¿Qué tan rápido podemos ordenar?

Todos los algoritmos de clasificación que hemos visto hasta ahora están basado en comparaciones: Es decir , solo utilizamos comparaciones para determinar el orden relativo de los elementos.

- insertion sort, merge sort, quicksort, heapsort.

El mejor tiempo de ejecución en el peor de los casos es $O(n \lg n)$.

Complejidad temporal

El tiempo mínimo que necesita un algoritmo para resolverlo.

Límite superior:

El problem P puede ser resuelto en $T_{upper}(n)$

Si existe un algoritmo A tal que:

- de la salida correcta
- en máximo ese tiempo

$\exists A, \forall I, \; A(I)=P(I) \quad y \; Tiempo(A,I) \leq T_{upper}(|I|)$

Eg: Para el caso de ordenamiento el tiempo es $T_{upper}(n) = O(n^2)$.

Complejidad temporal

El tiempo mínimo que necesita un algoritmo para resolverlo.

Limite inferior :

$T_{lower}(n)$ es un limite inferior

Si no existe un algorimto que pueda resolver P en un tiempo menor.

Puede haber algoritmos que den la respuesta correcta o se ejecuten rápidamente en alguna instancia de entrada.

Complejidad temporal

El tiempo mínimo que necesita un algoritmo para resolverlo.

Límite inferior:

$T_{upper}(n)$ es un límite inferior P

Si no existe un algorimto que pueda resolve P en un tiempo menor.

Pero para cada algoritmo, hay al menos una instancia I para la cual el algoritmo da una respuesta incorrecta o se ejecuta en demasiado tiempo.

$\forall A, \exists I, A(I) = P(I)$ o $Tiempo(A,I) \geq T_{lower}(|I|)$

No existe un algoritmo que ordene N elementos en $T_{lower} = \sqrt{(N)}$

Complejidad temporal

El tiempo mínimo que necesita un algoritmo para resolverlo.

Limite superior:

$$\exists\, A, \forall\, I,\ A(I)=P(I) \quad y \quad Tiempo(A,I) \leq T_{upper}(|I|)$$

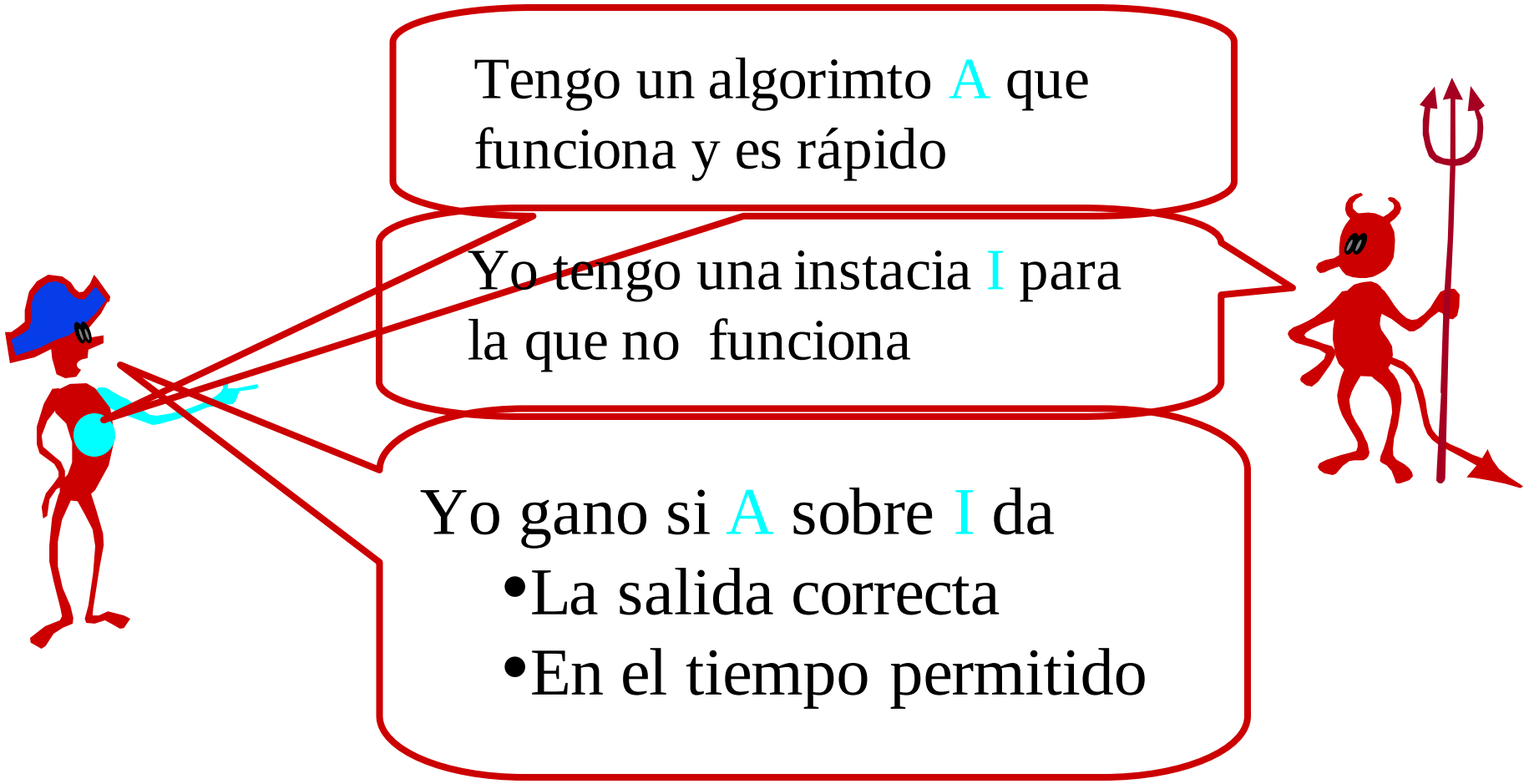
Límite inferior:

$$\forall\, A, \exists\, I,\ A(I) = P(I) \quad o \quad Tiempo(A,I) \geq T_{lower}(|I|)$$

Probador-Adversario

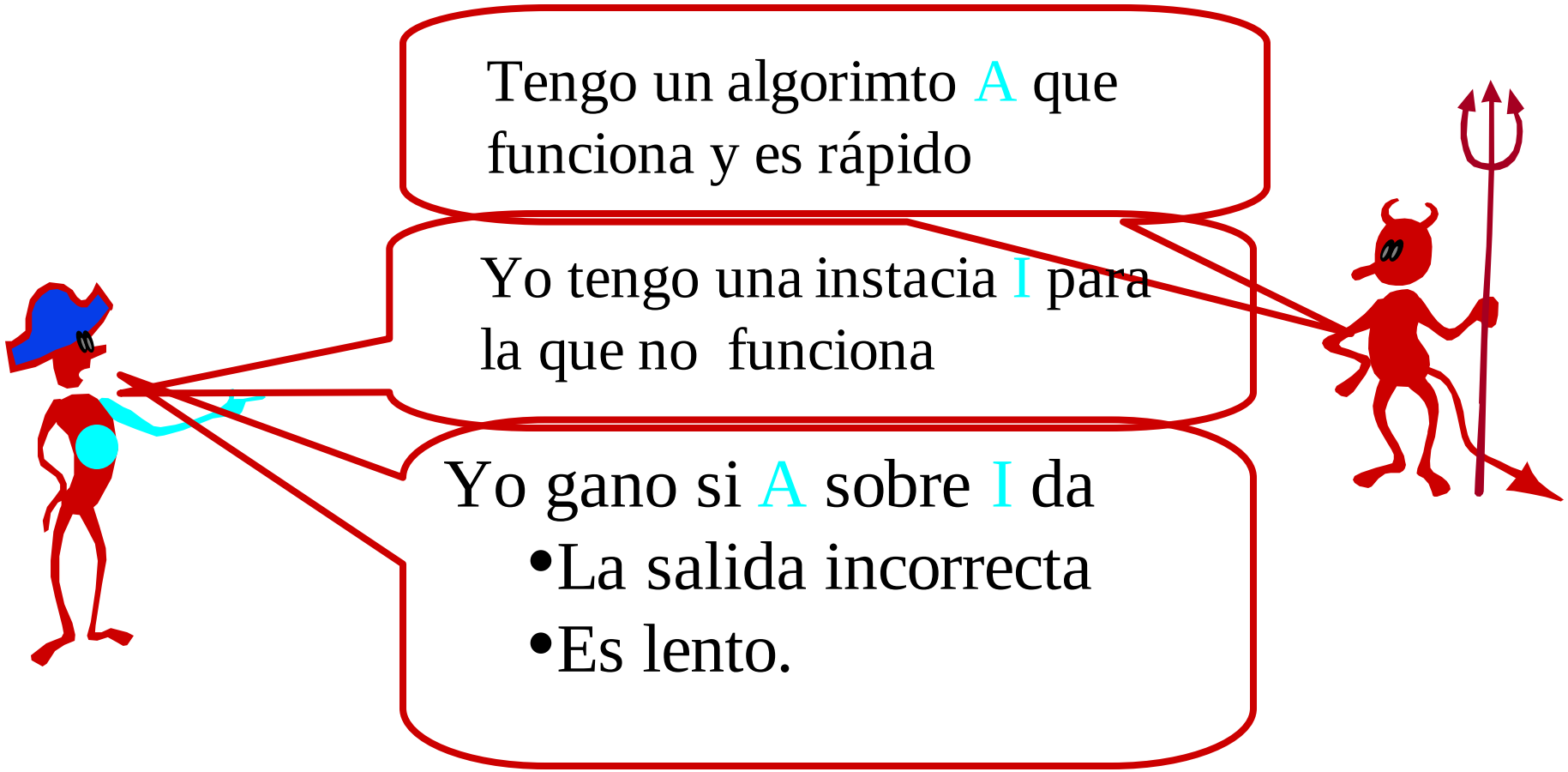
Límite superior:

$\exists$ A, $\forall$ I, A(I)=P(I) y Tiempo(A,I) $\leq T_{upper}(|I|)$



Probador-Adversario

Límite inferior:

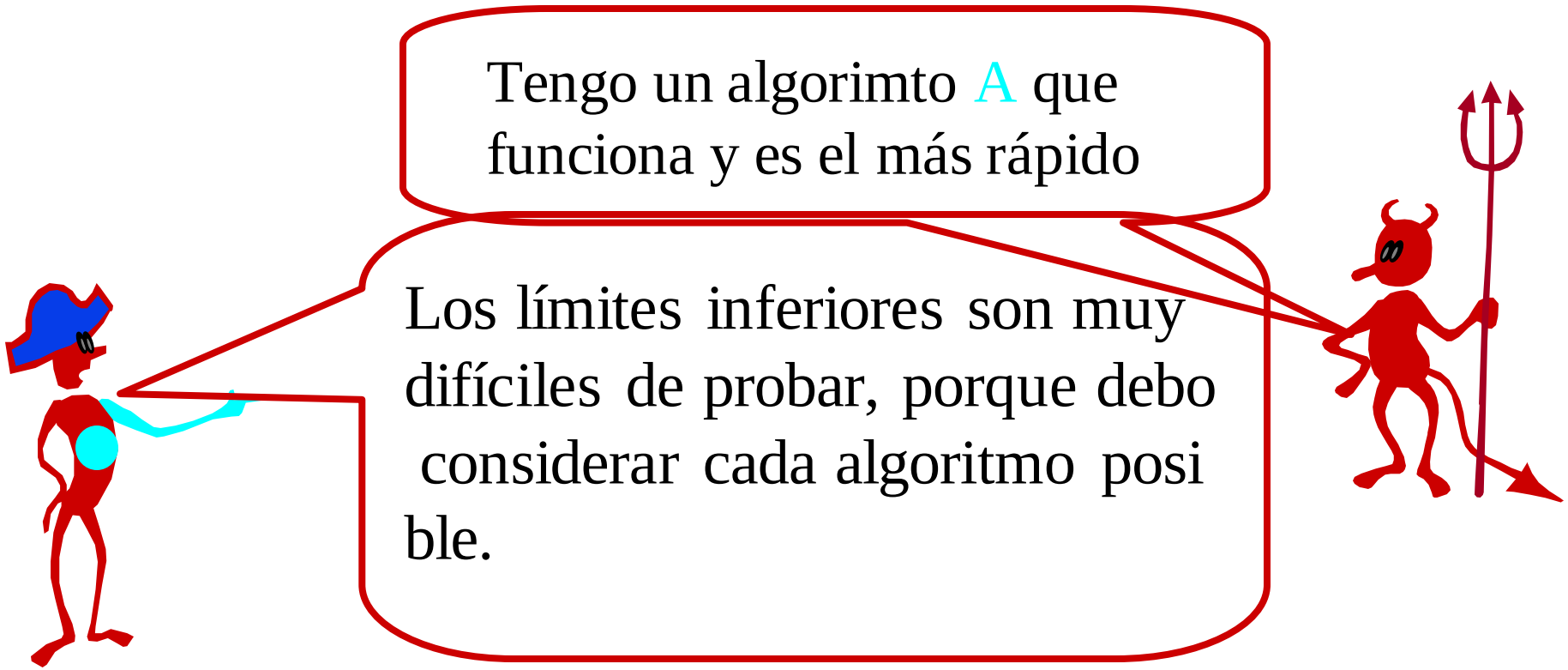


Prueba por contradición.

