Ayudantía 03

QuickSort y CountingSort



QuickSort

Pseudocódigo

Input: secuencia A y dos enteros i, f

Output: Nada

```
   QuickSort (A, i, f):
1      if i ≤ f :
2          p ← Partition(A, i, f)
3          Quicksort(A, i, p − 1)
4          Quicksort(A, p + 1, f)
```

El verdadero Héroe

- Elige el pivote
- Coloca al pivote en su posición correcta en el arreglo ordenado
- Pone los elementos menores que él a su izquierda y los mayores a su derecha

```
Partition (A, i, f):
1       x ← índice aleatorio en {i, ..., f}
2       p ← A[x]
3       A[x] ⇄ A[f]
4       j ← i
5       for k = i ... f − 1 :
6           if A[k] < p :
7               A[j] ⇄ A[k]
8               j ← j + 1
9       A[j] ⇄ A[f]
0       return j
```

Resumen de quicksort

- Elegir pivote
- Mover elementos a cada lado del pivote, a un lado los mayores y al otro los menores
- La lista va a quedar separada en dos sublistas, una con los elementos a la izq. del pivote y otra con los de la derecha
- Repetir recursivamente para cada sublista mientras estas contengan más de un elemento.
- Lista ordenada!

Complejidad

Mejor Caso	**O(n * log(n))**
Caso Promedio	**O(n * log(n))**
Peor Caso	**O(n^2)**

¿Cuándo ocurre cada uno de ellos?

Veamos la visualización !

Quicksort!



Ejercicio 1

Considere el siguiente pseudocódigo de una variación de Quicksort visto en clases con partición de Lomuto.

```
function quicksort(A, low, high):
    if low < high:
        p = partition(A, low, high)
        quicksort(A, low, p-1)
        quicksort(A, p+1, high)

function partition(A, low, high):
    pivot = A[high]
    i = low - 1
    for j = low to high-1:
        if A[j] <= pivot:
            i = i + 1
            swap(A[i], A[j])
    swap(A[i+1], A[high])
    return i+1
```

Parte A

Explique paso a paso qué realiza la función partition, indicando:

1. ¿Cómo se elige el pivote en este pseudocódigo?
2. ¿Qué rol cumple la variable i en el ciclo?
3. ¿Qué garantiza la posición final del pivote al terminar la función?

1. ¿Cómo se elige el pivote?

El pivote se elige como el último elemento del subarreglo, es decir:

$$pivote = A[high]$$

Esta es una estrategia determinística y fija que siempre selecciona el elemento en la posición más alta del rango actual [low,high].

2. ¿Qué rol cumple la variable i?

La variable i cumple el rol de frontera o separador del pivote durante todo el ciclo:

- Es el índice del último elemento que es menor o igual al pivote.
- Mantiene la invariante de partición: todos los elementos en A[low...i] son ≤ pivote.
- Actúa como frontera móvil que separa elementos procesados menores del pivote de elementos mayores.

.

3. ¿Qué garantiza la posición final del pivote?

Al terminar la función, el pivote queda en la posición i+1 y se garantiza que:

- $\forall k \in [low,i]$: A[k] ≤ pivote → (Elementos a la izquierda)
- $\forall k \in [i+2,high]$: A[k] > pivote → (Elementos a la derecha)
- A[i+1] = pivote → (Posición del pivote)

  Garantía fundamental: El pivote está en su posición final correcta en el arreglo completamente ordenado, y no necesitará moverse en futuras operaciones.

Parte B

1. ¿Qué ocurriría con la complejidad de Quicksort si siempre se elige el primer elemento como pivote y el arreglo está ya ordenado de manera creciente?
2. ¿Qué ventaja tiene elegir el pivote como la mediana?

1. ¿Que pasa si siempre se elige el primer elemento como pivote y el arreglo ya esta ordenado en orden creciente?

- El pivote será siempre el elemento más pequeño.
- Una partición tendrá 0 elementos, la otra tendrá n−1 elementos.
- Esto resulta en particiones completamente desbalanceadas.

$$T(n) = T(0) + T(n-1) + O(n) \tag{1}$$

$$T(n) = T(n-1) + O(n) \quad (\text{ya que } T(0) = O(1)) \tag{2}$$

$$T(n) = O(n) + O(n-1) + O(n-2) + \cdots + O(1) \tag{3}$$

$$T(n) = O\left(\sum_{k=1}^{n} k\right) = O\left(\frac{n(n+1)}{2}\right) \tag{4}$$

$$T(n) = \boxed{O(n^2)} \tag{5}$$

2. ¿Qué ventaja tiene elegir el pivote como la mediana?

Elegir la mediana (o una aproximación) como pivote proporciona ventajas significativas:

1. La mediana divide el arreglo exactamente por la mitad.
2. Cada partición tiene aproximadamente n/2 elementos.
3. Asegura O(nlogn) incluso en el peor caso.
4. Rendimiento predecible independientemente del orden inicial.

Ejercicio 2

a) Supongamos que al ejecutar Quicksort sobre un arreglo particular, la subrutina partition hace siempre
el mayor número posible de intercambios; ¿cuánto tiempo toma Quicksort en este caso? ¿Qué fracción
del mayor número posible de intercambios se harían en el mejor caso? Justifica

- Intercambios de líneas 3 y 9 se hacen siempre
- Dentro del loop, sólo se hacen cuando A[k] < p
- Si queremos hacer la mayor cantidad de intercambios posibles, queremos que siempre A[k]< p.
- Entonces si tenemos un subarreglo de tamaño m, se hacen 2 + m-1 = m+1 intercambios
- Quedan subarreglos de largos 0 y m-1

```
Partition (A, i, f):
1     x ← índice aleatorio en {i, ..., f}
2     p ← A[x]
3     A[x] ⇄ A[f]
4     j ← i
5     for k = i ... f − 1 :
6         if A[k] < p :
7             A[j] ⇄ A[k]
8             j ← j + 1
9     A[j] ⇄ A[f]
0     return j
```

Con un arreglo de tamaño n tenemos:

$$T(n) = n + (n-1) + (n-2) + \ldots + 1$$

$$= \frac{n \times (n+1)}{2} = \frac{1}{2}\left(n^2 + n\right)$$

$$\in O\left(n^2\right)$$

b) El algoritmo quicker-sort llama a la subrutina pq-partition, que utiliza dos pivotes $\boldsymbol{p}$ y $\boldsymbol{q}$ $(\boldsymbol{p} < \boldsymbol{q})$ para particionar el arreglo en 5 partes: los elementos menores que $\boldsymbol{p}$, el pivote $\boldsymbol{p}$, los elementos entre $\boldsymbol{p}$ y $\boldsymbol{q}$, el pivote $\boldsymbol{q}$, y los elementos mayores que $\boldsymbol{q}$. Escribe el pseudocódigo de pq-partition. ¿Es quicker-sort más eficiente que quick-sort? Justifica.

Ya que la definición dice que $p < q$, entonces es posible asumir que no hay datos repetidos. De todos modos, esto no afecta el análisis.

Lo más simple es implementar partition con listas ligadas como vimos en clases:

```
1:  procedure PQ-PARTITION(lista ligada L)
2:      p, q ← dos nodos de L tal que p < q. Quitar estos nodos de L.
3:      A, B, C ← listas vacias    ▷ Los elementos menores a p, entre p y q y mayores a q respectivamente
4:      for nodo x ∈ L do
5:          if x < p then
6:              agregar x al final de A
7:          else if x < q then
8:              agregar x al final de B
9:          else
10:             agregar x al final de C
11:         end if
12:     end for
13:     return A, p, B, q, C
14: end procedure
```

El paso en la linea 2 es fácil de hacer en $\mathcal{O}(1)$, basta con extraer los primeros dos elementos de L, e intercambiarlos si $p > q$.

