j+1]  
      else ⇒ break;
```

$[4, 2, 3, 1] \rightarrow [2, 4, 3, 1] \rightarrow [2, 3, 4, 1]$

(ordenado)	$a[i]$...
------------	--------	-----

=

$\leq a[i]$	$> a[i]$	$a[i]$...
-------------	----------	--------	-----

≈

$\leq a[i]$	$a[i]$	$> a[i]$...
-------------	--------	----------	-----

Inserción / Insertion Sort

- Razonable para arreglos pequeños ($N \leq 50$).
- Idea: vamos llevando un prefijo ordenado, y agregando elementos de a uno
- Hacemos eso $N - 1$ veces

```
insercion(A[N])  
  for i in 2..N ⇒  
    for j in i-1..1 ⇒  
      if A[j] > A[j+1] ⇒  
        A[j] ↔ A[j+1]  
      else ⇒ break;
```

$[4, 2, 3, 1] \rightarrow [2, 4, 3, 1] \rightarrow [2, 3, 4, 1]$
 $\rightarrow [2, 3, 4, 1]$

(ordenado)	$a[i]$...
------------	--------	-----

=

$\leq a[i]$	$> a[i]$	$a[i]$...
-------------	----------	--------	-----

↔

$\leq a[i]$	$a[i]$	$> a[i]$...
-------------	--------	----------	-----

Inserción / Insertion Sort

- Razonable para arreglos pequeños ($N \leq 50$).
- Idea: vamos llevando un prefijo ordenado, y agregando elementos de a uno
- Hacemos eso $N - 1$ veces

```
insercion(A[N])  
  for i in 2..N ⇒  
    for j in i-1..1 ⇒  
      if A[j] > A[j+1] ⇒  
        A[j] ↔ A[j+1]  
      else ⇒ break;
```

$[4, 2, 3, 1] \rightarrow [2, 4, 3, 1] \rightarrow [2, 3, 4, 1]$
 $\rightarrow [2, 3, 4, 1] \rightarrow [2, 3, 1, 4]$

(ordenado)	$a[i]$...
------------	--------	-----

=

$\leq a[i]$	$> a[i]$	$a[i]$...
-------------	----------	--------	-----

≈

$\leq a[i]$	$a[i]$	$> a[i]$...
-------------	--------	----------	-----

Inserción / Insertion Sort

- Razonable para arreglos pequeños ($N \leq 50$).
- Idea: vamos llevando un prefijo ordenado, y agregando elementos de a uno
- Hacemos eso $N - 1$ veces

```
insercion(A[N])  
  for i in 2..N ⇒  
    for j in i-1..1 ⇒  
      if A[j] > A[j+1] ⇒  
        A[j] ↔ A[j+1]  
      else ⇒ break;
```

$[4, 2, 3, 1] \rightarrow [2, 4, 3, 1] \rightarrow [2, 3, 4, 1]$
 $\rightarrow [2, 3, 4, 1] \rightarrow [2, 3, 1, 4] \rightarrow [2, 1, 3, 4]$

(ordenado)	$a[i]$...
------------	--------	-----

=

$\leq a[i]$	$> a[i]$	$a[i]$...
-------------	----------	--------	-----

≈

$\leq a[i]$	$a[i]$	$> a[i]$...
-------------	--------	----------	-----

Inserción / Insertion Sort

- Razonable para arreglos pequeños ($N \leq 50$).
- Idea: vamos llevando un prefijo ordenado, y agregando elementos de a uno
- Hacemos eso $N - 1$ veces

```
insercion(A[N])  
  for i in 2..N ⇒  
    for j in i-1..1 ⇒  
      if A[j] > A[j+1] ⇒  
        A[j] ↔ A[j+1]  
      else ⇒ break;
```

$[4, 2, 3, 1] \rightarrow [2, 4, 3, 1] \rightarrow [2, 3, 4, 1]$
 $\rightarrow [2, 3, 4, 1] \rightarrow [2, 3, 1, 4] \rightarrow [2, 1, 3, 4]$
 $\rightarrow [1, 2, 3, 4]$

(ordenado)	$a[i]$...
------------	--------	-----

=

$\leq a[i]$	$> a[i]$	$a[i]$...
-------------	----------	--------	-----

≈

$\leq a[i]$	$a[i]$	$> a[i]$...
-------------	--------	----------	-----

Inserción / Insertion Sort

- Razonable para arreglos pequeños ($N \leq 50$).
- Idea: vamos llevando un prefijo ordenado, y agregando elementos de a uno
- Hacemos eso $N - 1$ veces

```
insercion(A[N])  
  for i in 2..N ⇒  
    for j in i-1..1 ⇒  
      if A[j] > A[j+1] ⇒  
        A[j] ↔ A[j+1]  
      else ⇒ break;
```

(ordenado)	$a[i]$...
------------	--------	-----

=

$\leq a[i]$	$> a[i]$	$a[i]$...
-------------	----------	--------	-----

↔

$\leq a[i]$	$a[i]$	$> a[i]$...
-------------	--------	----------	-----

$[4, 2, 3, 1] \rightarrow [2, 4, 3, 1] \rightarrow [2, 3, 4, 1]$
 $\rightarrow [2, 3, 4, 1] \rightarrow [2, 3, 1, 4] \rightarrow [2, 1, 3, 4]$
 $\rightarrow [1, 2, 3, 4]$

- Tiempo de ejecución:

Insertión / Insertion Sort

- Razonable para arreglos pequeños ($N \leq 50$).
- Idea: vamos llevando un prefijo ordenado, y agregando elementos de a uno
- Hacemos eso $N - 1$ veces