Probador-Adversario

Límite inferior

$$\forall\ A, \exists\ I, [\ A(I) = P(I)\ \ o\ \ Tiempo(A,I) \geq T_{lower}(|I|)]$$

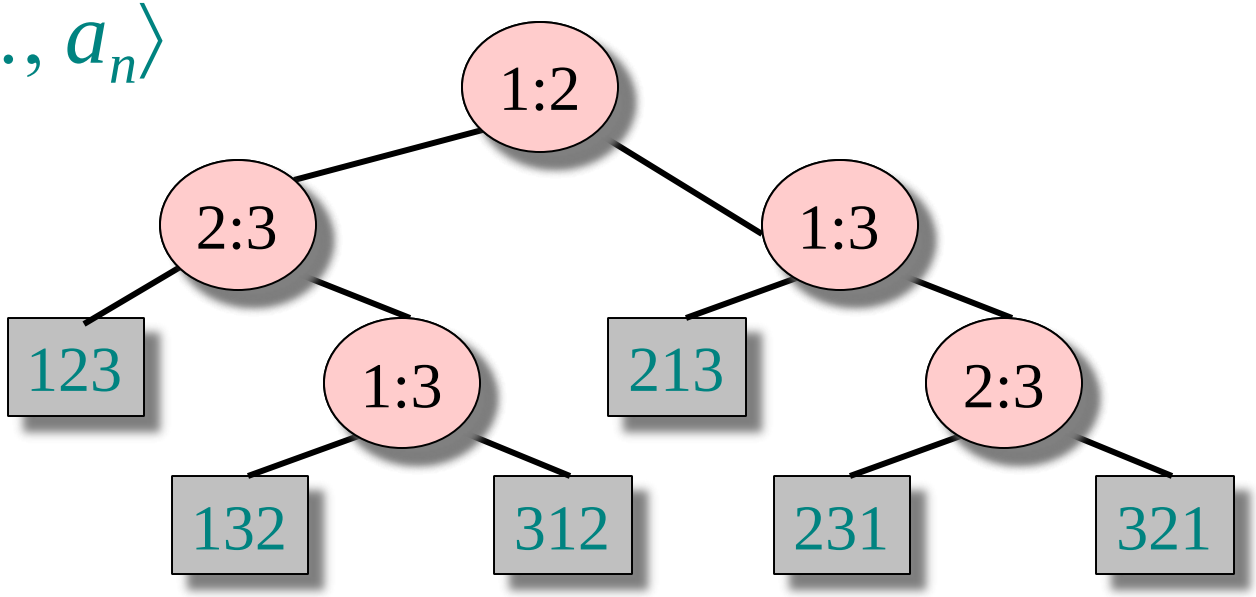


Probar el límite inferior para cualquier algitmo de ordenamiento basado en comparaciones.

Árboles de decisión

Árboles de decisión

Sort$\langle a_1, a_2, \ldots, a_n \rangle$

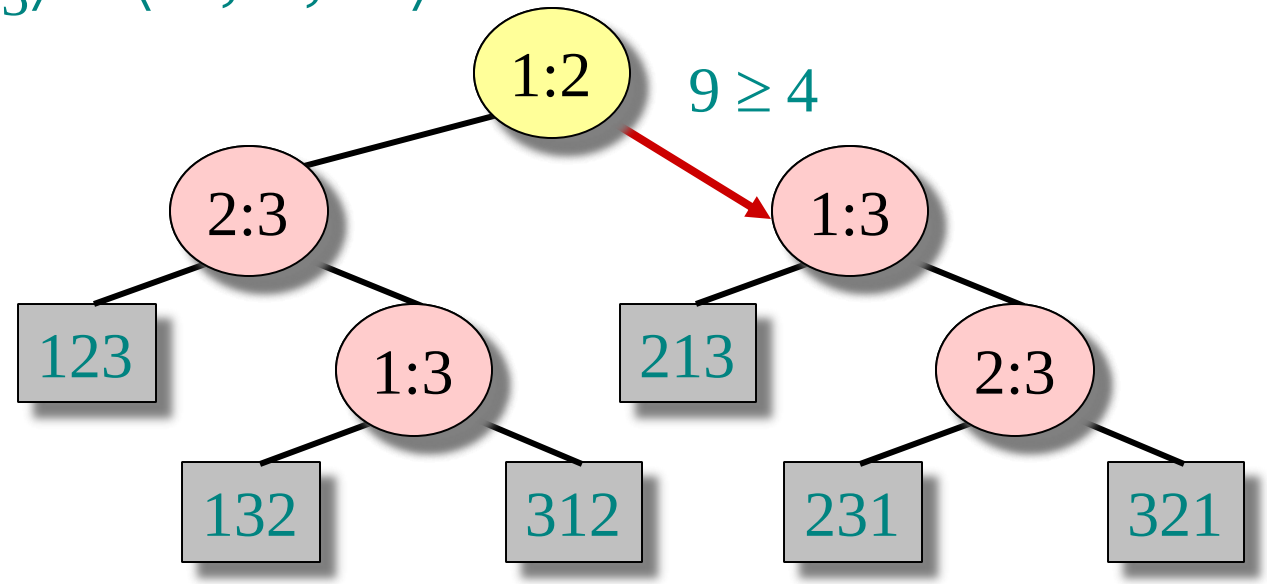


Cada nodo interno es etiquetado $i{:}j$ para $i, j \in \{1, 2, \ldots, n\}$.

- El árbol derecho muestra las comparaciónes donde $a_i \leq a_j$.
- El árbol derecho las comparaciones donde $a_i \geq a_j$.

Árboles de decisión

Sort $\langle a_1, a_2, a_3 \rangle = \langle 9, 4, 6 \rangle$:

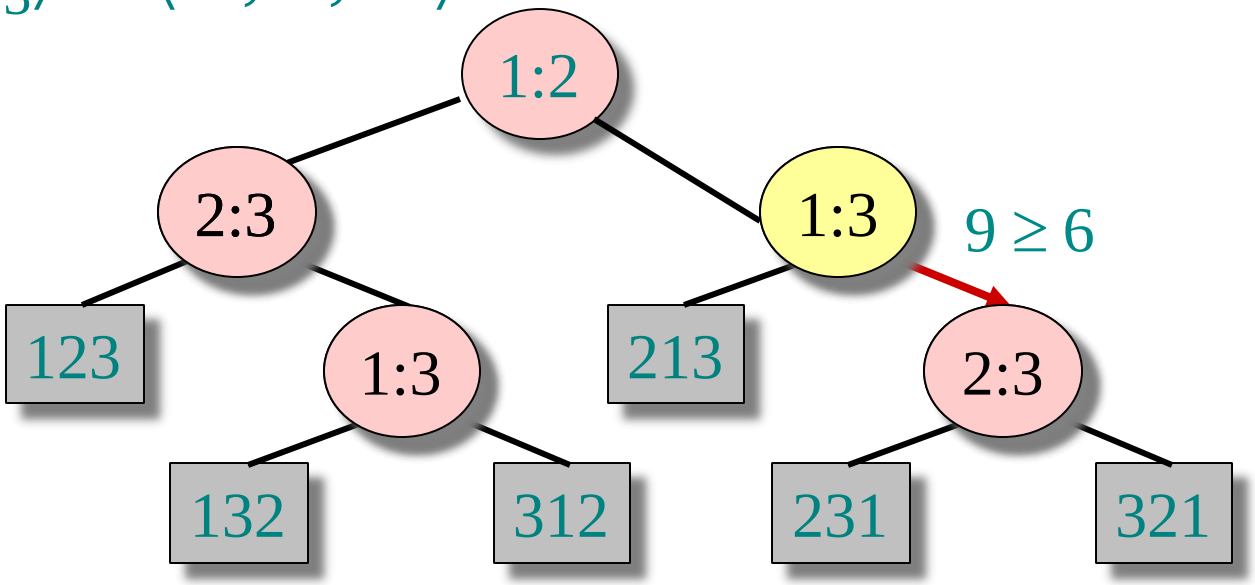


Cada nodo interno es etiquetado $i{:}j$ para $i, j \in \{1, 2, \ldots, n\}$.

- El árbol derecho muestra las comparaciónes donde $ai \leq aj$.
- El árbol derecho las comparaciones donde $ai \geq aj$.

Árboles de decisión

Sort $\langle a_1, a_2, a_3 \rangle = \langle\ 9, 4, 6\ \rangle$:

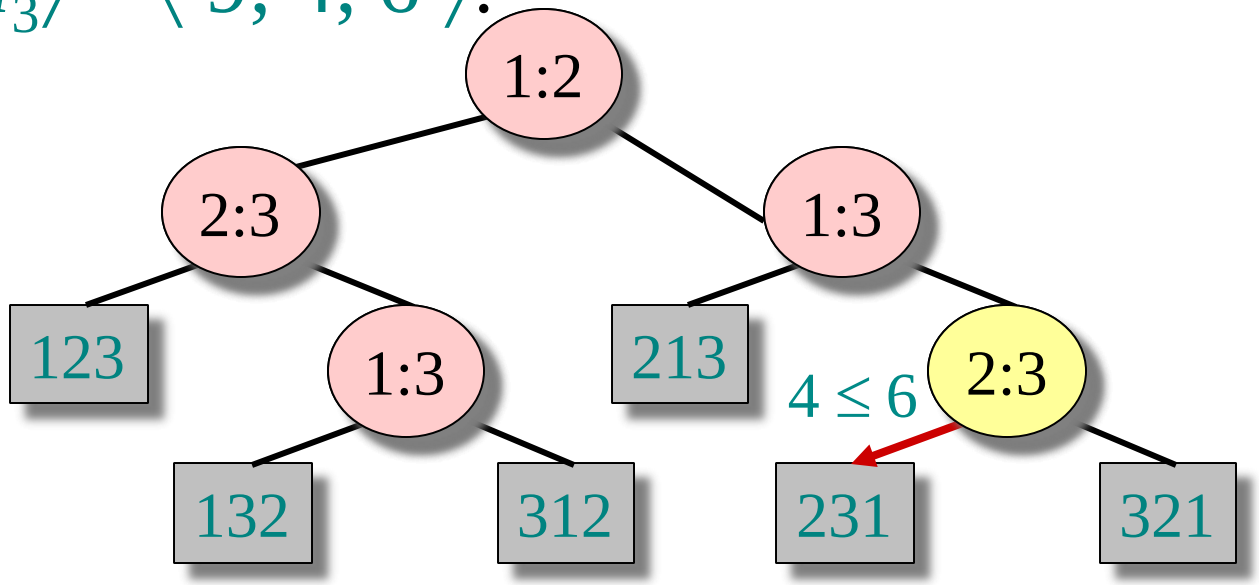


Cada nodo interno es etiquetado $i{:}j$ para $i, j \in \{1, 2, \ldots, n\}$.