Recordemos que el peor caso de Quicksort se produce cuando partition separa una secuencia de m datos en dos secuencias disparejas, de 0 y $m-1$ datos cada una. En ese caso la complejidad para una secuencia inicial de n datos es de:

$$T(n) = n + (n-1) + (n-2) + \cdots + 1$$

$$T(n) \in \mathcal{O}(n^2)$$

En este caso esto también puede suceder, solo que se separa en 3 secuencias disparejas, de 0, 0 y $m-2$ elementos cada una. En este caso la complejidad para una secuencia inicial de n datos es de:

$$T(n) = n + (n-2) + (n-4) + \cdots + 1$$

$$T(n) \in \mathcal{O}(n^2)$$

Ambos algoritmos son iguales en el peor caso, así que no podemos afirmar que quickersort sea mejor que quicksort.

Ordenación Lineal

CountingSort()

input : Arreglo $A[0 \ldots n-1]$, natural k
output: Arreglo $B[0 \ldots n-1]$

```
CountingSort (A, k):
1    B[0...n-1] ← arreglo vacío de n celdas
2    C[0...k] ← arreglo vacío de k+1 celdas
3    for i = 0...k :
4        C[i] ← 0
5    for j = 0...n-1 :
6        C[A[j]] ← C[A[j]] + 1
7    for p = 1...k :
8        C[p] ← C[p] + C[p-1]
9    for r = n-1...0 :
10       B[C[A[r]] - 1] ← A[r]
11       C[A[r]] ← C[A[r]] - 1
12   return B
```

- Este algoritmo guarda en un arreglo auxiliar, el número de elementos menores o iguales a cada elemento del arreglo inicial. Luego utiliza esta información para colocar cada elemento en la posición correcta en el arreglo ordenado.
- Complejidad O(n+k), si k < n, entonces O(n)
- Memoria adicional O(n+k)

CountingSort()

input : Arreglo $A[0 \ldots n-1]$, natural k
output: Arreglo $B[0 \ldots n-1]$

```
CountingSort (A, k):
1    B[0...n-1] ← arreglo vacío de n celdas
2    C[0...k] ← arreglo vacío de k+1 celdas
3    for i = 0...k :
4        C[i] ← 0
5    for j = 0...n-1 :
6        C[A[j]] ← C[A[j]] + 1
7    for p = 1...k :
8        C[p] ← C[p] + C[p-1]
9    for r = n-1...0 :
10       B[C[A[r]] - 1] ← A[r]
11       C[A[r]] ← C[A[r]] - 1
12   return B
```

A =

2	5	3	0	2	3	0	3
0	1	2	3	4	5	6	7

k = 5

C =

0	0	0	0	0	0
0	1	2	3	4	5

C =

2	0	2	3	0	1
0	1	2	3	4	5

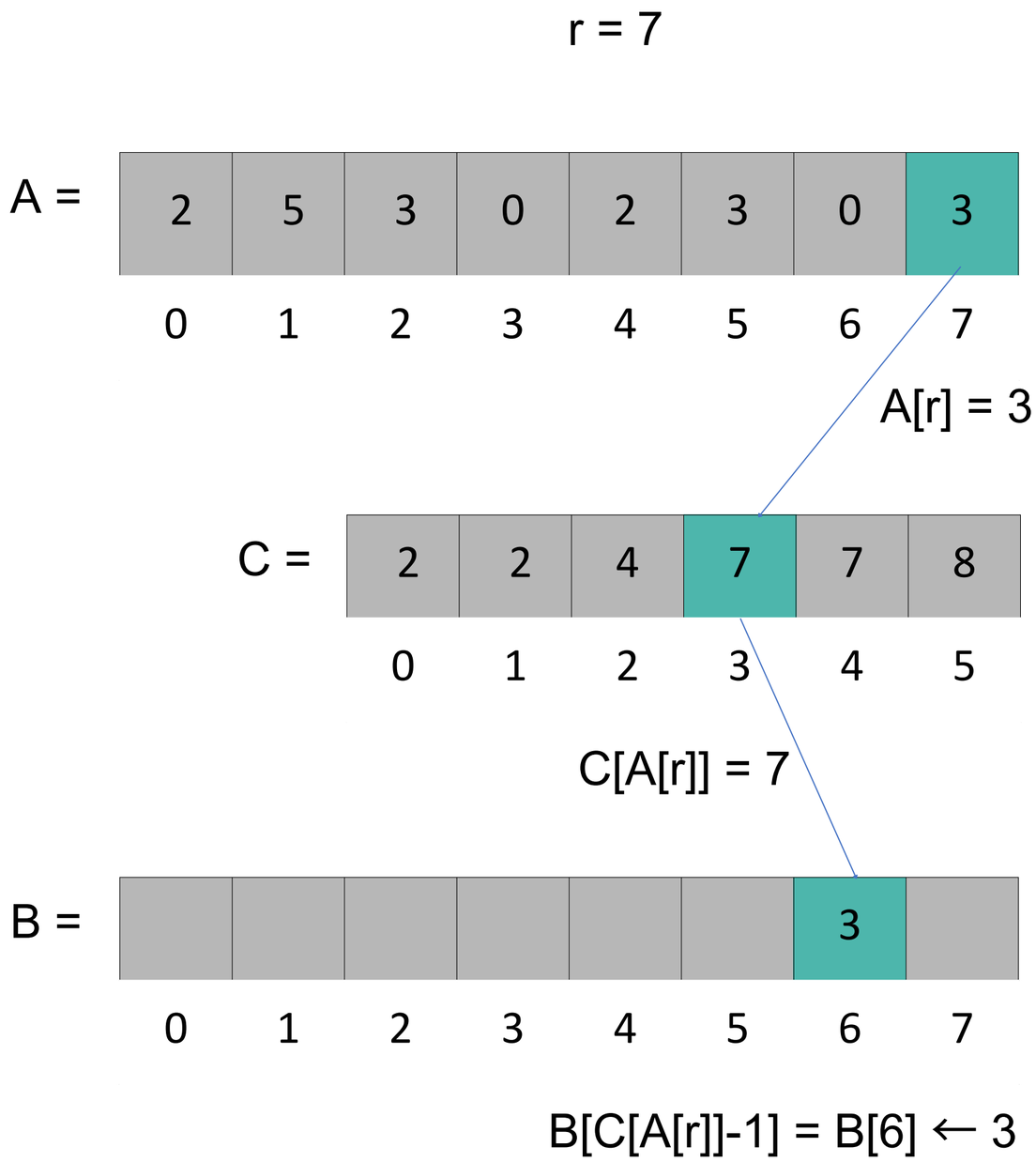
C =

2	2	4	7	7	8
0	1	2	3	4	5

CountingSort()

input : Arreglo $A[0 \ldots n-1]$, natural k
output: Arreglo $B[0 \ldots n-1]$

```
CountingSort (A, k):
1    B[0...n-1] ← arreglo vacío de n celdas
2    C[0...k] ← arreglo vacío de k+1 celdas
3    for i = 0...k :
4        C[i] ← 0
5    for j = 0...n-1 :
6        C[A[j]] ← C[A[j]] + 1
7    for p = 1...k :
8        C[p] ← C[p] + C[p-1]
9    for r = n-1...0 :
10       B[C[A[r]] - 1] ← A[r]
11       C[A[r]] ← C[A[r]] - 1
12   return B
```