```
insercion(A[N])  
  for i in 2..N ⇒  
    for j in i-1..1 ⇒  
      if A[j] > A[j+1] ⇒  
        A[j] ↔ A[j+1]  
      else ⇒ break;
```

(ordenado)	$a[i]$...
------------	--------	-----

=

$\leq a[i]$	$> a[i]$	$a[i]$...
-------------	----------	--------	-----

↪

$\leq a[i]$	$a[i]$	$> a[i]$...
-------------	--------	----------	-----

$[4, 2, 3, 1] \rightarrow [2, 4, 3, 1] \rightarrow [2, 3, 4, 1]$
 $\rightarrow [2, 3, 4, 1] \rightarrow [2, 3, 1, 4] \rightarrow [2, 1, 3, 4]$
 $\rightarrow [1, 2, 3, 4]$

- Tiempo de ejecución: $O(n^2)$

Inserción / Insertion Sort

- Razonable para arreglos pequeños ($N \leq 50$).
- Idea: vamos llevando un prefijo ordenado, y agregando elementos de a uno
- Hacemos eso $N - 1$ veces

```
insercion(A[N])  
  for i in 2..N =>  
    for j in i-1..1 =>  
      if A[j] > A[j+1] =>  
        A[j] ↔ A[j+1]  
      else => break;
```

(ordenado)	$a[i]$...
------------	--------	-----

=

$\leq a[i]$	$> a[i]$	$a[i]$...
-------------	----------	--------	-----

↪

$\leq a[i]$	$a[i]$	$> a[i]$...
-------------	--------	----------	-----

$[4, 2, 3, 1] \rightarrow [2, 4, 3, 1] \rightarrow [2, 3, 4, 1]$
 $\rightarrow [2, 3, 4, 1] \rightarrow [2, 3, 1, 4] \rightarrow [2, 1, 3, 4]$
 $\rightarrow [1, 2, 3, 4]$

- Tiempo de ejecución: $O(n^2)$
- Memoria:

Inserción / Insertion Sort

- Razonable para arreglos pequeños ($N \leq 50$).
- Idea: vamos llevando un prefijo ordenado, y agregando elementos de a uno
- Hacemos eso $N - 1$ veces

```
insercion(A[N])  
  for i in 2..N =>  
    for j in i-1..1 =>  
      if A[j] > A[j+1] =>  
        A[j] ↔ A[j+1]  
      else => break;
```

(ordenado)	$a[i]$...
------------	--------	-----

=

$\leq a[i]$	$> a[i]$	$a[i]$...
-------------	----------	--------	-----

↪

$\leq a[i]$	$a[i]$	$> a[i]$...
-------------	--------	----------	-----

$[4, 2, 3, 1] \rightarrow [2, 4, 3, 1] \rightarrow [2, 3, 4, 1]$
 $\rightarrow [2, 3, 4, 1] \rightarrow [2, 3, 1, 4] \rightarrow [2, 1, 3, 4]$
 $\rightarrow [1, 2, 3, 4]$

- Tiempo de ejecución: $O(n^2)$
- Memoria: $O(1)$

Inserción / Insertion Sort

- Razonable para arreglos pequeños ($N \leq 50$).
- Idea: vamos llevando un prefijo ordenado, y agregando elementos de a uno
- Hacemos eso $N - 1$ veces

```
insercion(A[N])  
  for i in 2..N =>  
    for j in i-1..1 =>  
      if A[j] > A[j+1] =>  
        A[j] ↔ A[j+1]  
      else => break;
```

(ordenado)	$a[i]$...
------------	--------	-----

=

$\leq a[i]$	$> a[i]$	$a[i]$...
-------------	----------	--------	-----

↔

$\leq a[i]$	$a[i]$	$> a[i]$...
-------------	--------	----------	-----

$[4, 2, 3, 1] \rightarrow [2, 4, 3, 1] \rightarrow [2, 3, 4, 1]$
 $\rightarrow [2, 3, 4, 1] \rightarrow [2, 3, 1, 4] \rightarrow [2, 1, 3, 4]$
 $\rightarrow [1, 2, 3, 4]$

- Tiempo de ejecución: $O(n^2)$
- Memoria: $O(1)$
- **Ejercicio:** Optimizar para evitar los swaps. En vez de eso, mover todo el subarray ($> a[i]$). Medir diferencia.

Selección / Selection Sort

- Razonable para arreglos pequeños... pero suele ser peor que Inserción
- Idea: buscamos el mínimo y lo ponemos al principio
- Hacemos eso $N - 1$ veces

Selección / Selection Sort

- Razonable para arreglos pequeños... pero suele ser peor que Inserción
- Idea: buscamos el mínimo y lo ponemos al principio
- Hacemos eso $N - 1$ veces

```
seleccion(A[N])  
  for i in 1..N-1 ⇒  
    minPos = i  
    for j in i+1..N ⇒  
      if A[j] < A[minPos] ⇒ minPos = j  
    A[i] ↔ A[minPos]
```

Selección / Selection Sort

- Razonable para arreglos pequeños... pero suele ser peor que Inserción
- Idea: buscamos el mínimo y lo ponemos al principio
- Hacemos eso $N - 1$ veces

```
seleccion(A[N])  
  for i in 1..N-1 ⇒  
    minPos = i  
    for j in i+1..N ⇒  
      if A[j] < A[minPos] ⇒ minPos = j  
    A[i] ↔ A[minPos]
```

Selección / Selection Sort

- Razonable para arreglos pequeños... pero suele ser peor que Inserción
- Idea: buscamos el mínimo y lo ponemos al principio
- Hacemos eso $N - 1$ veces

<code>seleccion(A[N])</code>		<code>[4,2,1,3]</code>
<code> for i in 1..N-1 ⇒</code>	<code>→</code>	<code>[1,2,4,3]</code>
<code>minPos = i</code>	<code>→</code>	<code>[1,2,4,3]</code>
<code>for j in i+1..N ⇒</code>	<code>→</code>	<code>[1,2,3,4]</code>
<code>if A[j] < A[minPos] ⇒ minPos = j</code>		
<code>A[i] ↔ A[minPos]</code>		

Selección / Selection Sort

- Razonable para arreglos pequeños... pero suele ser peor que Inserción
- Idea: buscamos el mínimo y lo ponemos al principio
- Hacemos eso $N - 1$ veces