- El árbol derecho muestra las comparaciónes donde $ai \leq aj$.
- El árbol derecho las comparaciones donde $ai \geq aj$.

Árboles de decisión

Sort $\langle a_1, a_2, a_3 \rangle = \langle\ 9, 4, 6\ \rangle$:

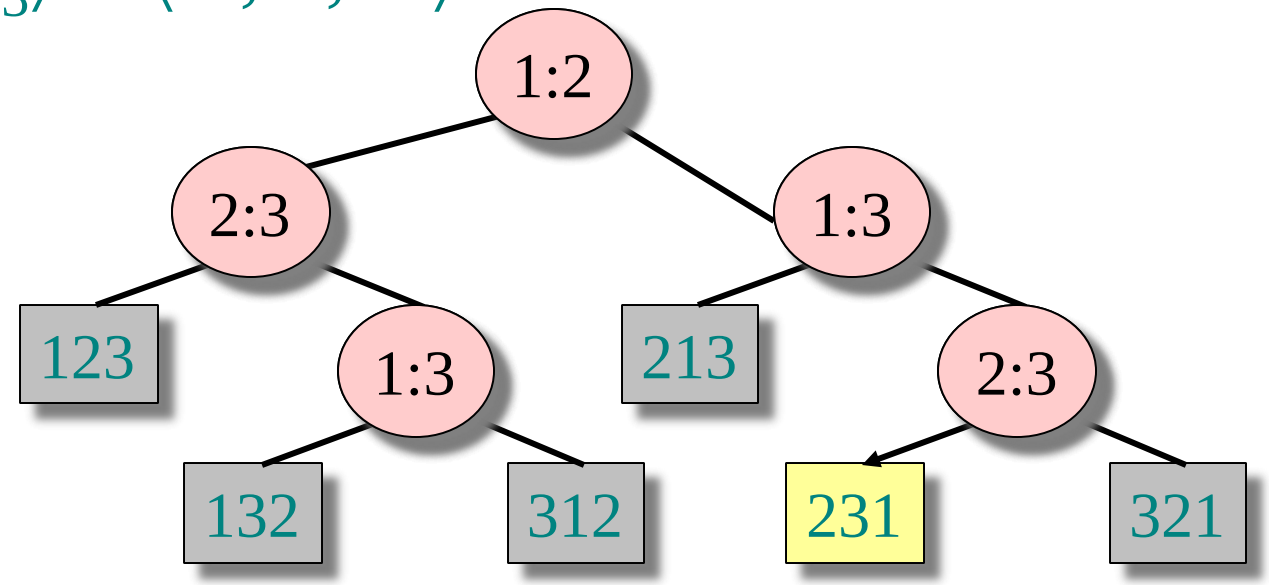


Cada nodo interno es etiquetado $i{:}j$ para $i, j \in \{1, 2, \ldots, n\}$.

- El árbol derecho muestra las comparaciónes donde $ai \leq aj$.
- El árbol derecho las comparaciones donde $ai \geq aj$.

Árboles de decisión

Sort $\langle a_1, a_2, a_3 \rangle = \langle\ 9, 4, 6\ \rangle$:



Cada hoja conriente una permutación $\langle \pi(1), \pi(2), \ldots, \pi(n) \rangle$ que indica el orden $a_{\pi(1)} \leq a_{\pi(2)} \leq \cdots \leq a_{\pi(n)}$

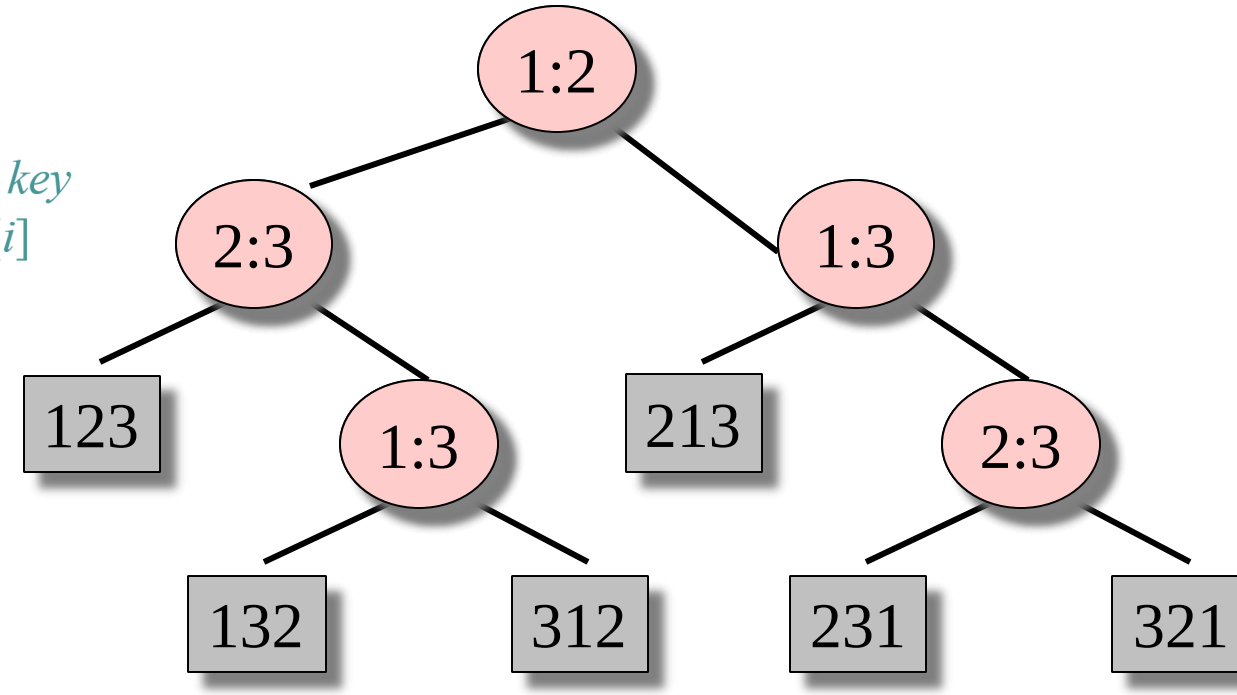
Árboles de decisión

Un árbol de decisiones puede representar la ejecución de cualquier tipo de comparación:

- Un árbol para cada tamaño de entrada n.
- Considere que el problema simpre se divide cuando se parte el problema.
- El árbol contiene todas las posibles ruta de comparación.
- El tiempo del algoritmo será equivalente a la longitud del camino tomado.

Cualquier ordernamiento puede ser Representado como un árbo de decisión

```
INSERTION-SORT (A, n)      ▷ A[1 . . n]
    for j ← 2 to n
        do key ← A[ j]
           i ← j – 1
           while i > 0 and A[i] > key
                 do A[i+1] ← A[i]
                    i ← i – 1
           A[i+1] = key
```



Límite inferior de un algorimito de árbol desición

Teorema. **Cualquier árbol de desicón que pueda ordenar** n **elementos será de altura** $\Omega(n \lg n)$.

Prueba. El árbol contendra $\geq n!$ *hojas, dado que hay* $n!$ *possibles permutaciones. Un árbol binario de altura* h *tiene* $\leq 2^h$ *hojas. Entonces,* $n! \leq 2^h$.

$\therefore \; h \geq \lg(n!)$ (lg es monotonicamente creciente)

$\geq \lg((n/e)^n)$ (Fórmula de Stirling)

$= n \lg n - n \lg e$

$= \Omega(n \lg n)$.

Límite inferior de un algorimito de árbol desición

Corolario. **Heapsort y merge sort son algoritmos de comparación asintóticamente optimos**

Límite inferior para Algoritmos de ordenamiento

¿Existe un algoritm más rápido?

¿Qué tal uno que no use comparaciones?

```
INSERTION-SORT (A, n)      ⊳ A[1 . . n]
    for j ← 2 to n
        do key ← A[ j]
           i ← j – 1
           while i > 0 and A[i] > key
                 do A[i+1] ← A[i]
                    i ← i – 1
           A[i+1] = key
```

Ordenado en tiempo lineal

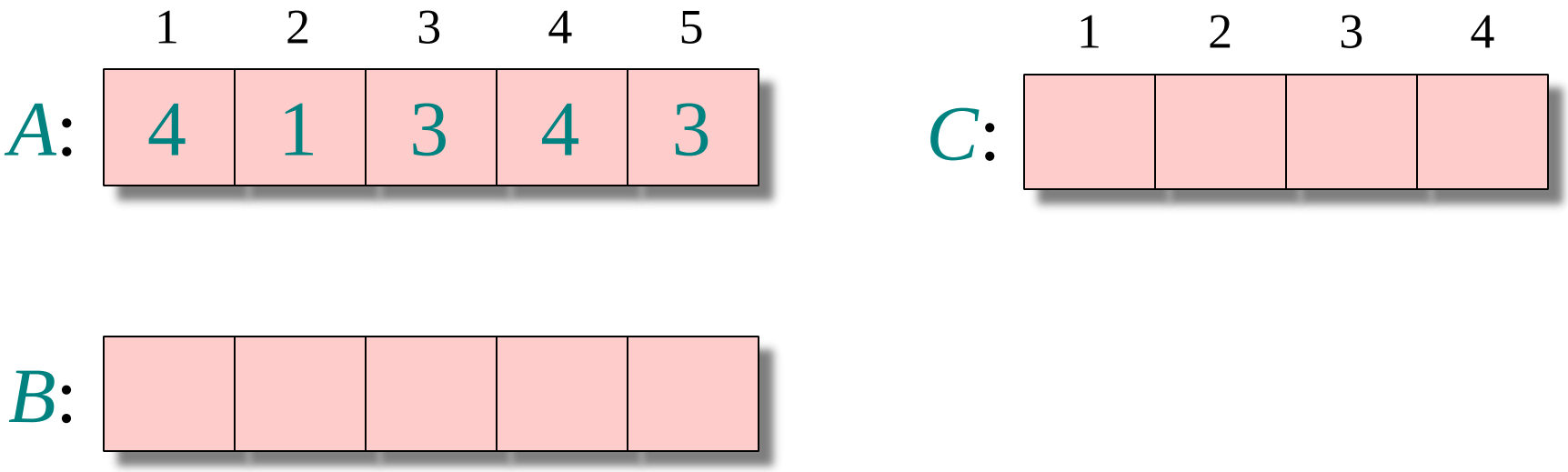
Counting sort: No utliza comparaciones entre elementos

- ***Input***: $A[1 \,..\, n]$, donde $A[j] \in \{1, 2, \ldots, k\}$.
- ***Output***: $B[1 \,..\, n]$, ordenado.
- ***Alamcenamiento auxiliar***: $C[1 \,..\, k]$.