CountingSort()

input : Arreglo $A[0 \ldots n-1]$, natural k
output: Arreglo $B[0 \ldots n-1]$

```
CountingSort (A, k):
1    B[0...n-1] ← arreglo vacío de n celdas
2    C[0...k] ← arreglo vacío de k+1 celdas
3    for i = 0...k :
4        C[i] ← 0
5    for j = 0...n-1 :
6        C[A[j]] ← C[A[j]] + 1
7    for p = 1...k :
8        C[p] ← C[p] + C[p-1]
9    for r = n-1...0 :
10       B[C[A[r]] - 1] ← A[r]
11       C[A[r]] ← C[A[r]] - 1
12   return B
```

r = 7

A =

2	5	3	0	2	3	0	3
0	1	2	3	4	5	6	7

A[r] = 3

C =

2	2	4	6	7	8
0	1	2	3	4	5

C[A[r]] ← C[A[r]] - 1 = 6

B =

						3	
0	1	2	3	4	5	6	7

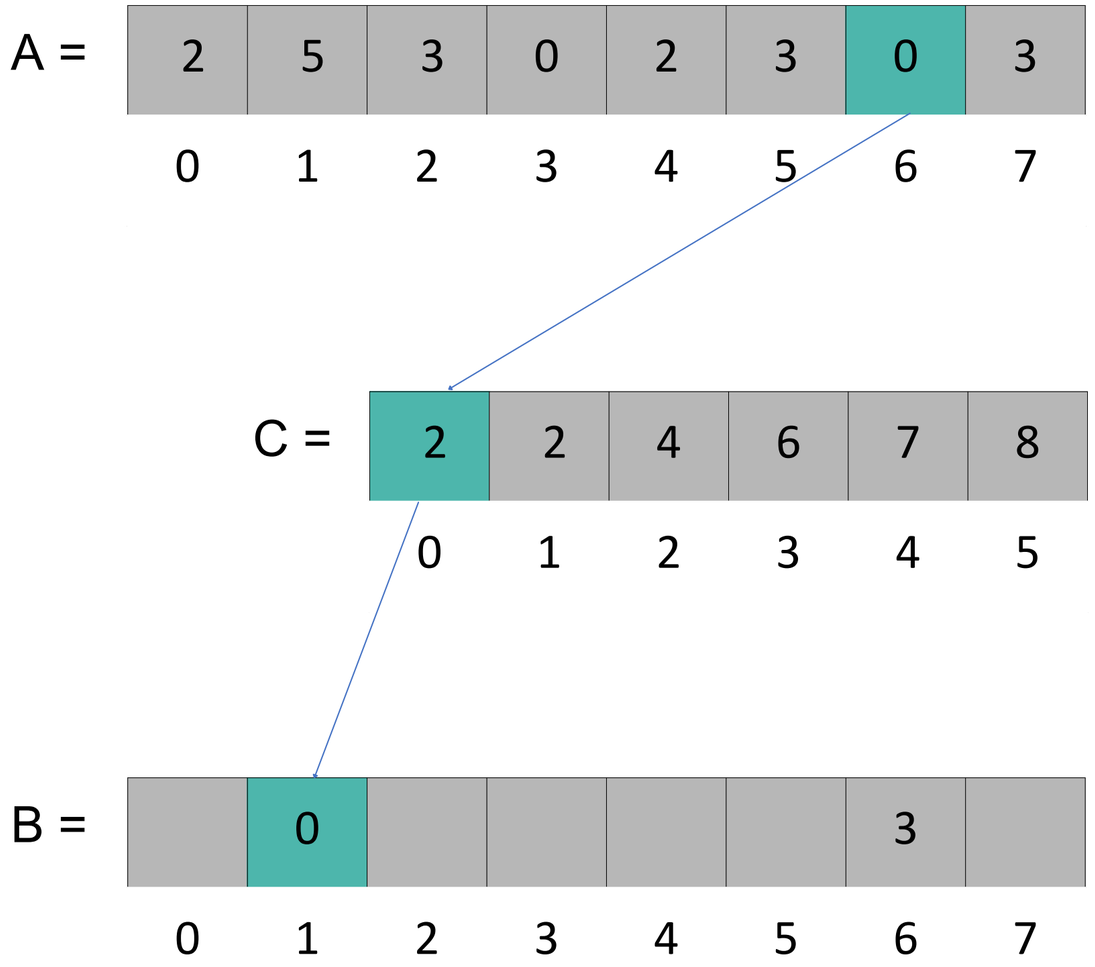
CountingSort()

input : Arreglo $A[0 \ldots n-1]$, natural k
output: Arreglo $B[0 \ldots n-1]$

`CountingSort` (A, k):

```
1   B[0...n-1] ← arreglo vacío de n celdas
2   C[0...k] ← arreglo vacío de k+1 celdas
3   for i = 0...k :
4       C[i] ← 0
5   for j = 0...n-1 :
6       C[A[j]] ← C[A[j]] + 1
7   for p = 1...k :
8       C[p] ← C[p] + C[p-1]
9   for r = n-1...0 :
10      B[C[A[r]] - 1] ← A[r]
11      C[A[r]] ← C[A[r]] - 1
12  return B
```

r = 6



A[r] = 0 C[A[r]] = 2 B[C[A[r]]-1] = B[1] ← 0

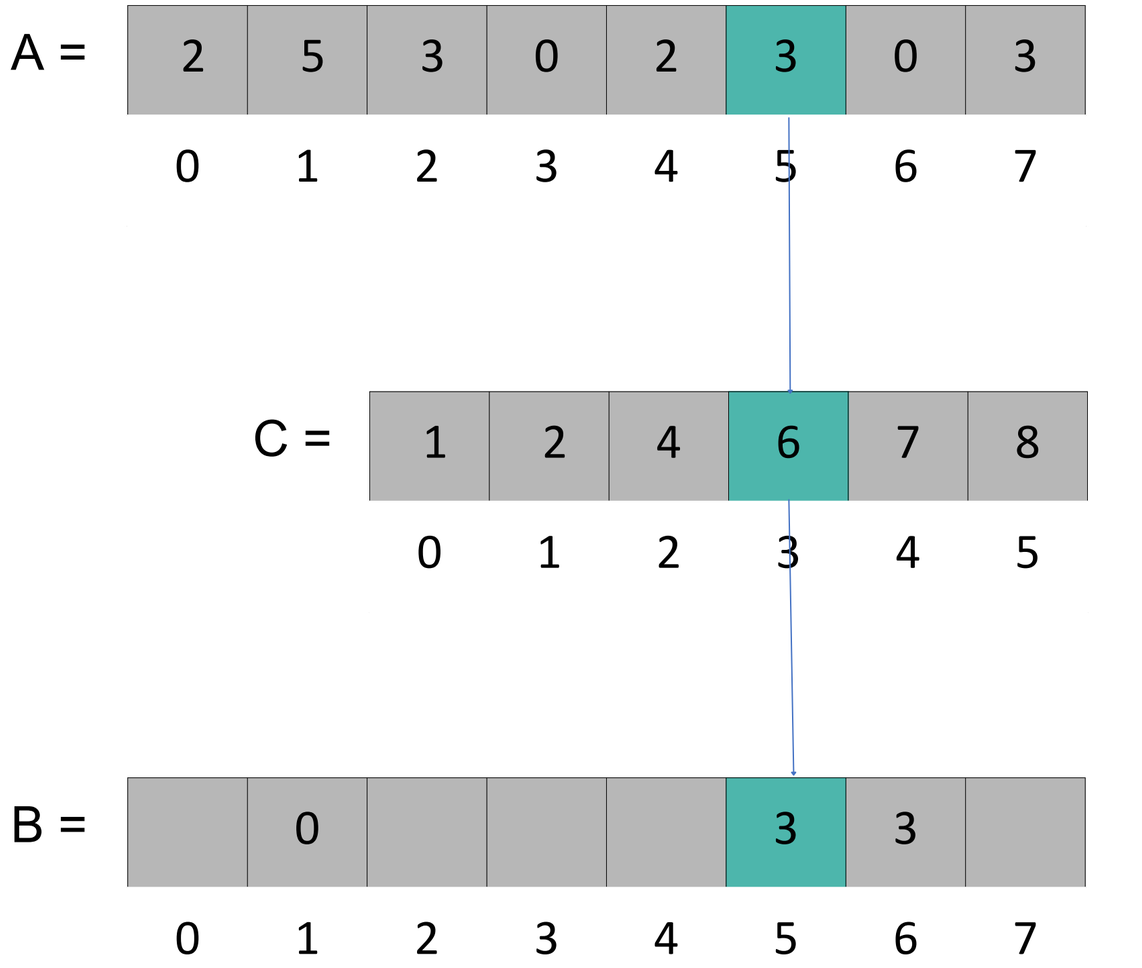
CountingSort()