```
seleccion(A[N])  
  for i in 1..N-1 ⇒  
    minPos = i  
    for j in i+1..N ⇒  
      if A[j] < A[minPos] ⇒ minPos = j  
    A[i] ↔ A[minPos]
```

[4,2,1,3]
→ [1,2,4,3]
→ [1,2,4,3]
→ [1,2,3,4]

- Tiempo de ejecución:

Selección / Selection Sort

- Razonable para arreglos pequeños... pero suele ser peor que Inserción
- Idea: buscamos el mínimo y lo ponemos al principio
- Hacemos eso $N - 1$ veces

```
seleccion(A[N])  
  for i in 1..N-1 ⇒  
    minPos = i  
    for j in i+1..N ⇒  
      if A[j] < A[minPos] ⇒ minPos = j  
    A[i] ↔ A[minPos]
```

[4,2,1,3]
→ [1,2,4,3]
→ [1,2,4,3]
→ [1,2,3,4]

- Tiempo de ejecución: $O(n^2)$

Selección / Selection Sort

- Razonable para arreglos pequeños... pero suele ser peor que Inserción
- Idea: buscamos el mínimo y lo ponemos al principio
- Hacemos eso $N - 1$ veces

```
seleccion(A[N])  
  for i in 1..N-1 ⇒  
    minPos = i  
    for j in i+1..N ⇒  
      if A[j] < A[minPos] ⇒ minPos = j  
    A[i] ↔ A[minPos]
```

[4,2,1,3]
→ [1,2,4,3]
→ [1,2,4,3]
→ [1,2,3,4]

- Tiempo de ejecución: $O(n^2)$
- Memoria:

Selección / Selection Sort

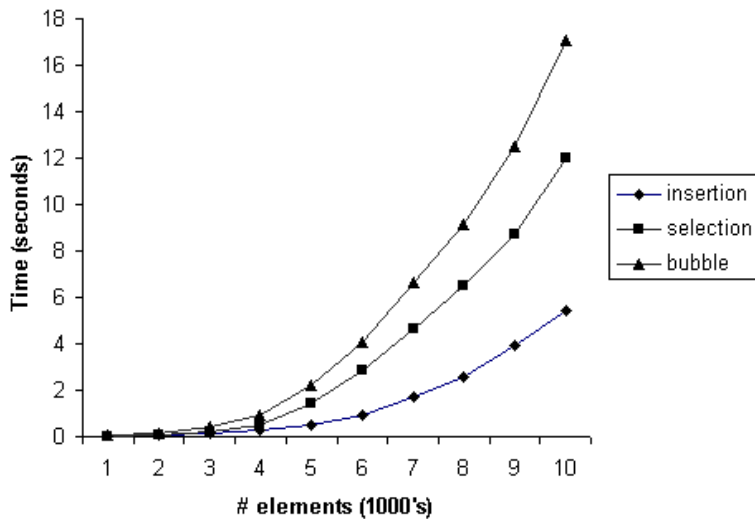
- Razonable para arreglos pequeños... pero suele ser peor que Inserción
- Idea: buscamos el mínimo y lo ponemos al principio
- Hacemos eso $N - 1$ veces