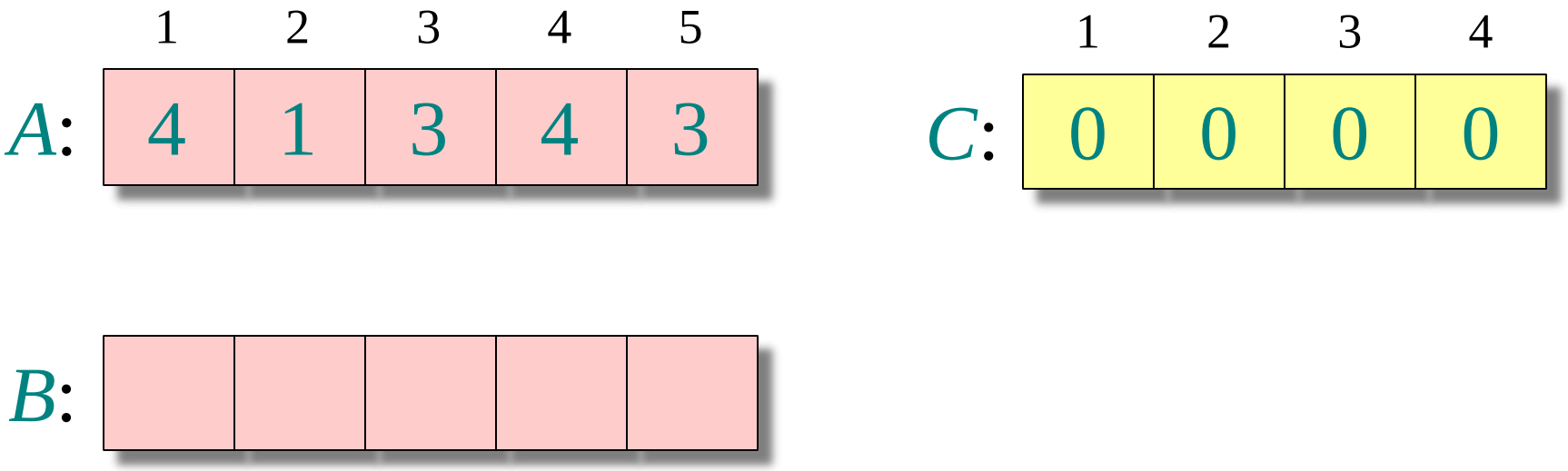
Counting sort

```
for i ← 1 to k
    do C[i] ← 0
for j ← 1 to n
    do C[A[ j]] ← C[A[ j]] + 1      ▷ C[i] = |{key = i}|
for i ← 2 to k
    do C[i] ← C[i] + C[i–1]         ▷ C[i] = |{key ≤ i}|
for j ← n downto 1
    do B[C[A[ j]]] ← A[ j]
       C[A[ j]] ← C[A[ j]] – 1
```