input : Arreglo $A[0 \ldots n-1]$, natural k
output: Arreglo $B[0 \ldots n-1]$

`CountingSort` (A, k):

```
1   B[0...n-1] ← arreglo vacío de n celdas
2   C[0...k] ← arreglo vacío de k+1 celdas
3   for i = 0...k :
4       C[i] ← 0
5   for j = 0...n-1 :
6       C[A[j]] ← C[A[j]] + 1
7   for p = 1...k :
8       C[p] ← C[p] + C[p-1]
9   for r = n-1...0 :
10      B[C[A[r]] - 1] ← A[r]
11      C[A[r]] ← C[A[r]] - 1
12  return B
```

r = 5



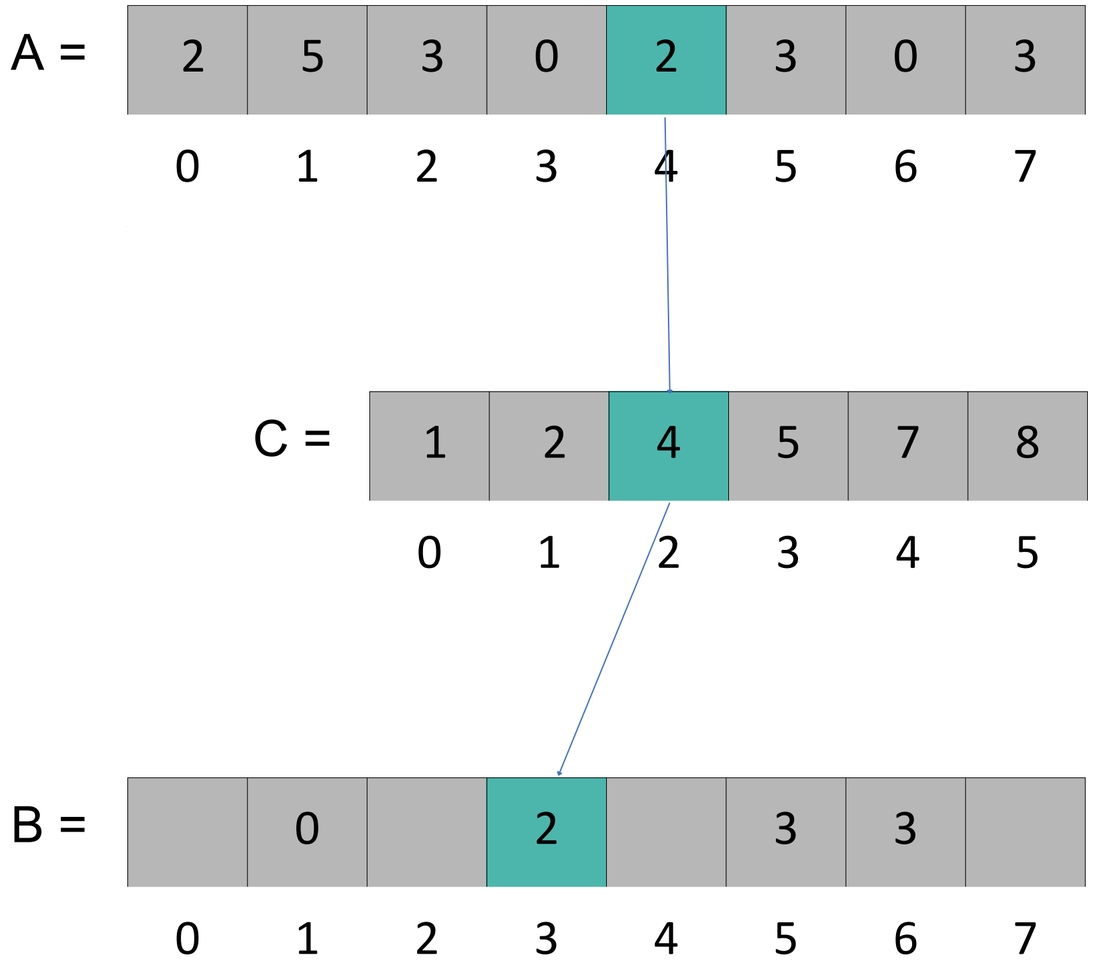
A[r] = 3 C[A[r]] = 6 B[C[A[r]]-1] = B[5] ← 3

CountingSort()

input : Arreglo $A[0 \ldots n-1]$, natural k
output: Arreglo $B[0 \ldots n-1]$

```
CountingSort (A, k):
1    B[0...n-1] ← arreglo vacío de n celdas
2    C[0...k] ← arreglo vacío de k+1 celdas
3    for i = 0...k :
4        C[i] ← 0
5    for j = 0...n-1 :
6        C[A[j]] ← C[A[j]] + 1
7    for p = 1...k :
8        C[p] ← C[p] + C[p-1]
9    for r = n-1...0 :
10       B[C[A[r]] - 1] ← A[r]
11       C[A[r]] ← C[A[r]] - 1
12   return B
```