```
seleccion(A[N])  
  for i in 1..N-1 ⇒  
    minPos = i  
    for j in i+1..N ⇒  
      if A[j] < A[minPos] ⇒ minPos = j  
    A[i] ↔ A[minPos]
```

[4,2,1,3]
→ [1,2,4,3]
→ [1,2,4,3]
→ [1,2,3,4]

- Tiempo de ejecución: $O(n^2)$
- Memoria: $O(1)$

Sorting Strings



Por mezcla / Mergesort

- Hasta ahora vimos 3 algoritmos cuadráticos. Cambiemos de enfoque.
- Idea 1: si tengo dos arrays ordenados, se pueden combinar en $O(n)$

Por mezcla / Mergesort

- Hasta ahora vimos 3 algoritmos cuadráticos. Cambiemos de enfoque.
- Idea 1: si tengo dos arrays ordenados, se pueden combinar en $O(n)$

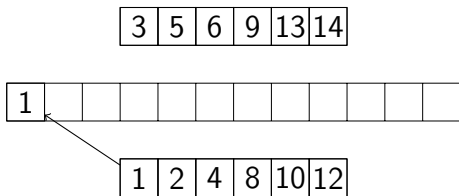
3	5	6	9	13	14
---	---	---	---	----	----

--	--	--	--	--	--	--	--	--	--	--	--

1	2	4	8	10	12
---	---	---	---	----	----

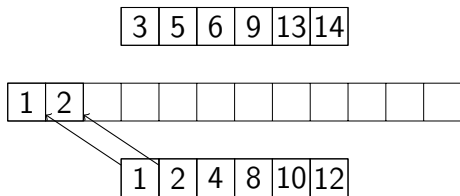
Por mezcla / Mergesort

- Hasta ahora vimos 3 algoritmos cuadráticos. Cambiemos de enfoque.
- Idea 1: si tengo dos arrays ordenados, se pueden combinar en $O(n)$



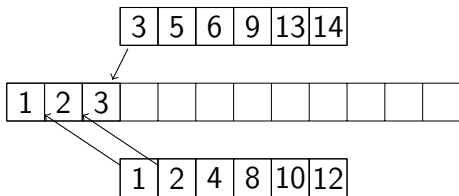
Por mezcla / Mergesort

- Hasta ahora vimos 3 algoritmos cuadráticos. Cambiemos de enfoque.
- Idea 1: si tengo dos arrays ordenados, se pueden combinar en $O(n)$



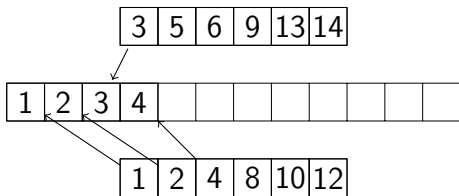
Por mezcla / Mergesort

- Hasta ahora vimos 3 algoritmos cuadráticos. Cambiemos de enfoque.
- Idea 1: si tengo dos arrays ordenados, se pueden combinar en $O(n)$



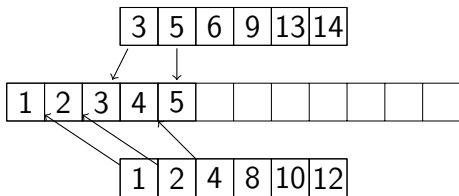
Por mezcla / Mergesort

- Hasta ahora vimos 3 algoritmos cuadráticos. Cambiemos de enfoque.
- Idea 1: si tengo dos arrays ordenados, se pueden combinar en $O(n)$



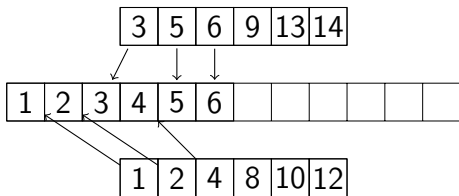
Por mezcla / Mergesort

- Hasta ahora vimos 3 algoritmos cuadráticos. Cambiemos de enfoque.
- Idea 1: si tengo dos arrays ordenados, se pueden combinar en $O(n)$



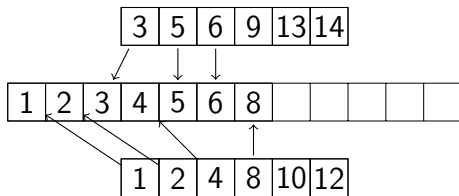
Por mezcla / Mergesort

- Hasta ahora vimos 3 algoritmos cuadráticos. Cambiemos de enfoque.
- Idea 1: si tengo dos arrays ordenados, se pueden combinar en $O(n)$



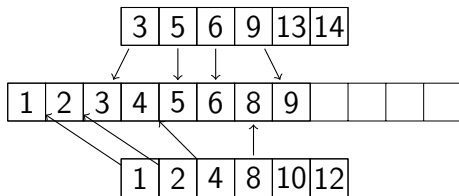
Por mezcla / Mergesort

- Hasta ahora vimos 3 algoritmos cuadráticos. Cambiemos de enfoque.
- Idea 1: si tengo dos arrays ordenados, se pueden combinar en $O(n)$



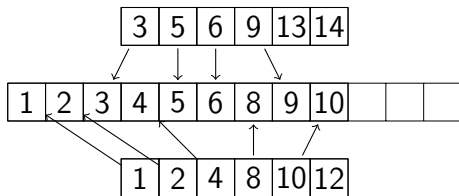
Por mezcla / Mergesort

- Hasta ahora vimos 3 algoritmos cuadráticos. Cambiemos de enfoque.
- Idea 1: si tengo dos arrays ordenados, se pueden combinar en $O(n)$



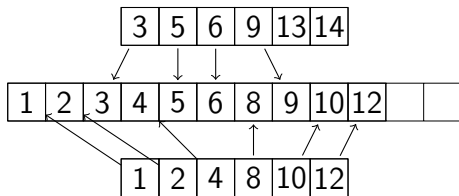
Por mezcla / Mergesort

- Hasta ahora vimos 3 algoritmos cuadráticos. Cambiemos de enfoque.
- Idea 1: si tengo dos arrays ordenados, se pueden combinar en $O(n)$



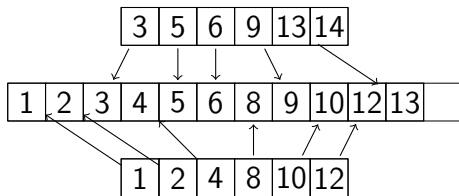
Por mezcla / Mergesort

- Hasta ahora vimos 3 algoritmos cuadráticos. Cambiemos de enfoque.
- Idea 1: si tengo dos arrays ordenados, se pueden combinar en $O(n)$



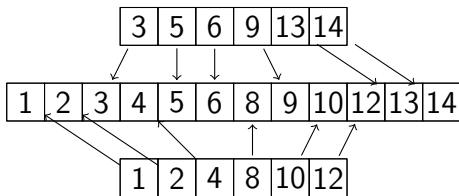
Por mezcla / Mergesort

- Hasta ahora vimos 3 algoritmos cuadráticos. Cambiemos de enfoque.
- Idea 1: si tengo dos arrays ordenados, se pueden combinar en $O(n)$



Por mezcla / Mergesort

- Hasta ahora vimos 3 algoritmos cuadráticos. Cambiemos de enfoque.
- Idea 1: si tengo dos arrays ordenados, se pueden combinar en $O(n)$

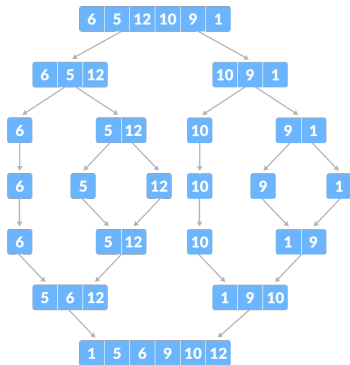


Por mezcla / Mergesort

- Idea 2: Parto mi arreglo en N subarreglos de 1 elemento.
- Todos están ordenados, así que puedo mezclarlos y llegar a uno con todos los elementos

Por mezcla / Mergesort

- Idea 2: Parto mi arreglo en N subarreglos de 1 elemento.
- Todos están ordenados, así que puedo mezclarlos y llegar a uno con todos los elementos



Por mezcla / Mergesort