Counting-sort



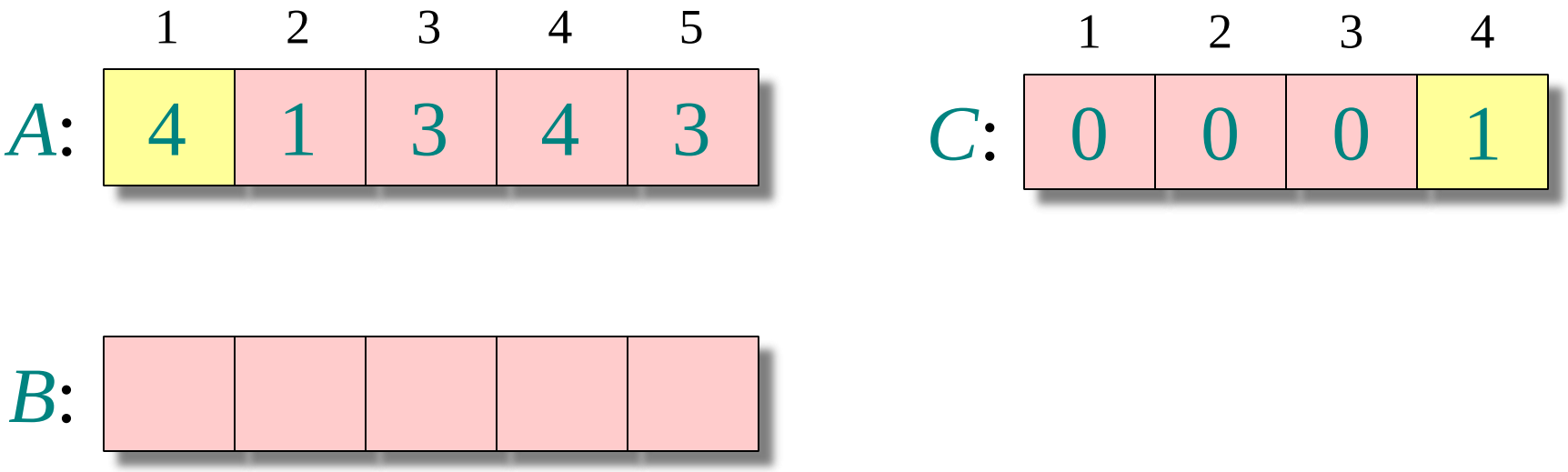
Ciclo 1



for $i \leftarrow 1$ **to** k

do $C[i] \leftarrow 0$

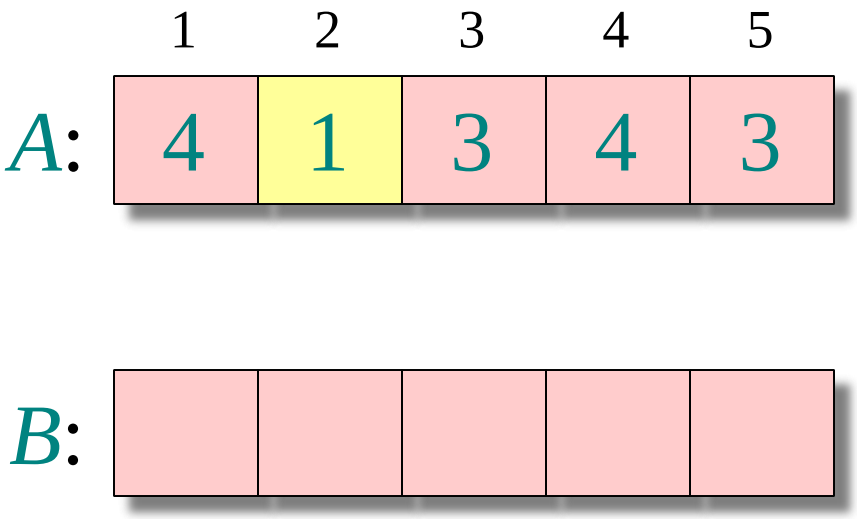
Ciclo 2

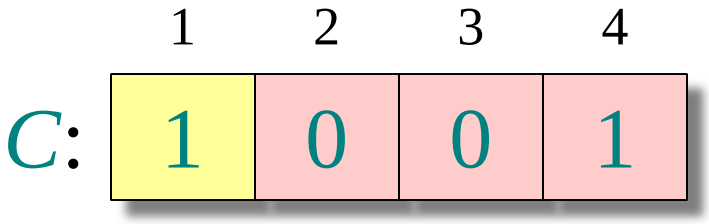


for $j \leftarrow 1$ **to** n

do $C[A[\,j]] \leftarrow C[A[\,j]] + 1$ ▷ $C[i] = |\{\text{key} = i\}|$

Ciclo 2

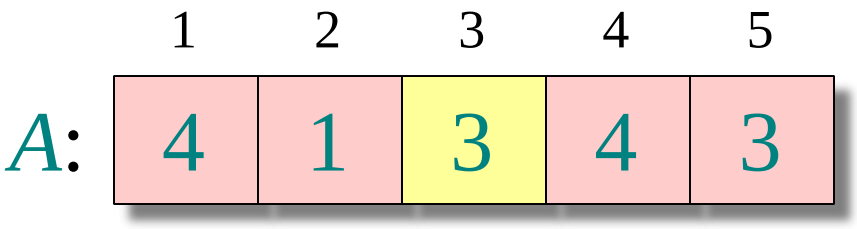


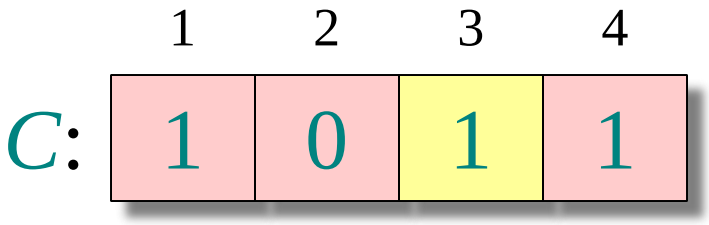


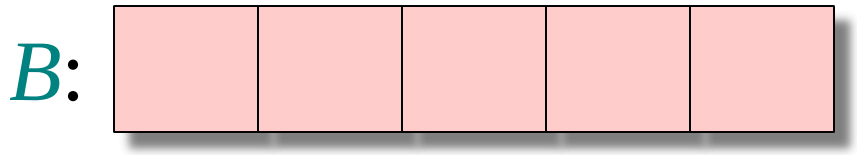
for $j \leftarrow 1$ **to** n

do $C[A[\,j]] \leftarrow C[A[\,j]] + 1$ ▷ $C[i] = |\{\text{key} = i\}|$

Ciclo 2



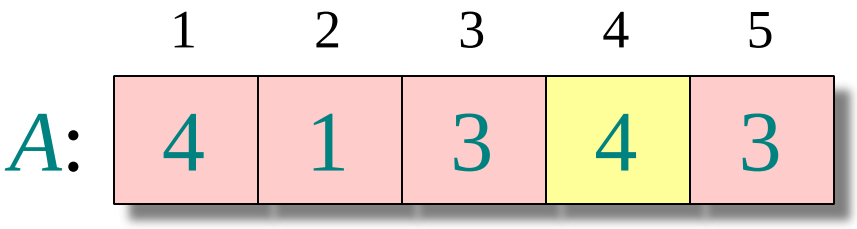


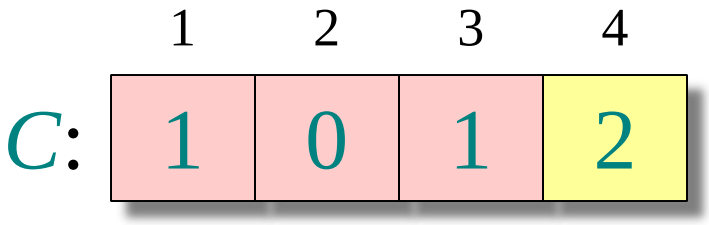


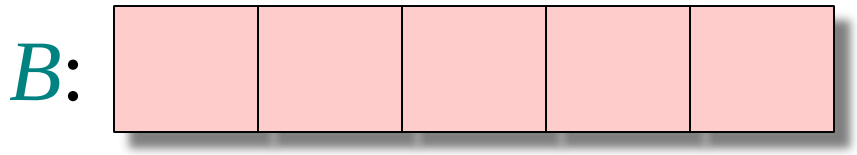
for $j \leftarrow 1$ **to** n

do $C[A[j]] \leftarrow C[A[j]] + 1$ ▷ $C[i] = |\{\text{key} = i\}|$

Ciclo 2



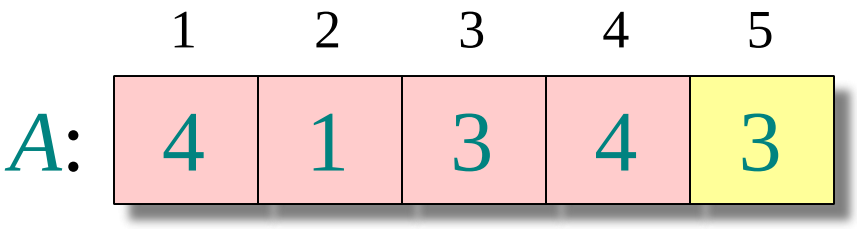


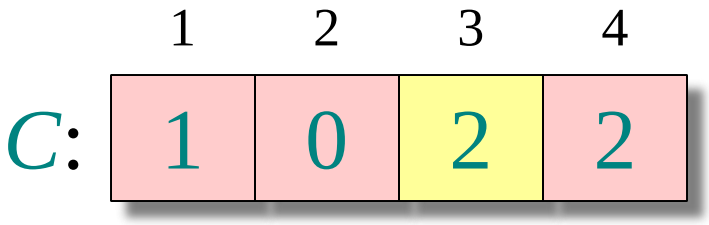


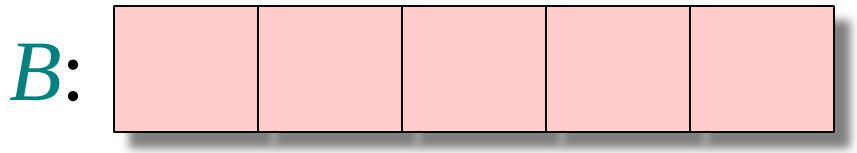
for $j \leftarrow 1$ **to** n

do $C[A[j]] \leftarrow C[A[j]] + 1$ ▷ $C[i] = |\{\text{key} = i\}|$

Ciclo 2



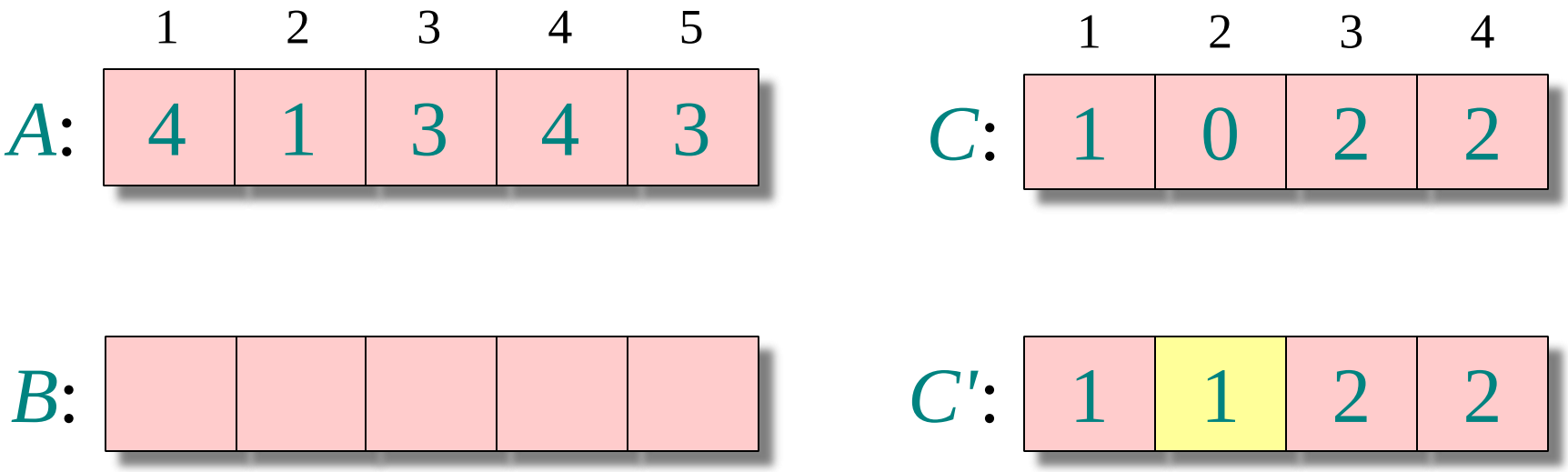




for $j \leftarrow 1$ **to** n

do $C[A[j]] \leftarrow C[A[j]] + 1$ ▷ $C[i] = |\{\text{key} = i\}|$

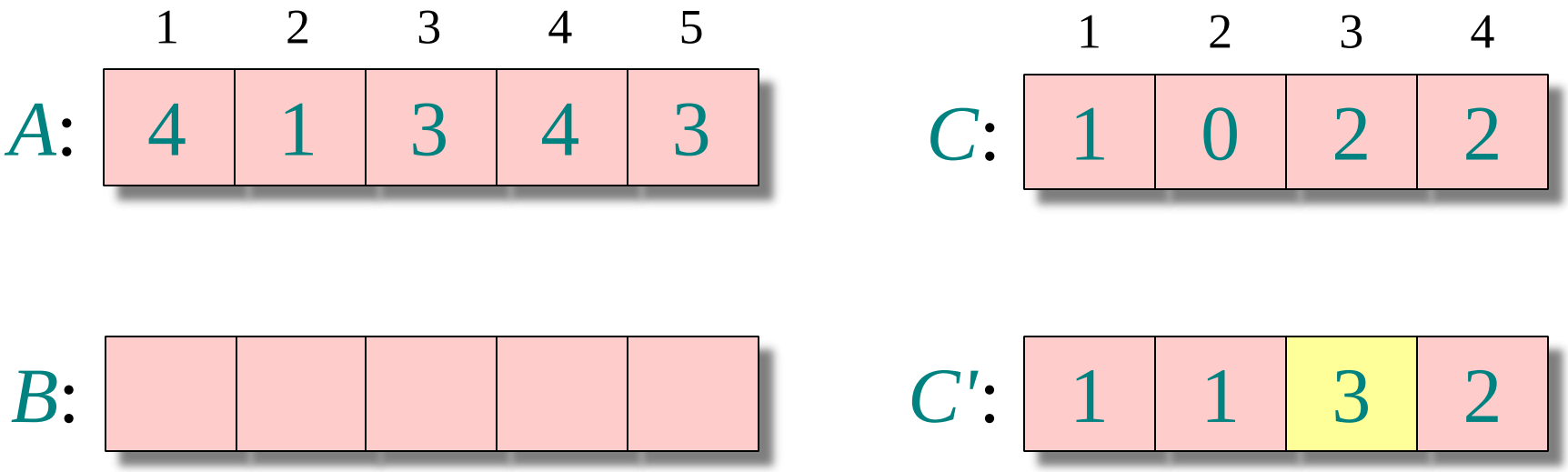
Ciclo 3



for $i \leftarrow 2$ **to** k

do $C[i] \leftarrow C[i] + C[i–1]$ ▷ $C[i] = |\{\text{key} \leq i\}|$

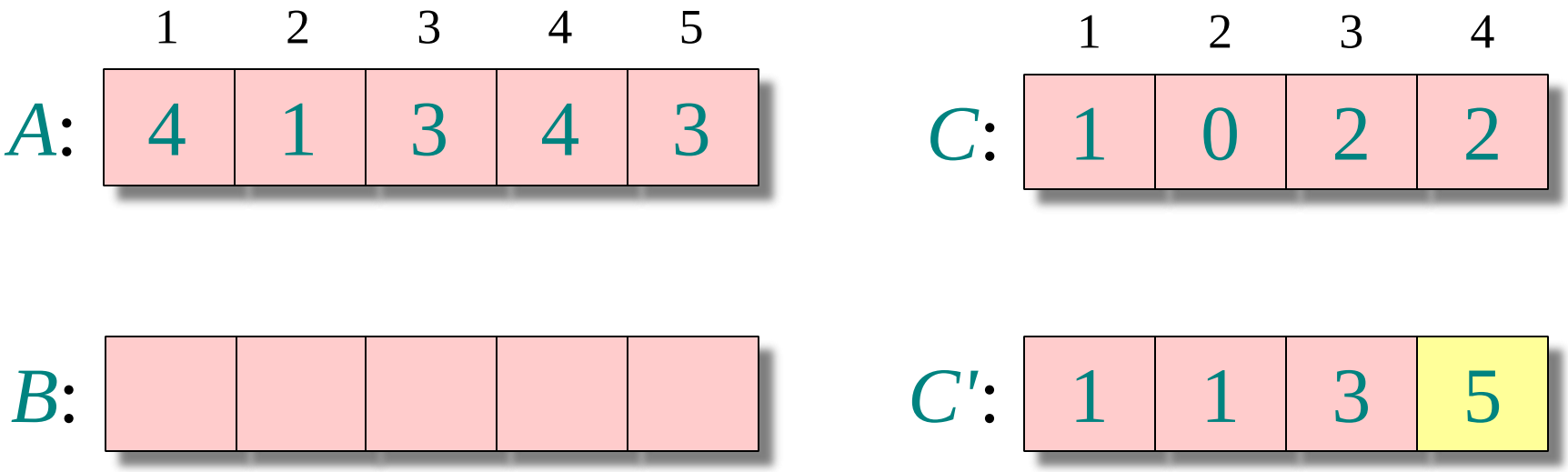
Ciclo 3



for $i \leftarrow 2$ **to** k

do $C[i] \leftarrow C[i] + C[i-1]$ ▷ $C[i] = |\{\text{key} \leq i\}|$

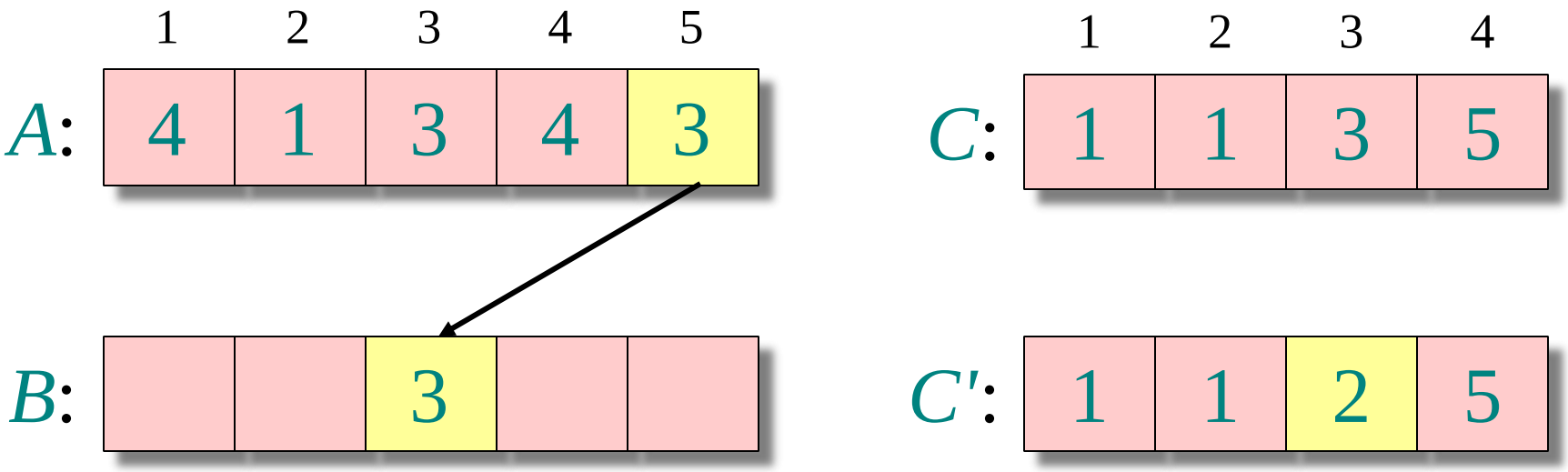
Ciclo 3



for $i \leftarrow 2$ **to** k

do $C[i] \leftarrow C[i] + C[i-1]$ ▷ $C[i] = |\{\text{key} \le i\}|$