r = 4



A[r] = 2 C[A[r]] = 4 B[C[A[r]]-1] = B[3] ← 2

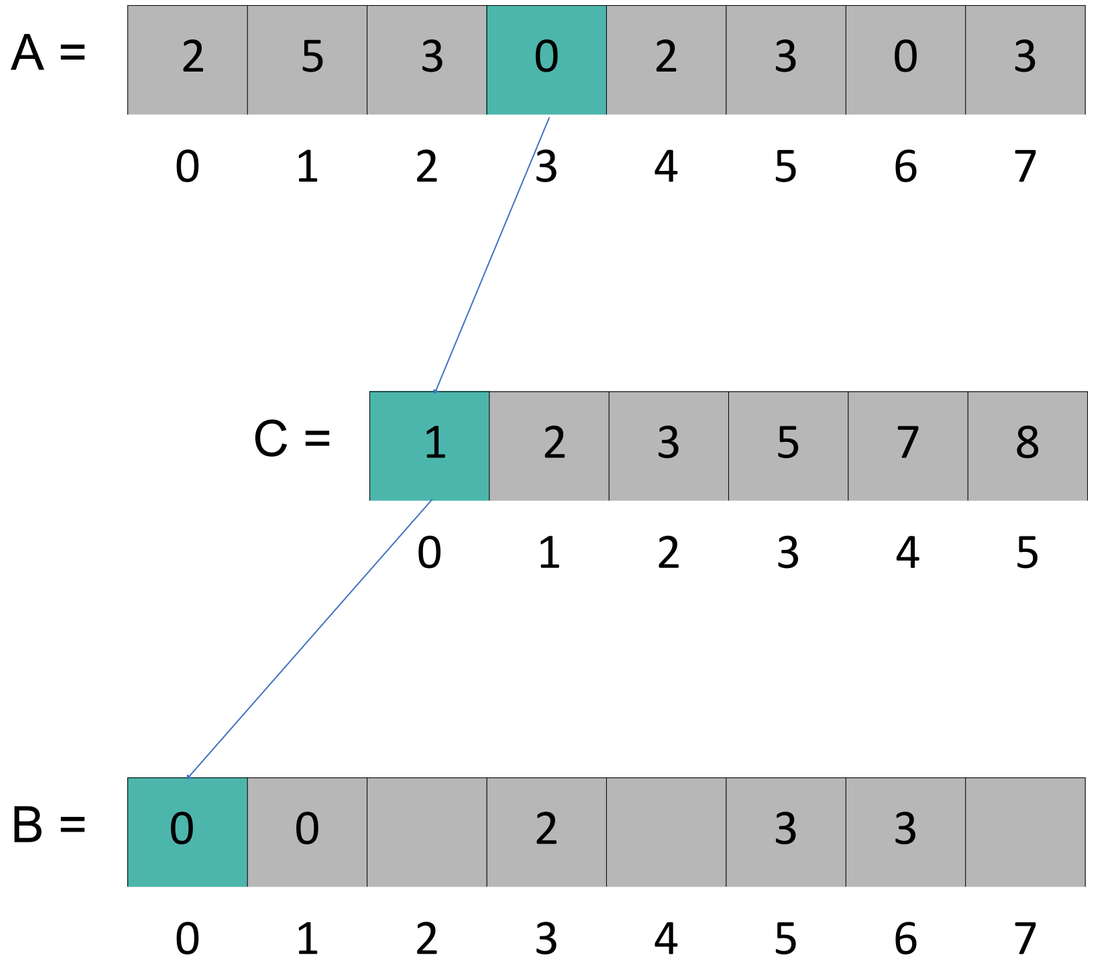
CountingSort()

input : Arreglo $A[0 \ldots n-1]$, natural k
output: Arreglo $B[0 \ldots n-1]$

`CountingSort` (A, k):

```
1   B[0...n-1] ← arreglo vacío de n celdas
2   C[0...k] ← arreglo vacío de k+1 celdas
3   for i = 0...k :
4       C[i] ← 0
5   for j = 0...n-1 :
6       C[A[j]] ← C[A[j]] + 1
7   for p = 1...k :
8       C[p] ← C[p] + C[p-1]
9   for r = n-1...0 :
10      B[C[A[r]] - 1] ← A[r]
11      C[A[r]] ← C[A[r]] - 1
12  return B
```

r = 3



A[r] = 0 C[A[r]] = 1 B[C[A[r]]-1] = B[0] ← 0

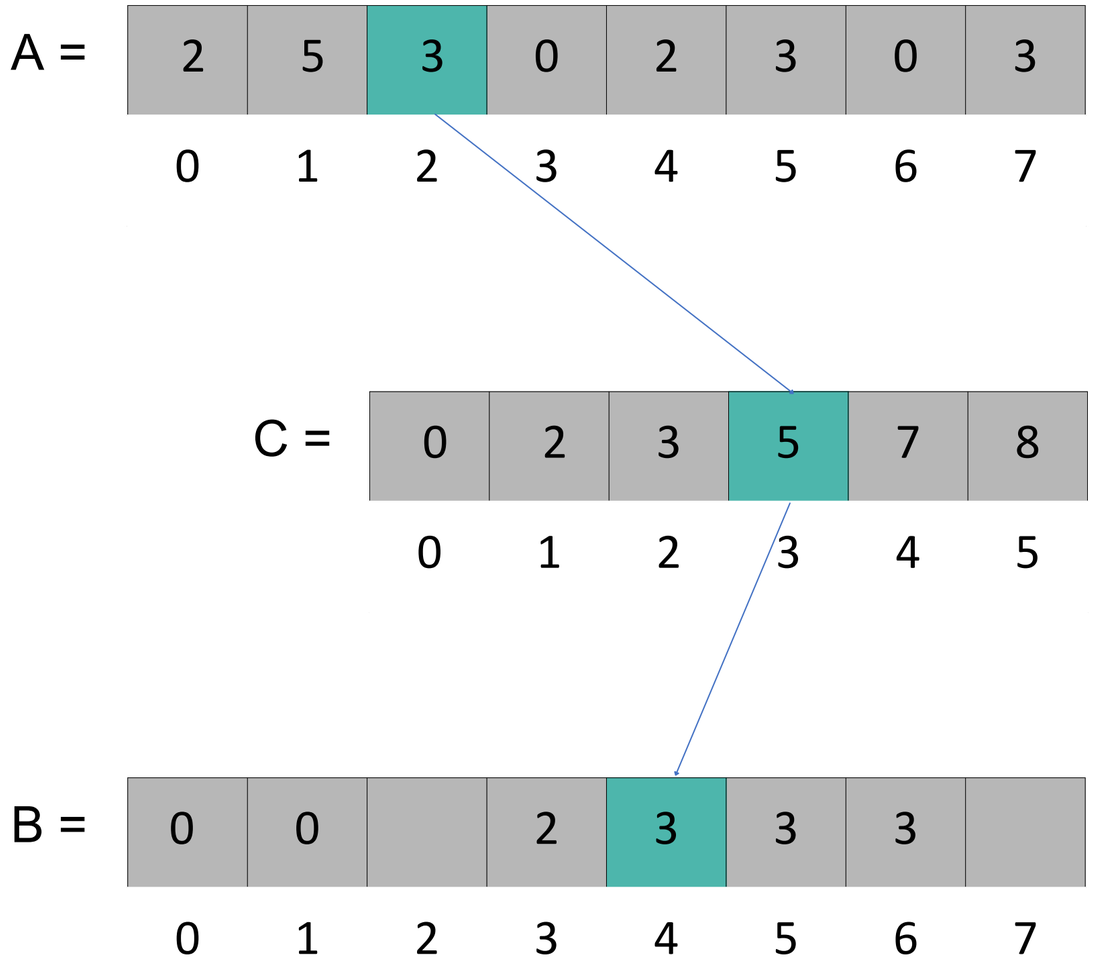
CountingSort()

input : Arreglo $A[0 \ldots n-1]$, natural k
output: Arreglo $B[0 \ldots n-1]$

`CountingSort` (A, k):

```
1   B[0...n-1] ← arreglo vacío de n celdas
2   C[0...k] ← arreglo vacío de k+1 celdas
3   for i = 0...k :
4       C[i] ← 0
5   for j = 0...n-1 :
6       C[A[j]] ← C[A[j]] + 1
7   for p = 1...k :
8       C[p] ← C[p] + C[p-1]
9   for r = n-1...0 :
10      B[C[A[r]] - 1] ← A[r]
11      C[A[r]] ← C[A[r]] - 1
12  return B
```