```
mergesort(A[N]) // Devuelve un array nuevo
  if N < 2 ⇒
    return A
  else ⇒
    m = N/2
    A1 = A[1..m]
    A2 = A[m+1..N]
    B1 = mergesort(A1)
    B2 = mergesort(A2)
    return mezclar(B1, B2)
```

Por mezcla / Mergesort

```
mergesort(A[N]) // Devuelve un array nuevo
  if N < 2 ⇒
    return A
  else ⇒
    m = N/2
    A1 = A[1..m]
    A2 = A[m+1..N]
    B1 = mergesort(A1)
    B2 = mergesort(A2)
    return mezclar(B1, B2)
```

- **Pregunta:** si en vez de partir al medio, partimos en $N - 1$ y 1 ¿cómo queda el algoritmo?

Eficiencia de Mergesort

Sea $W(n)$ el costo en tiempo de mergesort para un array de tamaño n .

Advertencia: prueba muy informal.

Eficiencia de Mergesort

Sea $W(n)$ el costo en tiempo de mergesort para un array de tamaño n .

Advertencia: prueba muy informal.

$$W(n) = 2W(n/2) + O(n)$$

Eficiencia de Mergesort

Sea $W(n)$ el costo en tiempo de mergesort para un array de tamaño n .

Advertencia: prueba muy informal.

$$W(n) = 2W(n/2) + O(n)$$

$$\begin{aligned} & W(n) \\ = & 2W(n/2) + O(n) \end{aligned}$$

Eficiencia de Mergesort

Sea $W(n)$ el costo en tiempo de mergesort para un array de tamaño n .

Advertencia: prueba muy informal.

$$W(n) = 2W(n/2) + O(n)$$

$$\begin{aligned} & W(n) \\ = & 2W(n/2) + O(n) \\ = & 2[2W(n/4) + O(n/2)] + O(n) \end{aligned}$$

Eficiencia de Mergesort

Sea $W(n)$ el costo en tiempo de mergesort para un array de tamaño n .

Advertencia: prueba muy informal.

$$W(n) = 2W(n/2) + O(n)$$

$$\begin{aligned} & W(n) \\ = & 2W(n/2) + O(n) \\ = & 2[2W(n/4) + O(n/2)] + O(n) \\ = & 4W(n/4) + O(n) + O(n) \end{aligned}$$

Eficiencia de Mergesort

Sea $W(n)$ el costo en tiempo de mergesort para un array de tamaño n .

Advertencia: prueba muy informal.

$$W(n) = 2W(n/2) + O(n)$$

$$\begin{aligned} & W(n) \\ = & 2W(n/2) + O(n) \\ = & 2[2W(n/4) + O(n/2)] + O(n) \\ = & 4W(n/4) + O(n) + O(n) \\ = & 8W(n/8) + O(n) + O(n) + O(n) \\ = & \dots \end{aligned}$$

Eficiencia de Mergesort

Sea $W(n)$ el costo en tiempo de mergesort para un array de tamaño n .

Advertencia: prueba muy informal.

$$W(n) = 2W(n/2) + O(n)$$

$$\begin{aligned} & W(n) \\ = & 2W(n/2) + O(n) \\ = & 2[2W(n/4) + O(n/2)] + O(n) \\ = & 4W(n/4) + O(n) + O(n) \\ = & 8W(n/8) + O(n) + O(n) + O(n) \\ = & \dots \\ = & nW(1) + \lg_2 n * O(n) \end{aligned}$$

Eficiencia de Mergesort

Sea $W(n)$ el costo en tiempo de mergesort para un array de tamaño n .

Advertencia: prueba muy informal.

$$W(n) = 2W(n/2) + O(n)$$

$$\begin{aligned} & W(n) \\ = & 2W(n/2) + O(n) \\ = & 2[2W(n/4) + O(n/2)] + O(n) \\ = & 4W(n/4) + O(n) + O(n) \\ = & 8W(n/8) + O(n) + O(n) + O(n) \\ = & \dots \\ = & nW(1) + \lg_2 n * O(n) \end{aligned}$$

$$W(n) = O(n \lg n)$$

- Asintóticamente mejor que los anteriores.

Eficiencia de Mergesort

Sea $W(n)$ el costo en tiempo de mergesort para un array de tamaño n .

Advertencia: prueba muy informal.

$$W(n) = 2W(n/2) + O(n)$$

$$\begin{aligned} & W(n) \\ = & 2W(n/2) + O(n) \\ = & 2[2W(n/4) + O(n/2)] + O(n) \\ = & 4W(n/4) + O(n) + O(n) \\ = & 8W(n/8) + O(n) + O(n) + O(n) \\ = & \dots \\ = & nW(1) + \lg_2 n * O(n) \end{aligned}$$

$$W(n) = O(n \lg n)$$

- Asintóticamente mejor que los anteriores.
- Uso de memoria:

Eficiencia de Mergesort

Sea $W(n)$ el costo en tiempo de mergesort para un array de tamaño n .

Advertencia: prueba muy informal.

$$W(n) = 2W(n/2) + O(n)$$

$$\begin{aligned} & W(n) \\ = & 2W(n/2) + O(n) \\ = & 2[2W(n/4) + O(n/2)] + O(n) \\ = & 4W(n/4) + O(n) + O(n) \\ = & 8W(n/8) + O(n) + O(n) + O(n) \\ = & \dots \\ = & nW(1) + \lg_2 n * O(n) \end{aligned}$$

$$W(n) = O(n \lg n)$$

- Asintóticamente mejor que los anteriores.
- Uso de memoria: $O(n)$.

Quicksort

- Idea: elijo un “pivote” p y separo el array en los $\leq p$ y $> p$.

Quicksort

- Idea: elijo un “pivote” p y separo el array en los $\leq p$ y $> p$.
- Ordeno recursivamente las dos mitades

7	2	1	9	3	8	5
----------	---	---	---	---	---	---

Quicksort

- Idea: elijo un “pivote” p y separo el array en los $\leq p$ y $> p$.
- Ordeno recursivamente las dos mitades

7	2	1	9	3	8	5
----------	---	---	---	---	---	---

7	2	1	5	3	8	9
----------	---	---	---	---	---	---

Quicksort

- Idea: elijo un “pivote” p y separo el array en los $\leq p$ y $> p$.
- Ordeno recursivamente las dos mitades

7	2	1	9	3	8	5
---	---	---	---	---	---	---

7	2	1	5	3	8	9
---	---	---	---	---	---	---

Quicksort

- Idea: elijo un “pivote” p y separo el array en los $\leq p$ y $> p$.
- Ordeno recursivamente las dos mitades

7	2	1	9	3	8	5
---	---	---	---	---	---	---

7	2	1	5	3	8	9
---	---	---	---	---	---	---

1	2	3	5	7	8	9
---	---	---	---	---	---	---

Particionar

- Hay varias formas...

Particionar

- Hay varias formas...
- Llevamos una parte $\leq p$ y otra $> p$.

$\leq p$	$> p$	$a[i]$...
----------	-------	--------	-----

Particionar

- Hay varias formas...
- Llevamos una parte $\leq p$ y otra $> p$.

$\leq p$	$> p$	$a[i]$...
----------	-------	--------	-----

$a[i] > p?$

$\leq p$	$> p, a[i]$...
----------	-------------	-----

Particionar

- Hay varias formas...
- Llevamos una parte $\leq p$ y otra $> p$.

$\leq p$	$> p$	$a[i]$...
----------	-------	--------	-----

$a[i] > p?$

$\leq p$	$> p, a[i]$...
----------	-------------	-----

$a[i] \leq p?$

$\leq p, a[i]$	$> p$...
----------------	-------	-----

Quicksort