Ciclo 4

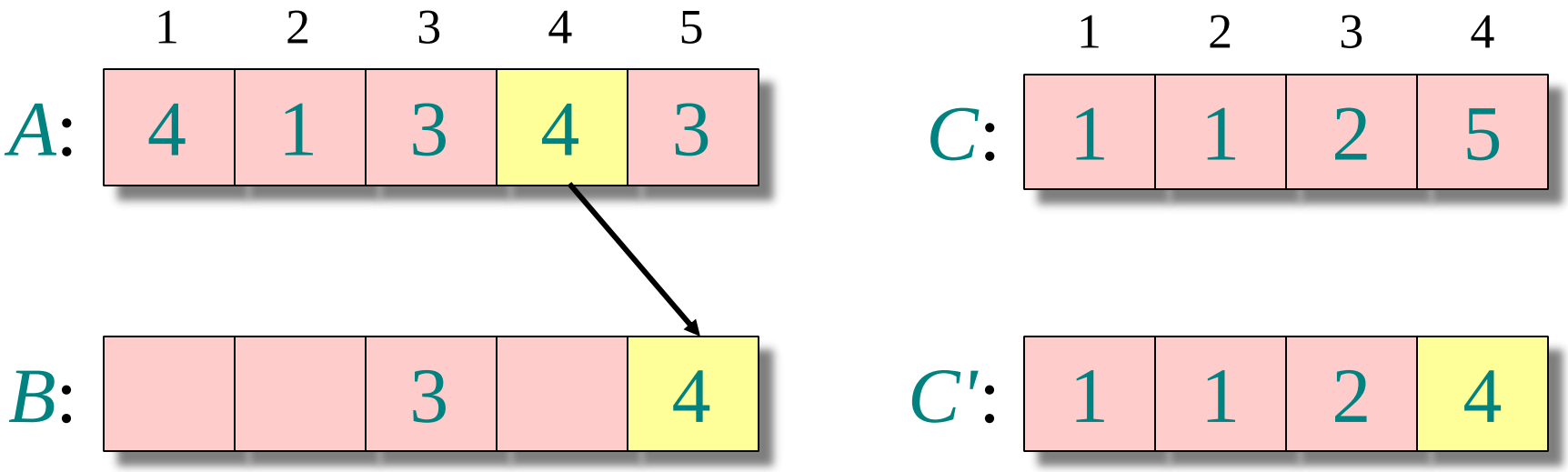


for $j \leftarrow n$ **downto** 1

do $B[C[A[j]]] \leftarrow A[j]$

$C[A[j]] \leftarrow C[A[j]] - 1$

Ciclo 4

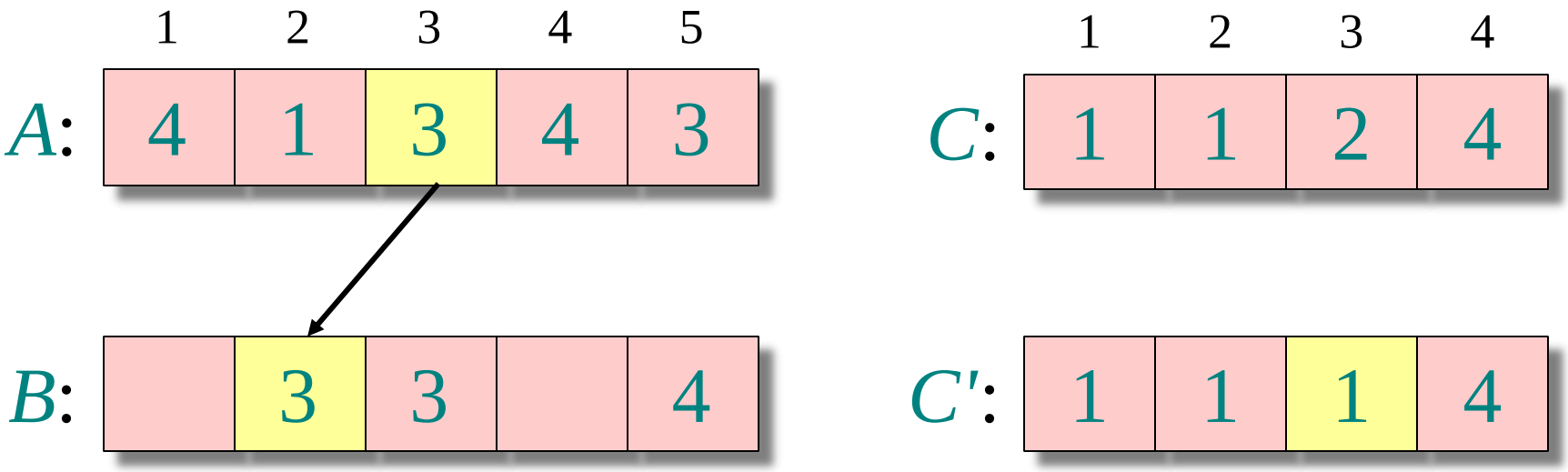


for $j \leftarrow n$ **downto** 1

do $B[C[A[j]]] \leftarrow A[j]$

$C[A[j]] \leftarrow C[A[j]] - 1$

Ciclo 4

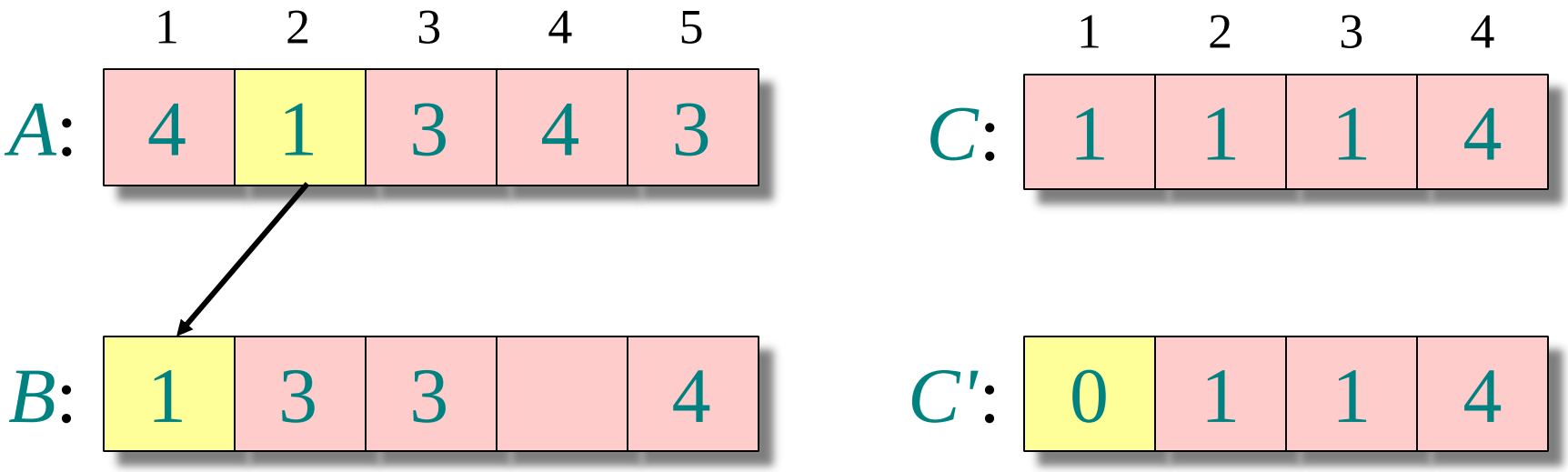


for $j \leftarrow n$ **downto** 1

do $B[C[A[j]]] \leftarrow A[j]$

$C[A[j]] \leftarrow C[A[j]] - 1$

Ciclo 4

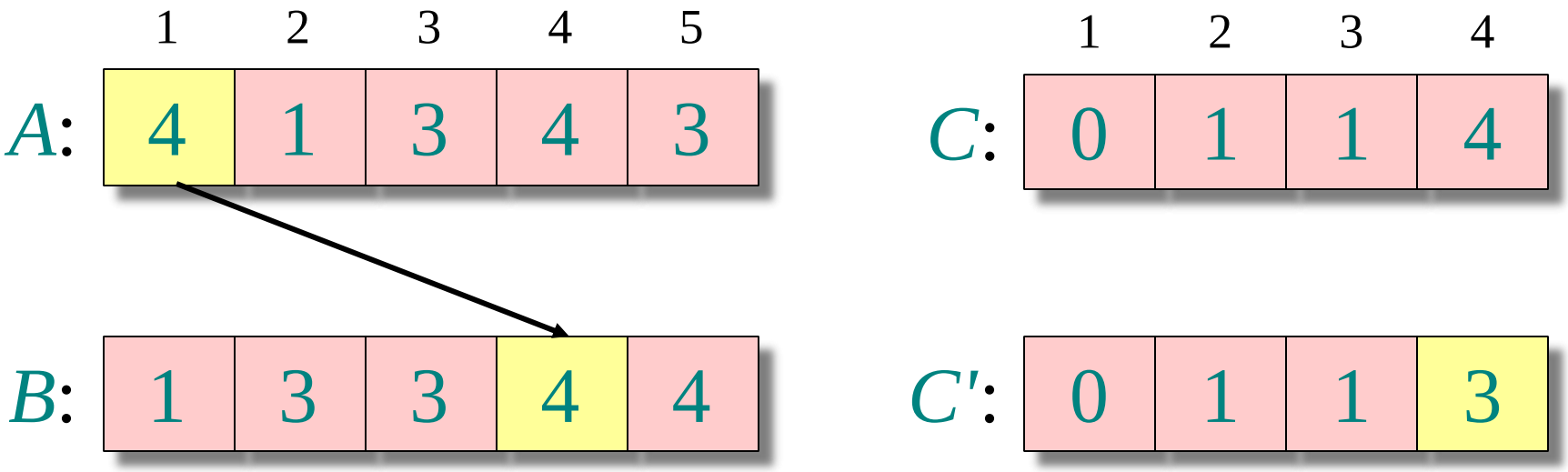


for $j \leftarrow n$ **downto** 1

do $B[C[A[j]]] \leftarrow A[j]$

$C[A[j]] \leftarrow C[A[j]] - 1$

Ciclo 4



for $j \leftarrow n$ **downto** 1

do $B[C[A[j]]] \leftarrow A[j]$

$C[A[j]] \leftarrow C[A[j]] - 1$

Análisis

$\Theta(k)$ **for** $i \leftarrow 1$ **to** k
do $C[i] \leftarrow 0$

$\Theta(n)$ **for** $j \leftarrow 1$ **to** n
do $C[A[j]] \leftarrow C[A[j]] + 1$

$\Theta(k)$ **for** $i \leftarrow 2$ **to** k
do $C[i] \leftarrow C[i] + C[i-1]$

$\Theta(n)$ **for** $j \leftarrow n$ **downto** 1
do $B[C[A[j]]] \leftarrow A[j]$
$C[A[j]] \leftarrow C[A[j]] - 1$

$\Theta(n + k)$

Tiempo de ejecución

Si $k = O(n)$, entonces $\Theta(n)$.

- Pero ordenar cuesta, $\Omega(n \lg n)$
- ¿Cuál es la falacia?

Tiempo de ejecución

Si $k = O(n)$, entonces $\Theta(n)$.

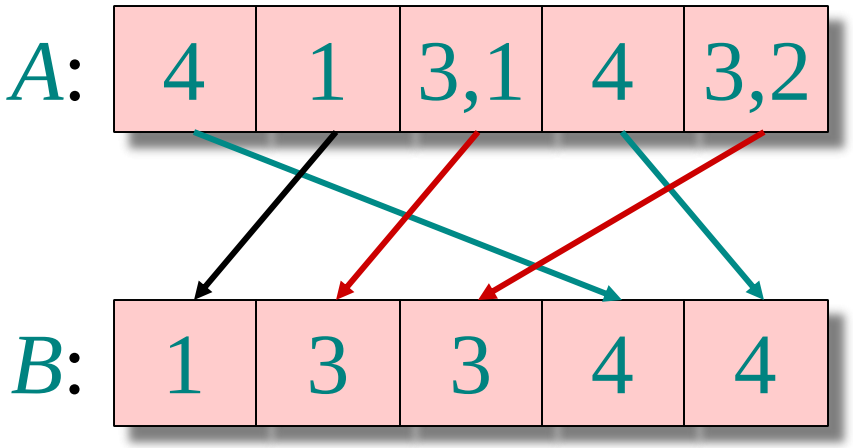
- Pero ordenar cuesta, $\Omega(n \lg n)$
- ¿Cuál es la falacia?

Respuesta:

- ***Ordenamiento por comparación*** ***toma*** $\Omega(n \lg n)$.
- Counting sort no usa ***comparaciones***

Ordenamiento estable

Counting sort es ***estable***: es decir mantiene el orden relativo entre los elementos.