r = 2



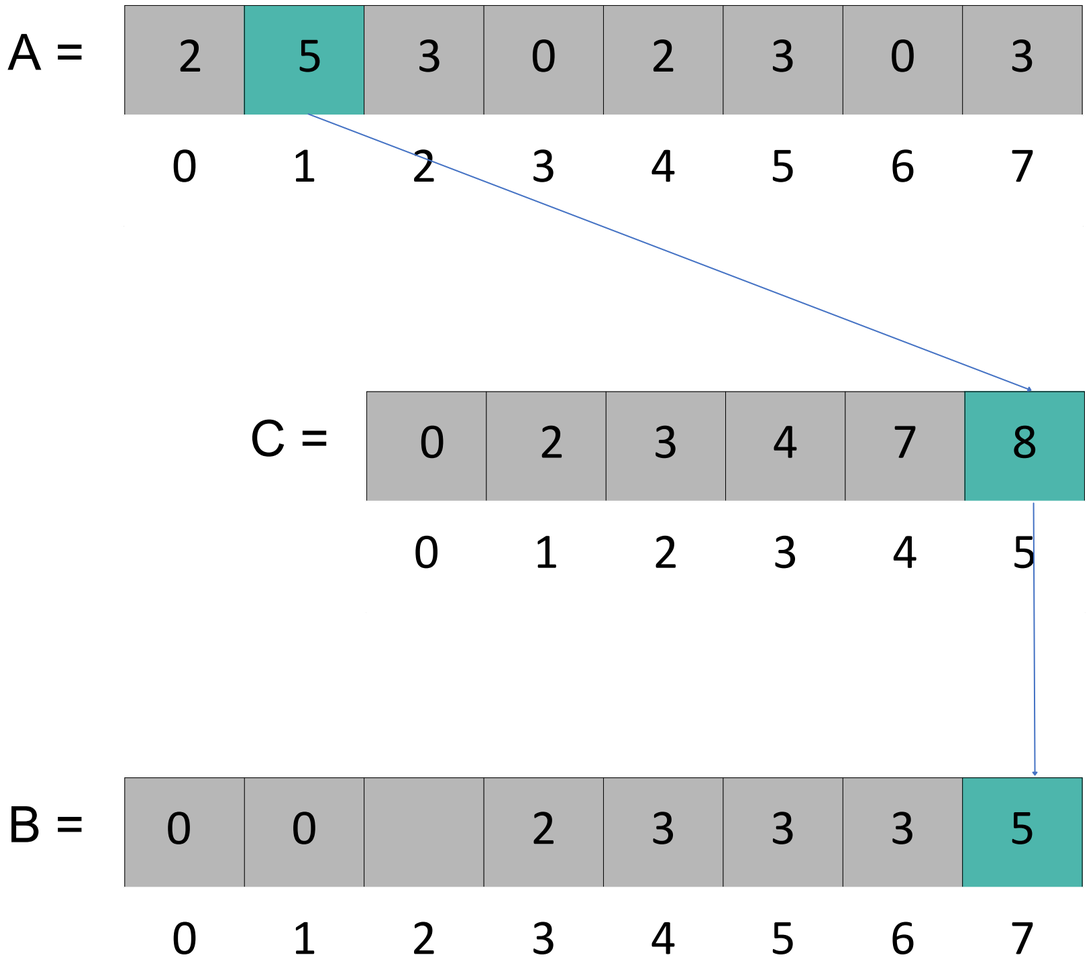
A[r] = 3 C[A[r]] = 5 B[C[A[r]]-1] = B[4] ← 3

CountingSort()

input : Arreglo $A[0 \ldots n-1]$, natural k
output: Arreglo $B[0 \ldots n-1]$

```
CountingSort (A, k):
1    B[0...n-1] ← arreglo vacío de n celdas
2    C[0...k] ← arreglo vacío de k+1 celdas
3    for i = 0...k :
4        C[i] ← 0
5    for j = 0...n-1 :
6        C[A[j]] ← C[A[j]] + 1
7    for p = 1...k :
8        C[p] ← C[p] + C[p-1]
9    for r = n-1...0 :
10       B[C[A[r]] - 1] ← A[r]
11       C[A[r]] ← C[A[r]] - 1
12   return B
```

r = 1



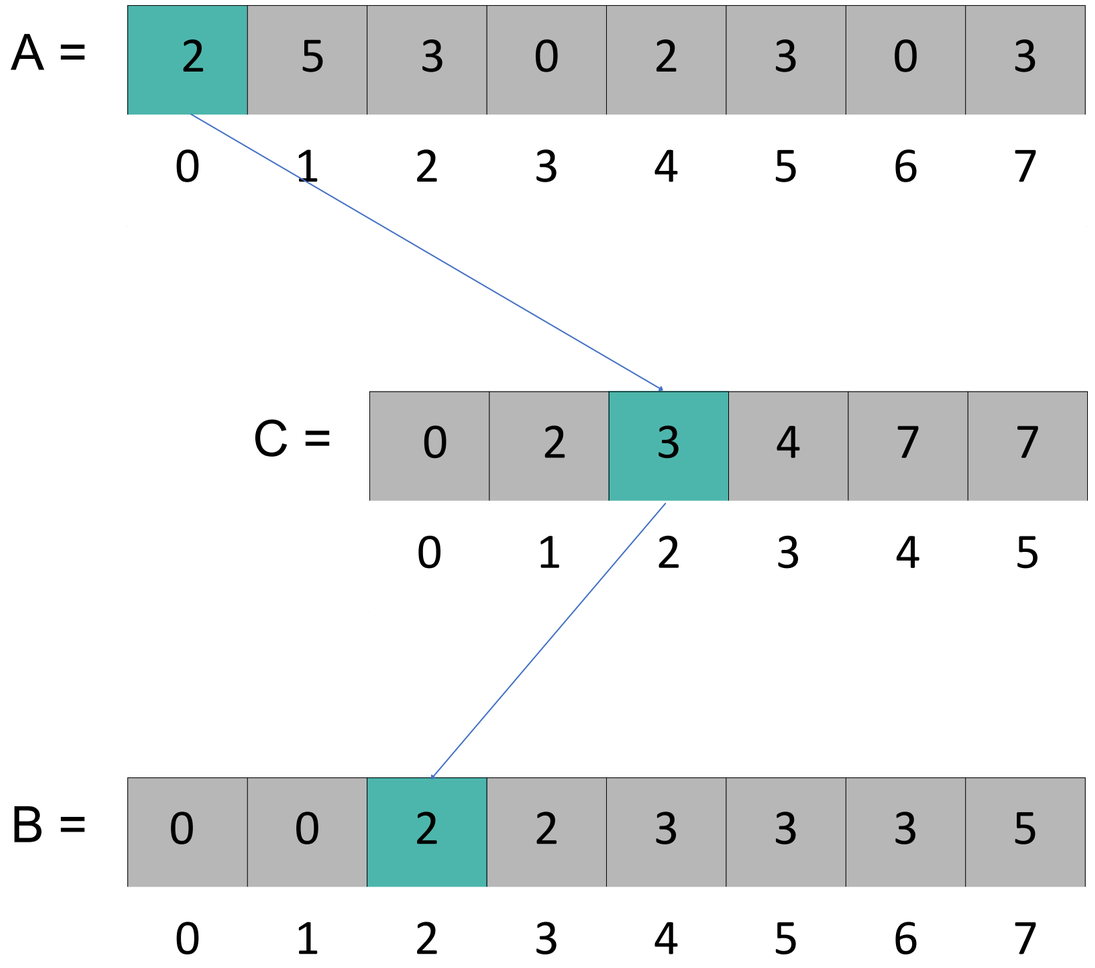
A[r] = 5 C[A[r]] = 8 B[C[A[r]]-1] = B[7] ← 5