```
qsort(A[N])  
  if N < 2  $\Rightarrow$  return // no hacer nada  
  p = A[1] // primer elem  
  nizq = 1 + particionar(A[2..N], p) // nizq = cant  $\leq$   
  A[1]  $\leftrightarrow$  A[nizq] // mover el pivote a la frontera  
  qsort(A[1..nizq-1]) // parte izq  
  qsort(A[nizq+1..N]) // parte der
```

```
// Partición de Lomuto  
// Devuelve cant de elementos  $\leq p$   
particionar(A[N], p)  
  j = 0  
  for i in 1..N  $\Rightarrow$   
    if A[i]  $\leq$  p  
      A[i]  $\leftrightarrow$  A[j]  
      j++  
  return j
```

Quicksort

```
qsort(A[N])
```

```
  if N < 2  $\Rightarrow$  return // no hacer nada
```

```
  p = A[1] // primer elem
```

```
  nizq = 1 + particionar(A[2..N], p) // nizq = cant  $\leq$ 
```

```
  A[1]  $\leftrightarrow$  A[nizq] // mover el pivote a la frontera
```

```
  qsort(A[1..nizq-1]) // parte izq
```

```
  qsort(A[nizq+1..N]) // parte der
```

- Tiempo de ejecución:

```
// Partición de Lomuto
```

```
// Devuelve cant de elementos  $\leq p$ 
```

```
particionar(A[N], p)
```

```
  j = 0
```

```
  for i in 1..N  $\Rightarrow$ 
```

```
    if A[i]  $\leq$  p
```

```
      A[i]  $\leftrightarrow$  A[j]
```

```
      j++
```

```
  return j
```

Quicksort

```
qsort(A[N])
```

```
  if N < 2 ⇒ return // no hacer nada
```

```
  p = A[1] // primer elem
```

```
  nizq = 1 + particionar(A[2..N], p) // nizq = mucho
```

```
  A[1] ↔ A[nizq] // mover el pivote a la frontera
```

```
  qsort(A[1..nizq-1]) // parte izq
```

```
  qsort(A[nizq+1..N]) // parte der
```

- Tiempo de ejecución: depende mucho de la elección del pivote. Peor caso: $O(n^2)$ (cuando el arreglo ya estaba ordenado!).

```
// Partición de Lomuto
```

```
// Devuelve cant de elementos  $\leq p$ 
```

```
particionar(A[N], p)
```

```
  j = 0
```

```
  for i in 1..N ⇒
```

```
    if A[i] ≤ p
```

```
      A[i] ↔ A[j]
```

```
      j++
```

```
  return j
```

Quicksort

```
qsort(A[N])
```

```
  if N < 2 ⇒ return // no hacer nada
```

```
  p = A[1] // primer elem
```

```
  nizq = 1 + particionar(A[2..N], p) // nizq = mucho < de la elección del
```

```
  A[1] ↔ A[nizq] // mover el pivote a la frontera
```

```
  qsort(A[1..nizq-1]) // parte izq
```

```
  qsort(A[nizq+1..N]) // parte der
```

```
// Partición de Lomuto
```

```
// Devuelve cant de elementos  $\leq p$ 
```

```
particionar(A[N], p)
```

```
  j = 0
```

```
  for i in 1..N ⇒
```

```
    if A[i] ≤ p
```

```
      A[i] ↔ A[j]
```

```
      j++
```

```
  return j
```

- Tiempo de ejecución: depende **mucho** de la elección del pivote. Peor caso: $O(n^2)$ **(cuando el arreglo ya estaba ordenado!)**.
- Si el pivote corta “cerca” del medio: $O(n \lg n)$

Quicksort