Radix sort

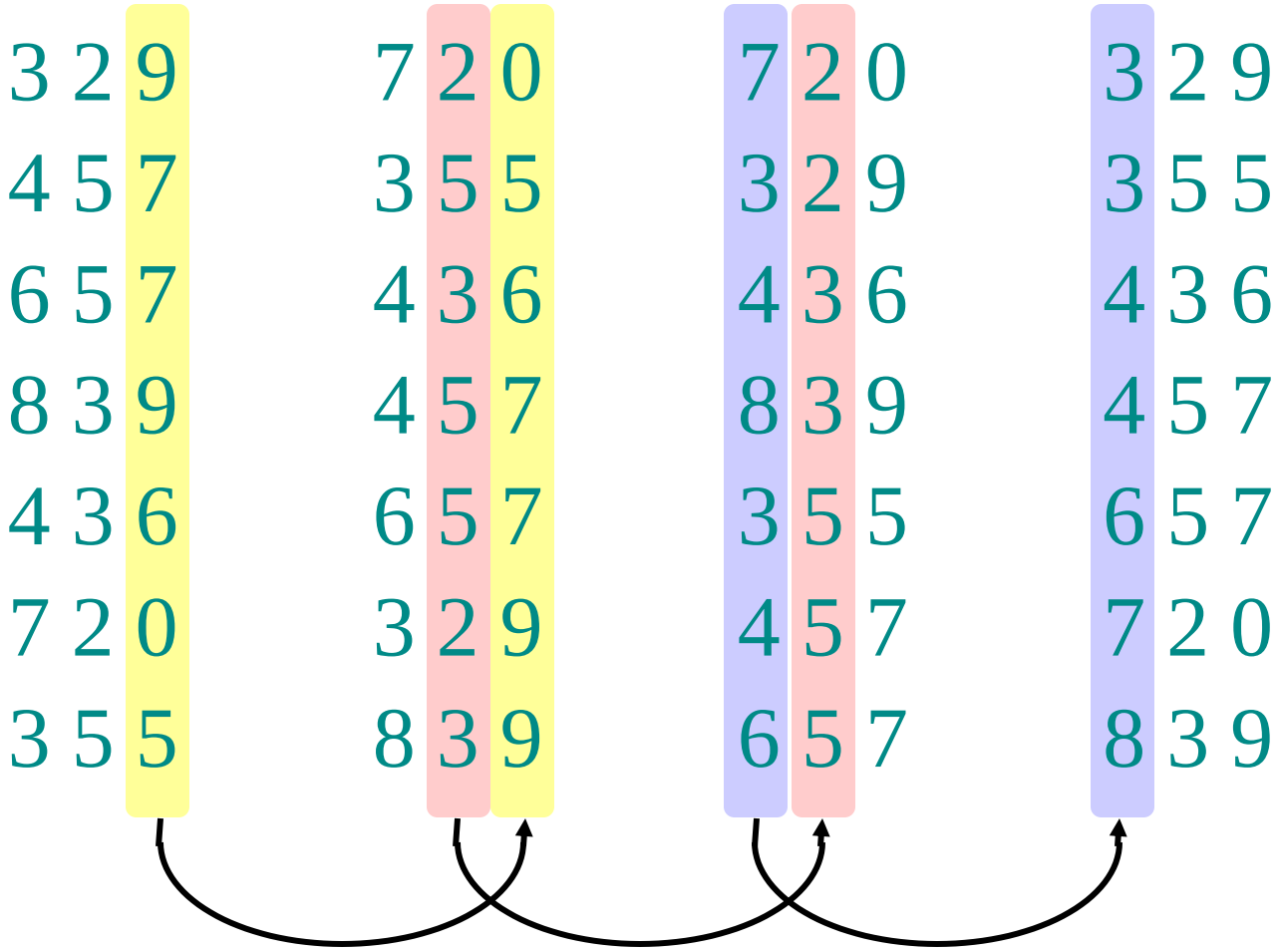
- ***Ordenamiento digito a digito.***
- La idea origina del Hollerith : era ordenar desde el digito más significativo
- Es mejor ordenar primero desde el ***-digito menos ssignificativo*** en conjutno con un algotimo de ordenamiento ***estable*** .

“Modern” IBM card

- Un caracter por columna.

Producido en WWW Virtual Punch-Card Server.

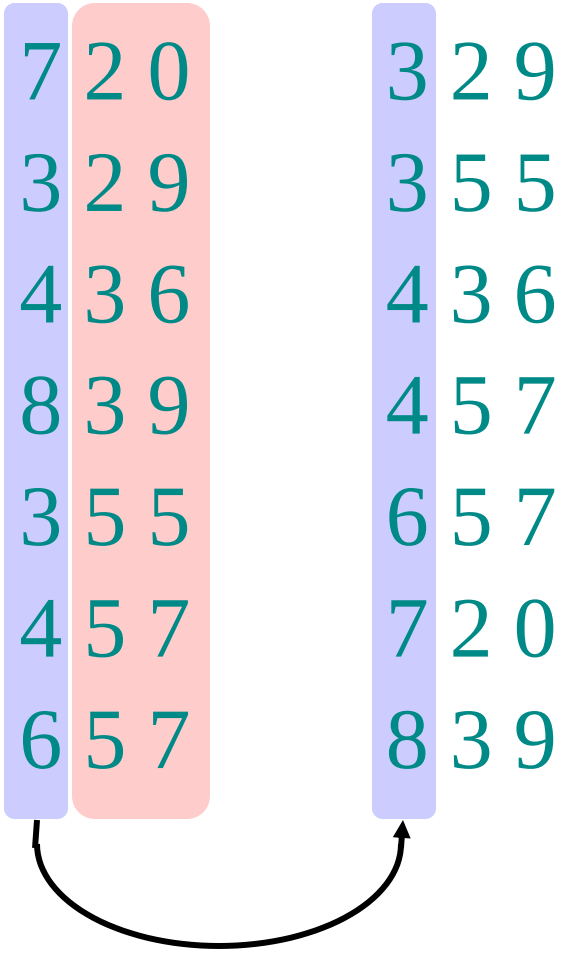
Funcionamiento de radix sort



Radix sort produce el resultado correcto.

Indución sobre la posición del digito

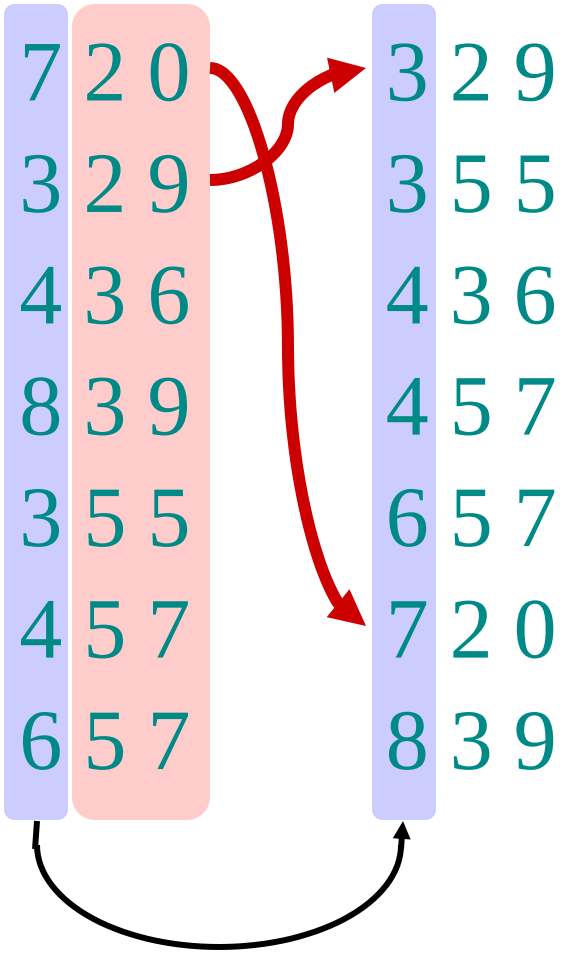
- Asumir que los números ya están ordenados con respecto del $t - 1$ digitos menos significativos.
- Ordenar con respecto del dígito en la posición t



Radix sort produce el resultado correcto (I).

Indución sobre la posición del digito

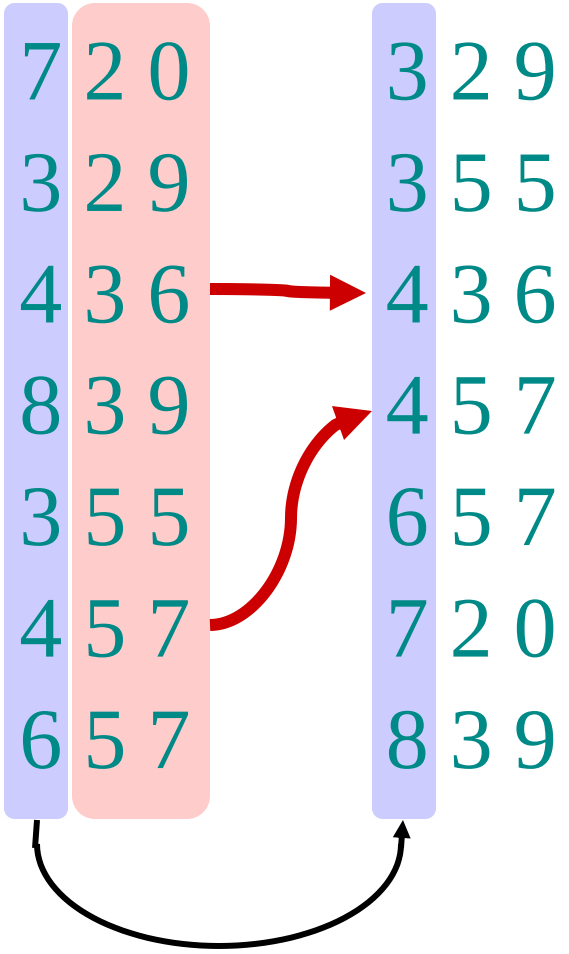
- Asumir que los números ya están ordenados con respecto del *t*-1 digitos menos significativos.
- Ordenar por el digito en la posición *t*
  - Los números que difieren en el digito *t* están correctamente ordenados.



Radix sort produce el resultado correcto (II).

Indución sobre la posición del digito

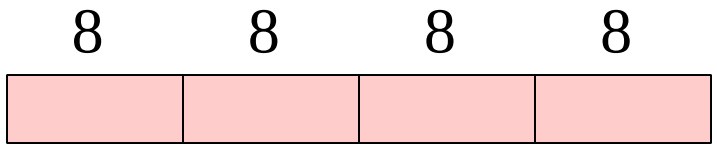
- Asumir que los números ya están ordenados con respecto del *t*-1 digitos menos significativos.
- Ordenar por el digito en la posición *t*
  - Los números que difieren en el digito *t* están correctamente ordenados.
  - Dos números iguales en el digito *t* se colocan en la posición correcta



Análisis de radix sort

- Asuma que ordena el arreglo utilizando como algoritmo auxiliar counting sort.
- Oredar n palabras de b bits cada una.
- Cada palabra puede ser vista como si tuviera b/r base-2^r digitos.

Ejemplo: 32-bit



$r = 8 \Rightarrow b/r = 4$ *pasadas con* counting sort *sobre los* base-2^8 *digitos*; *o* $r = 16 \Rightarrow b/r = 2$ *pasadas de* counting sort *sobre los base-*2^{16} *digitos*.

Análisis (II)

Recuerde: Counting sort tiene una complejidad de $\Theta(n + k)$ ordenando n numeros en un rango de 0 a $k - 1$.

Cada palababra de b-bits es partida en piezas de r-bits , cada pasada de counting sort toma $\Theta(n + 2^r)$. Como se requieren b/r pasadas, tenemos: .

Elegir r que minimice $T(n, b)$:

- Incrementar r implica menos pasadas, pero como $r > \lg n$, el tiempo crece exponencialmente.

Eligiendo r

Minimizar $T(n, b)$ diferenciando y estableciando a 0.

Observe que no se quiere que $2^r > n$, y no hay problema en elegir r tan grande como sea posible mientras se satisfaga la restricción.

Elegir $r = \lg n$ implica $T(n, b) = \Theta(bn/\lg n)$.

- Para numeros en el rango de 0 a $n^d - 1$, se tiene $b = d \lg n \Rightarrow$ radix sort corre en $\Theta(dn)$ tiempo.