CountingSort()

input : Arreglo $A[0 \ldots n-1]$, natural k
output: Arreglo $B[0 \ldots n-1]$

```
CountingSort (A, k):
1    B[0...n-1] ← arreglo vacío de n celdas
2    C[0...k] ← arreglo vacío de k+1 celdas
3    for i = 0...k :
4        C[i] ← 0
5    for j = 0...n-1 :
6        C[A[j]] ← C[A[j]] + 1
7    for p = 1...k :
8        C[p] ← C[p] + C[p-1]
9    for r = n-1...0 :
10       B[C[A[r]] - 1] ← A[r]
11       C[A[r]] ← C[A[r]] - 1
12   return B
```

r = 0



A[r] = 2 C[A[r]] = 3 B[C[A[r]]-1] = B[2] ← 2

RadixSort()

```
RadixSort(A, d):
    for j = 0 . . . d − 1 :
        StableSort(A, j)
```

Ordenamos desde el dígito menos significativo (0 se refiere a unidades, 1 a decenas, 2 a centenas... y así)

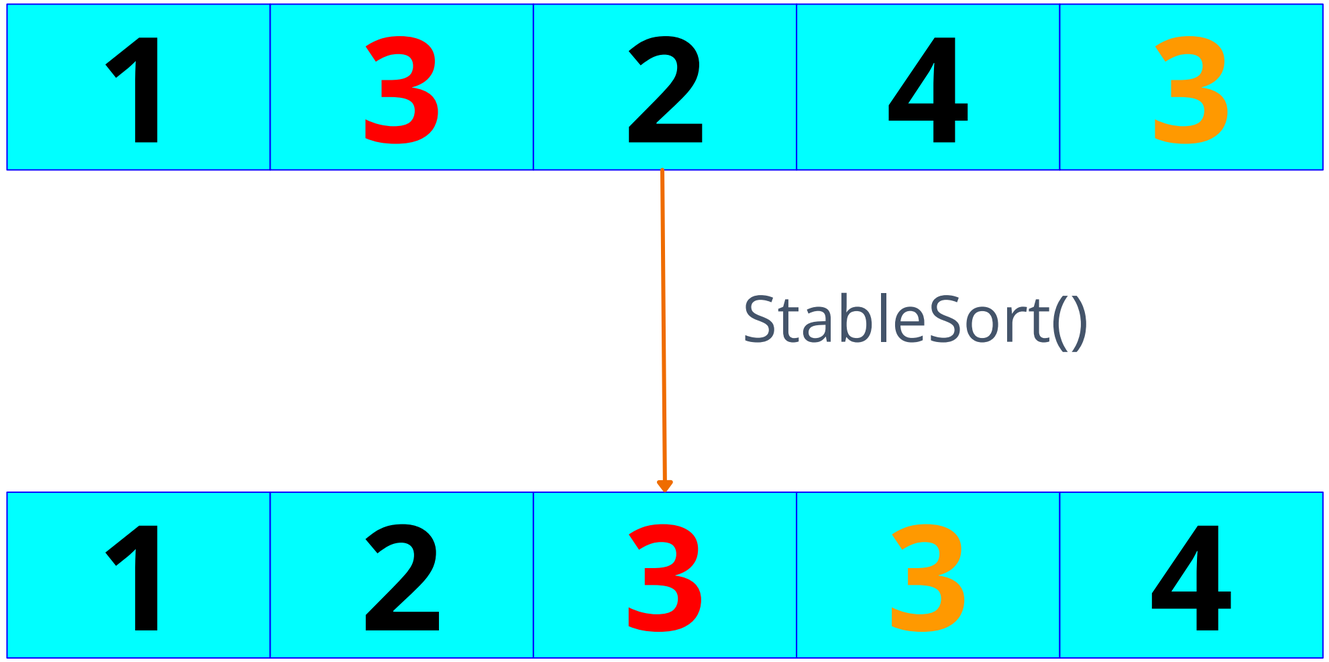
Se necesita un algoritmo de ordenamiento que sea **estable** como subrutina (CountingSort nos sirve!)

Un algoritmo es estable cuando los elementos que tenían el mismo valor antes de ordenar, mantienen su mismo orden original entre sí después de la ejecución del algoritmo.

Complejidad:
O(d(n+k))

Ejemplo estabilidad

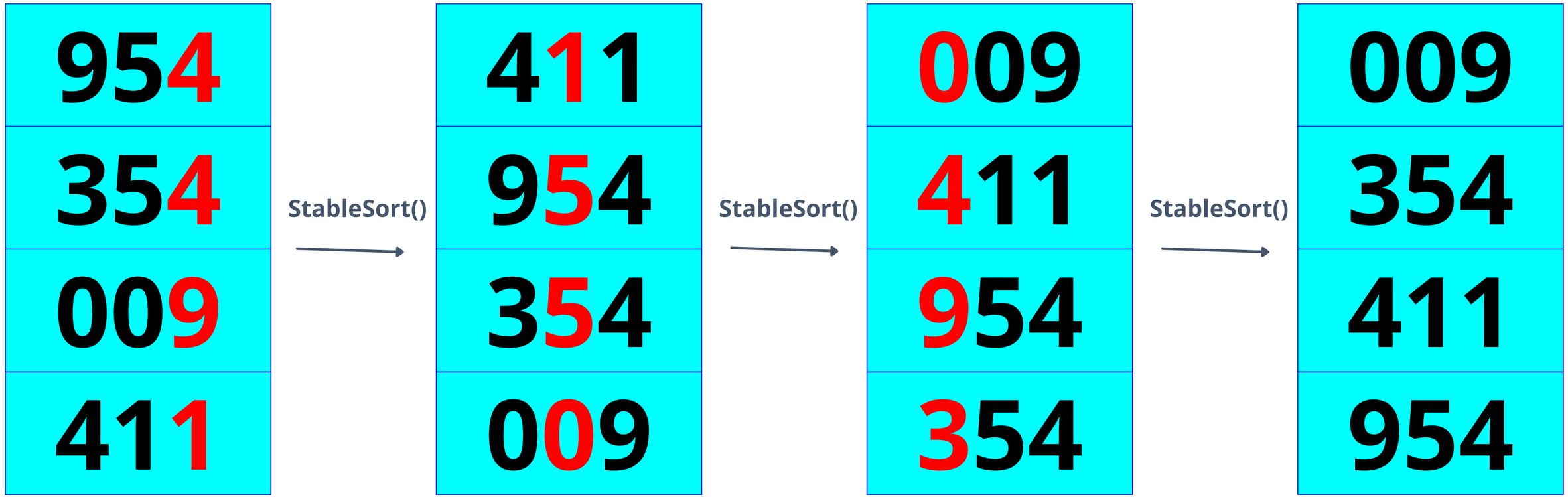
Sea StableSort() un algoritmo de ordenación estable:



Notar que el orden entre el 3 rojo y el 3 naranjo se mantiene después de la ejecución del algoritmo de ordenación estable

Para los números con menos dígitos que el número con mayor cantidad de dígitos en el arreglo, podemos considerar la ausencia de un dígito determinado como 0 (Ver 009 en el ejemplo)

RadixSort()



Ejercicio 3

P3 I2-2022-2

Para procesar la información obtenida desde el telescopio espacial James Webb, los objetos del campo de observación son identificados utilizando el Guide Star Catalog (GSC) el cual asocia a cada objeto un identificador de la forma: `fffff0nnnnn` en que `fffff` es un valor alfanumérico (0-9,A-Z) que identifica la región del espacio en que se encuentra el objeto y `nnnnn` es un correlativo del objeto en esa región (el 0 se usa como separador). El procesamiento requiere ordenar muy eficientemente (con eficiencia lineal ya que se deben ordenar 100 mil objetos distintos cada vez) los datos obtenidos desde múltiples campos de observación distintos, utilizando como criterio de orden dicho identificador.