```
qsort(A[N])
```

```
  if N < 2 ⇒ return // no hacer nada
```

```
  p = A[1] // primer elem
```

```
  nizq = 1 + particionar(A[2..N], p) // nizq = mucho
```

```
  A[1] ↔ A[nizq] // mover el pivote a la frontera
```

```
  qsort(A[1..nizq-1]) // parte izq
```

```
  qsort(A[nizq+1..N]) // parte der
```

```
// Partición de Lomuto
```

```
// Devuelve cant de elementos  $\leq p$ 
```

```
particionar(A[N], p)
```

```
  j = 0
```

```
  for i in 1..N ⇒
```

```
    if A[i] ≤ p
```

```
      A[i] ↔ A[j]
```

```
      j++
```

```
  return j
```

- Tiempo de ejecución: depende mucho de la elección del pivote. Peor caso: $O(n^2)$ (cuando el arreglo ya estaba ordenado!).

- Si el pivote corta “cerca” del medio: $O(n \lg n)$
- Memoria:

Quicksort

```
qsort(A[N])
```

```
  if N < 2 ⇒ return // no hacer nada
```

```
  p = A[1] // primer elem
```

```
  nizq = 1 + particionar(A[2..N], p) // nizq = mucho < de la elección del
```

```
  A[1] ↔ A[nizq] // mover el pivote a la frontera
```

```
  qsort(A[1..nizq-1]) // parte izq
```

```
  qsort(A[nizq+1..N]) // parte der
```

```
// Partición de Lomuto
```

```
// Devuelve cant de elementos  $\leq p$ 
```

```
particionar(A[N], p)
```

```
  j = 0
```

```
  for i in 1..N ⇒
```

```
    if A[i] ≤ p
```

```
      A[i] ↔ A[j]
```

```
      j++
```

```
  return j
```

- Tiempo de ejecución: depende **mucho** de la elección del pivote. Peor caso: $O(n^2)$ **(cuando el arreglo ya estaba ordenado!)**.
- Si el pivote corta “cerca” del medio: $O(n \lg n)$
- Memoria: $O(\lg n)$ de las llamadas recursivas.. pero nada más. No copia el array.

Detalles de Quicksort

- La elección del pivote es importante. Una solución es tomarlo al azar.
- Hay que tener cuidado con los repetidos: también pueden causar costo cuadrático.

Detalles de Quicksort

- La elección del pivote es importante. Una solución es tomarlo al azar.
- Hay que tener cuidado con los repetidos: también pueden causar costo cuadrático.
- Mejor forma de particionar: esquema de Hoare.

Detalles de Quicksort

- La elección del pivote es importante. Una solución es tomarlo al azar.
- Hay que tener cuidado con los repetidos: también pueden causar costo cuadrático.
- Mejor forma de particionar: esquema de Hoare.
- Así y todo... es de lo más eficiente que hay

¿Podemos mejorar?

Cota óptima

- ¿Será óptima la cota de $O(n \lg n)$?

Cota óptima

- ¿Será óptima la cota de $O(n \lg n)$?
- Podemos modelar **cualquier** algoritmo como un árbol de decisión. El algoritmo tiene que encontrar en cual “permutación” está.

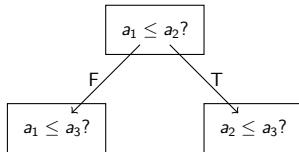
Cota óptima

- ¿Será óptima la cota de $O(n \lg n)$?
- Podemos modelar **cualquier** algoritmo como un árbol de decisión. El algoritmo tiene que encontrar en cual “permutación” está.

$$a_1 \leq a_2?$$

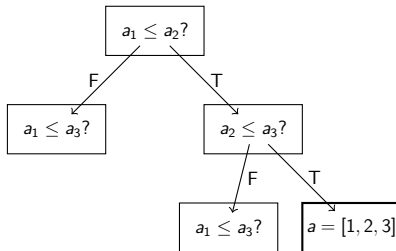
Cota óptima

- ¿Será óptima la cota de $O(n \lg n)$?
- Podemos modelar **cualquier** algoritmo como un árbol de decisión. El algoritmo tiene que encontrar en cual “permutación” está.



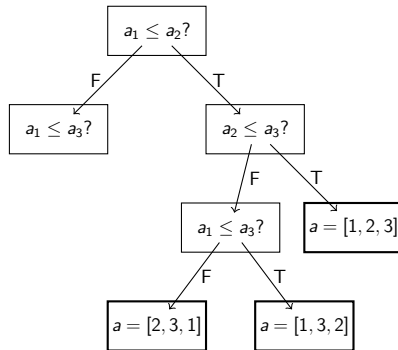
Cota óptima

- ¿Será óptima la cota de $O(n \lg n)$?
- Podemos modelar **cualquier** algoritmo como un árbol de decisión. El algoritmo tiene que encontrar en cual “permutación” está.



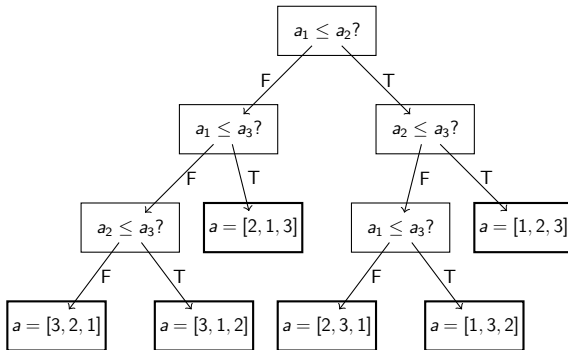
Cota óptima

- ¿Será óptima la cota de $O(n \lg n)$?
- Podemos modelar **cualquier** algoritmo como un árbol de decisión. El algoritmo tiene que encontrar en cual “permutación” está.



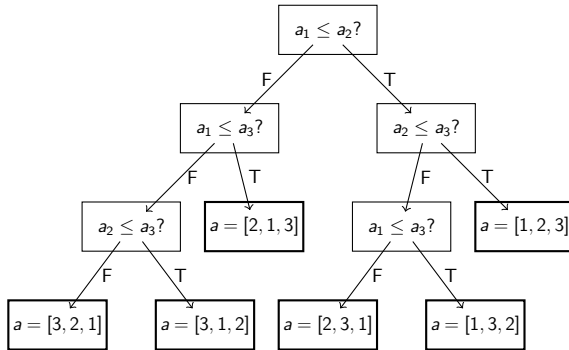
Cota óptima

- ¿Será óptima la cota de $O(n \lg n)$?
- Podemos modelar **cualquier** algoritmo como un árbol de decisión. El algoritmo tiene que encontrar en cual “permutación” está.



Cota óptima

- ¿Será óptima la cota de $O(n \lg n)$?
- Podemos modelar **cualquier** algoritmo como un árbol de decisión. El algoritmo tiene que encontrar en cual “permutación” está.



- Mejores casos = Caminos cortos. Peores casos = Caminos largos.

Cota óptima

- Si el camino más largo es de h pasos, como mucho tenemos 2^h nodos.

Cota óptima