Conclusiones

En la práctica, radix sort es rápido para entradas grandes.

Ejemplo (32-bit):

- A lo mas 3 pasadas ordenando ≥ 2000 numeros.
- Merge sort y quicksort realizan al menos $\lceil \lg 2000 \rceil = 11$ pasadas.

Desventajas:

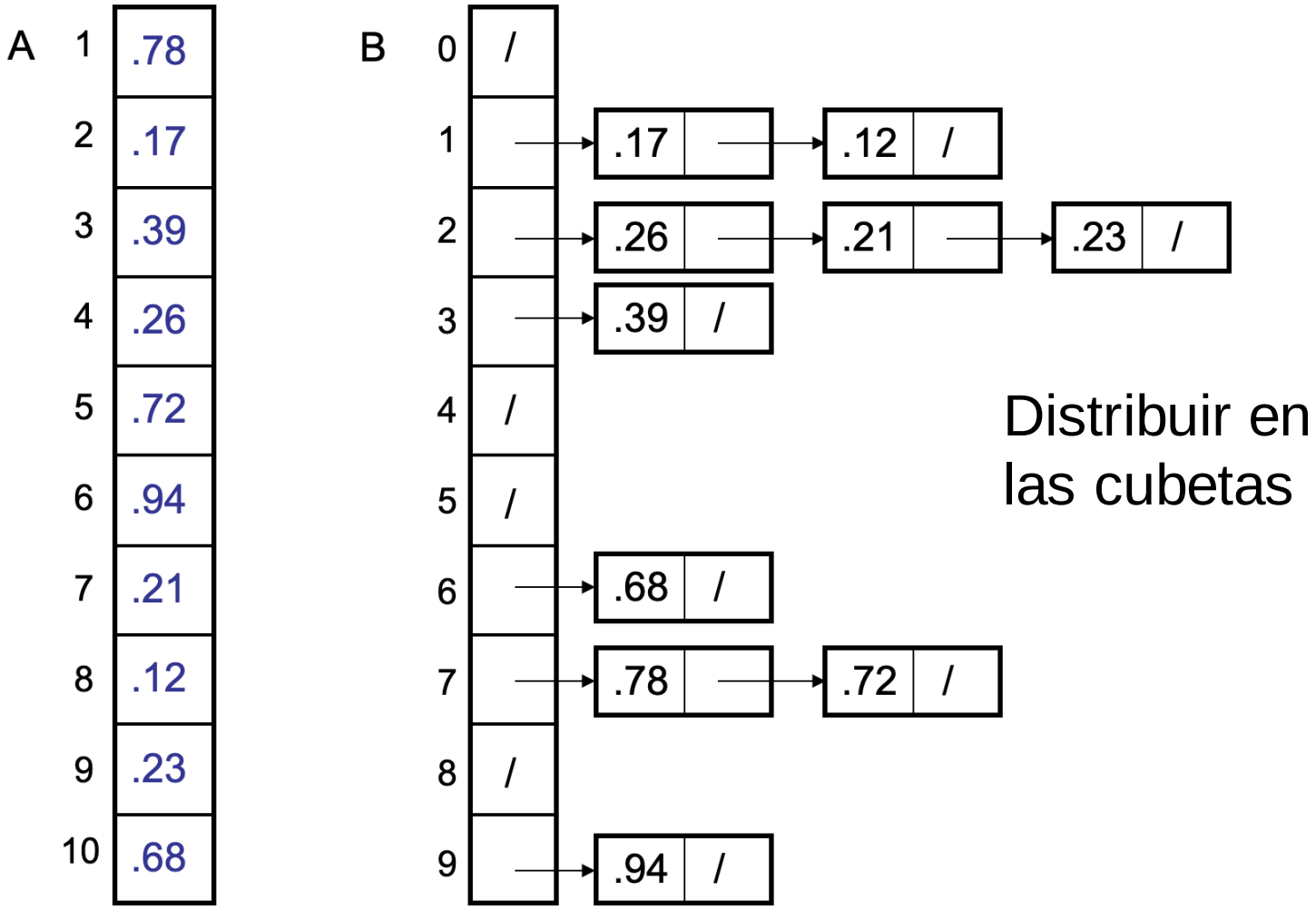
No se puede ordenar en sitio utilizando counting sort. Además, a diferencia de quicksort, radix sort muestra poca localidad de referencia y, por lo tanto, un quicksort bien ajustado podría obtener mejores resultados a veces en los procesadores modernos.

Bucket sort

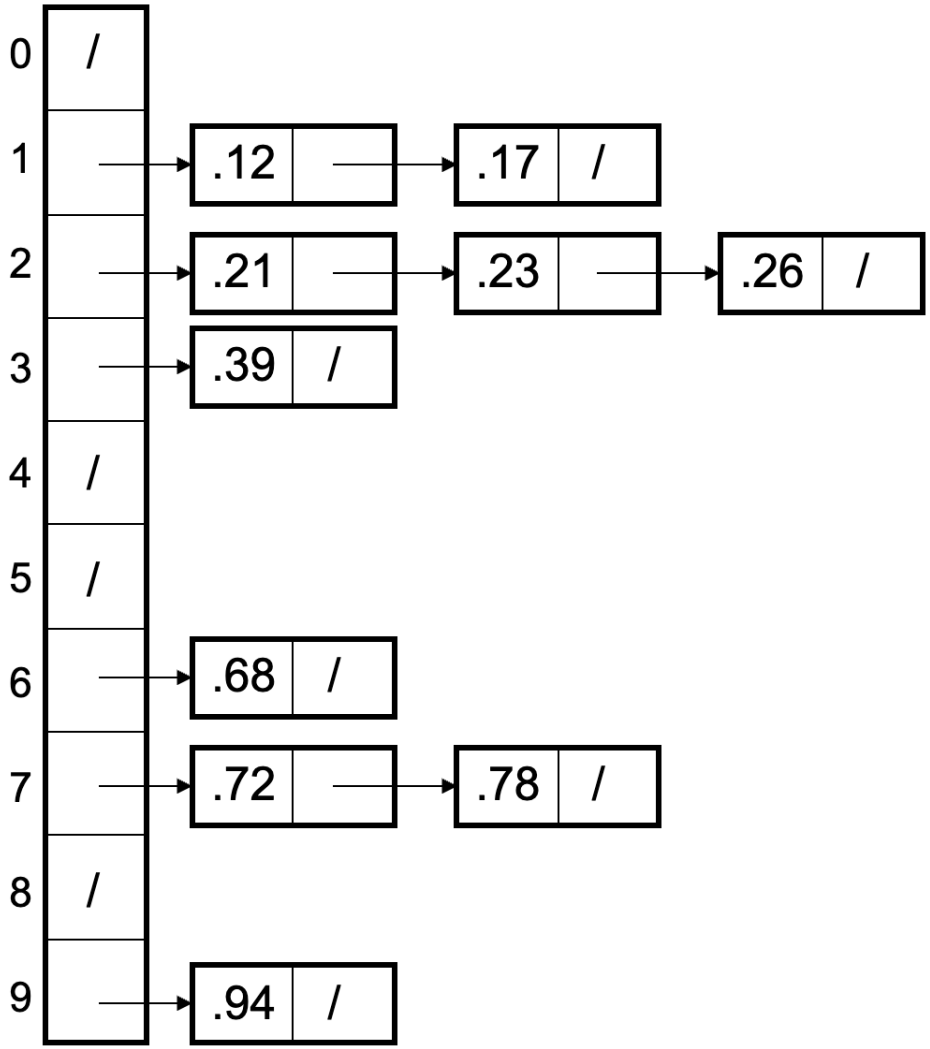
Se asume que la entrada se genera mediante un proceso aleatorio que distribuye los elementos uniformemente en [0, 1)

- Idea:
  - Dividir [0, 1) en k cubetas(buckets) del mismo tamaño ($k=\Theta(n)$)
  - Distribuir las n variables de entrada en cada una de las cubetas
  - Ordernar cada cubeta (e.g., usando quicksort)
  - Recorrer las cubetas en orden en cada uno de los elementos.
- **Input:** $A[1 \ldots n]$, donde $0 \leq A[i] < 1$ para todo i
- **Output:** elementos en $A[i]$ ordenados

Ejemplo de Bucket sort

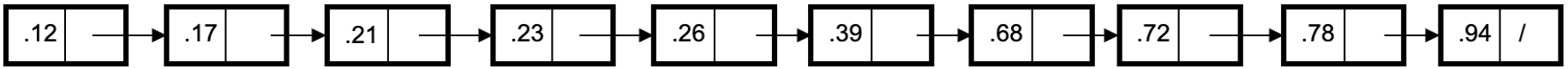


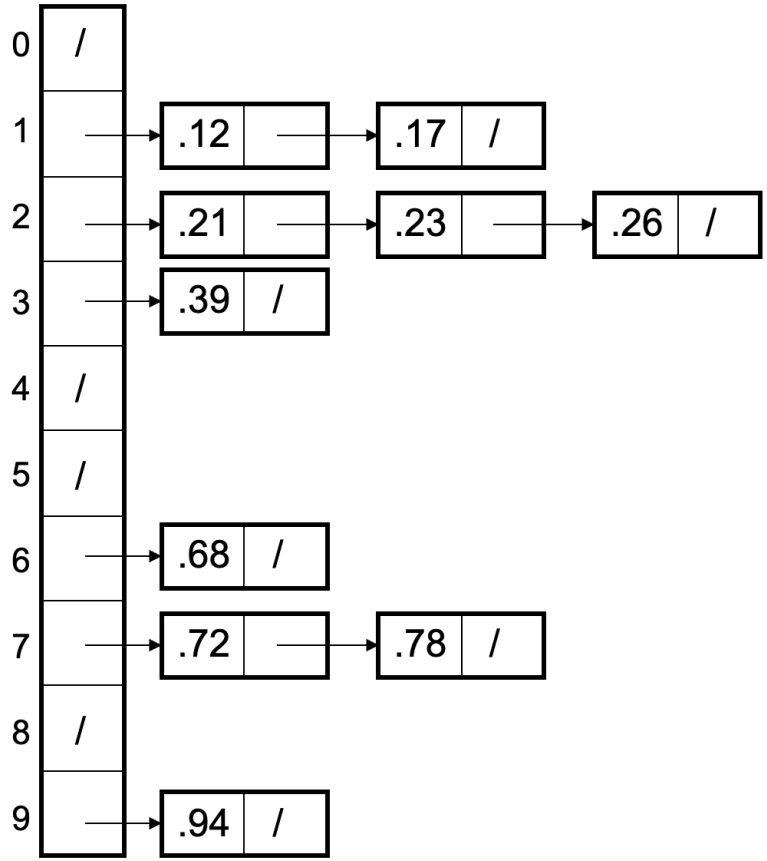
Ejemplo de Bucket sort



Ordenar cada cubetas

Ejemplo de Bucket sort





Concatenar las cubetas en orden

Análisis de Bucket sort

BUCKET-SORT(A, n)

for i ← 1 **to** n

do insertar A[i] en la lista B[⌊nA[i]⌋]

O(n)

for i ← 0 **to** k - 1

do ordenar B[i] usando quicksort

k O(n/k log(n/k))
=O(nlog(n/k)

concatenar B[0], B[1], . . . , B[n -1] en orden

return las listas concatenadas

O(k)

O(n) (if k=Θ(n))

Bucket sort

- Bucket sort asume que la entrada viene de una distribución uniforme.
- Las estimación de complejidad computacional involucran el número de cubetas.
- Bucket sort puede ser excepcionalmente rápida debido a la forma en que se asignan los elementos a cada una de las cubetas, normalmente mediante una matriz en la que el índice es el valor.
- Se requiere más memoria auxiliar para cada cubeta a costa del tiempo de ejecución.
- La complejidad de tiempo promedio para la clasificación de cubetas es O $(n + k)$. El peor caso $O(n \log n)$.
- La complejidad del espacio para la clasificación de cubetas es

  $O (n + k)$