P3 I2-2022-2

(a) [2 ptos.] Aplique RadixSort con CountingSort para realizar lo solicitado, detallando los ajustes necesarios al pseudo código visto en clases para que sean adecuados a las características del problema planteado.

P3 I2-2022-2

Solución: El orden es el “natural”, menos significativo a la derecha y más significativo a la izquierda (de 0 a d−1 con d = 11). Modifica Radix sort para incorporar la cardinalidad del dominio del dígito a ordenar en la llamada a counting sort, y modifica counting sort para utilizar dicho parámetro:

```
RadixSort (A, d) :
1.    for j = 0 ... d − 1 :
2.        if j ≥ 0 ∧ j < 6 :     ▷ rango numérico, incluye al 0 central
3.            CountSort (A, j, 10)
4.        else:     ▷ rango alfanumérico
5.            CountSort (A, j, 37)

input : Arreglo A[0 ..., n − 1], j dígito a ordenar de A, k valores posibles
CountSort (A, j, k) :
1.    B[0 ... n − 1] ← arreglo vacío
2.    C[0 ... k] ← arreglo vacío
3.    for i = 0 ... k :
4.        C[i] ← 0
5.    for m = 0 ... n − 1 :
6.        C[A[m][j]] ← C[A[m][j]] + 1    ▷ A[m][j] dígito j de la palabra m
7.    for p = 1 ... k :
8.        C[p] ← C[p] + C[p − 1]
9.    for r = n − 1 ... 0 :
10.       B[C[A[r][j]]] ← A[r]
11.       C[A[r][j]] ← C[A[r][j]] − 1
12.   for q = 0 ... n − 1 :
13.       A[q] ← B[q]
```

P3 I2-2022-2

(b) [2 ptos.] Utilizando el resultado de (a) proponga el pseudo código para permitir seleccionar si se desea ordenar en forma ascendente o descendente, manteniendo la complejidad de tiempo en $\mathcal{O}(n)$.

P3 I2-2022-2

(b) [2 ptos.] Utilizando el resultado de (a) proponga el pseudo código para permitir seleccionar si se desea ordenar en forma ascendente o descendente, manteniendo la complejidad de tiempo en $\mathcal{O}(n)$.

Solución.

Usando la parte (a) se obtiene el orden ascendente en $\mathcal{O}(n)$, si piden el orden descendente basta con recorrer el resultado de (a) y entregarlo en orden inverso, esto también es $\mathcal{O}(n)$ y como se "suma", el desempeño global sigue en $\mathcal{O}(n)$.

P3 I2-2022-2

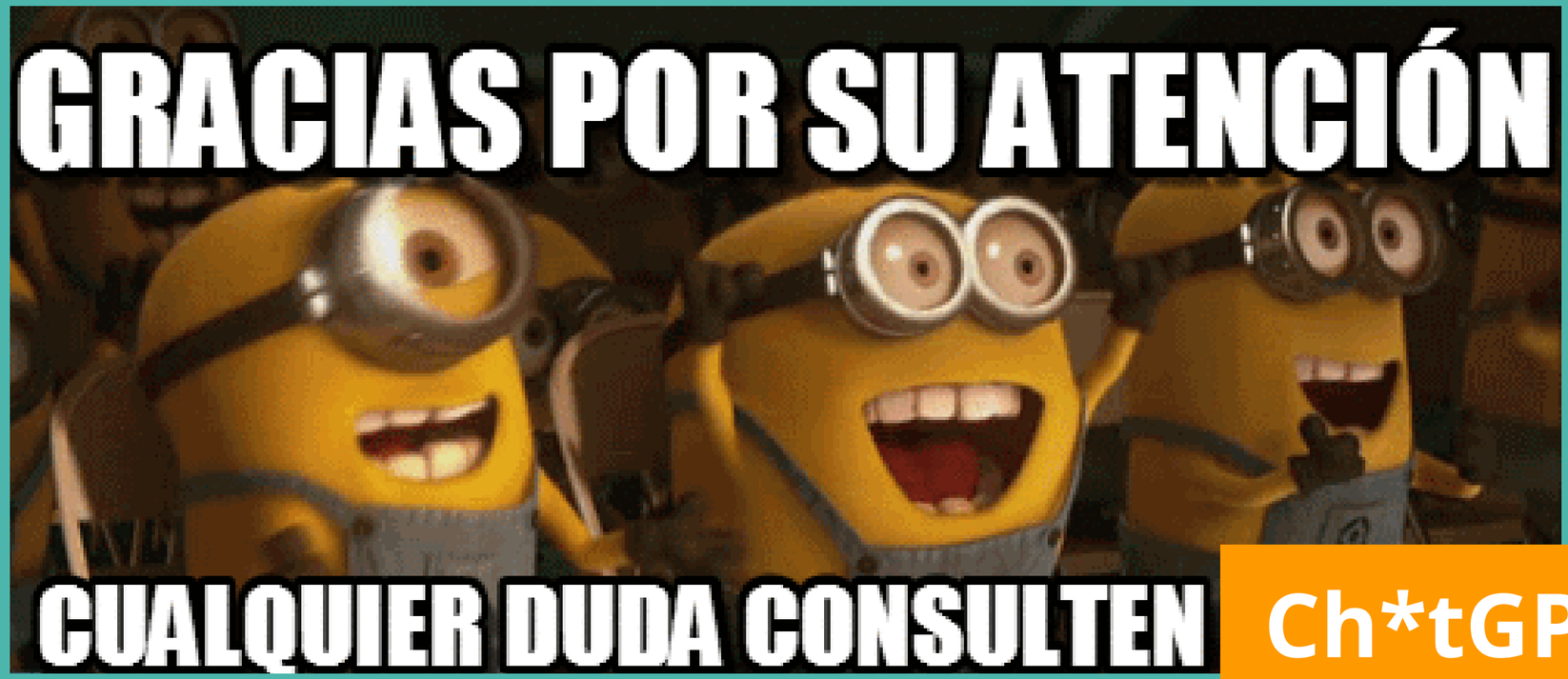
(c) [2 ptos.] Un compañero sugiere reemplazar CountingSort por QuickSort. Detalle el impacto que esto tendría desde el punto de la correctitud del algoritmo y su desempeño.

P3 I2-2022-2

(c) [2 ptos.] Un compañero sugiere reemplazar CountingSort por QuickSort. Detalle el impacto que esto tendría desde el punto de la correctitud del algoritmo y su desempeño.

Solución.

QuickSort es $\mathcal{O}(n\log(n))$ y no es estable, luego el algoritmo si bien termina (al ordenar todos los dígitos) no cumple su propósito, ya que no entrega la salida ordenada correctamente. Además el desempeño pasa de ser $\mathcal{O}(n+d) = \mathcal{O}(n)$ a ser $\mathcal{O}(n\log(n)+d) = \mathcal{O}(n\log(n)+d)$.


GRACIAS POR SU ATENCIÓN
CUALQUIER DUDA CONSULTEN
Ch*tGPT