- Si el camino más largo es de h pasos, como mucho tenemos 2^h nodos.
- Pero tenemos que distinguir $n!$ posibilidades distintas (todas las permutaciones).

Cota óptima

- Si el camino más largo es de h pasos, como mucho tenemos 2^h nodos.
- Pero tenemos que distinguir $n!$ posibilidades distintas (todas las permutaciones).

$$2^h \geq n!$$

Cota óptima

- Si el camino más largo es de h pasos, como mucho tenemos 2^h nodos.
- Pero tenemos que distinguir $n!$ posibilidades distintas (todas las permutaciones).

$$2^h \geq n! \implies h \geq \ln(n!)$$

Cota óptima

- Si el camino más largo es de h pasos, como mucho tenemos 2^h nodos.
- Pero tenemos que distinguir $n!$ posibilidades distintas (todas las permutaciones).

$$2^h \geq n! \implies h \geq \ln(n!) \implies h \geq n \ln n - n$$

(Resulta que $\ln(n!) \approx n \ln n$; ver Aproximación de Stirling)

- Entonces, h (peor caso) es al menos $O(n \ln n)$.

Para profundizar

Otras propiedades que puede tener un algoritmo de ordenamiento:

- *Adaptivo*: aprovecha orden de la entrada.

Para profundizar

Otras propiedades que puede tener un algoritmo de ordenamiento:

- *Adaptivo*: aprovecha orden de la entrada. Ej: inserción (mejor caso $O(n)$).

Para profundizar

Otras propiedades que puede tener un algoritmo de ordenamiento:

- *Adaptivo*: aprovecha orden de la entrada. Ej: inserción (mejor caso $O(n)$).
- *Estable*: no mezcla elementos “iguales”/incomparables.

Para profundizar

Otras propiedades que puede tener un algoritmo de ordenamiento:

- *Adaptivo*: aprovecha orden de la entrada. Ej: inserción (mejor caso $O(n)$).
- *Estable*: no mezcla elementos “iguales”/incomparables. Ej: selección, inserción, mergesort (pero en todos hay que tener cuidado). En general el sort de un lenguaje no es estable (excepción: Python). Algunos proveen también una versión estable (ej: C++, Java).

Para profundizar

Otras propiedades que puede tener un algoritmo de ordenamiento:

- *Adaptivo*: aprovecha orden de la entrada. Ej: inserción (mejor caso $O(n)$).
- *Estable*: no mezcla elementos “iguales”/incomparables. Ej: selección, inserción, mergesort (pero en todos hay que tener cuidado). En general el sort de un lenguaje no es estable (excepción: Python). Algunos proveen también una versión estable (ej: C++, Java).
- *Online*: puede trabajar a medida que recibe el array.

Para profundizar

Otras propiedades que puede tener un algoritmo de ordenamiento:

- *Adaptivo*: aprovecha orden de la entrada. Ej: inserción (mejor caso $O(n)$).
- *Estable*: no mezcla elementos “iguales”/incomparables. Ej: selección, inserción, mergesort (pero en todos hay que tener cuidado). En general el sort de un lenguaje no es estable (excepción: Python). Algunos proveen también una versión estable (ej: C++, Java).
- *Online*: puede trabajar a medida que recibe el array. Ej: inserción.

Para profundizar

Otras propiedades que puede tener un algoritmo de ordenamiento:

- *Adaptivo*: aprovecha orden de la entrada. Ej: inserción (mejor caso $O(n)$).
- *Estable*: no mezcla elementos “iguales”/incomparables. Ej: selección, inserción, mergesort (pero en todos hay que tener cuidado). En general el sort de un lenguaje no es estable (excepción: Python). Algunos proveen también una versión estable (ej: C++, Java).
- *Online*: puede trabajar a medida que recibe el array. Ej: inserción.
- *In-place*: trabaja sobre el mismo array sin copiarlo.

Para profundizar

Otras propiedades que puede tener un algoritmo de ordenamiento:

- *Adaptivo*: aprovecha orden de la entrada. Ej: inserción (mejor caso $O(n)$).
- *Estable*: no mezcla elementos “iguales”/incomparables. Ej: selección, inserción, mergesort (pero en todos hay que tener cuidado). En general el sort de un lenguaje no es estable (excepción: Python). Algunos proveen también una versión estable (ej: C++, Java).
- *Online*: puede trabajar a medida que recibe el array. Ej: inserción.
- *In-place*: trabaja sobre el mismo array sin copiarlo. Ej: todos menos mergesort.

Otros

- Por comparación: Shell sort, Heapsort
- Para enteros: Counting sort, Radix sort