Clase Práctica Sorting

1C2025
Belén Alvariñas

Pequeño repaso

Algoritmo	¿Complejidad en peor caso?😯	Estable
Insertion Sort		
Selection Sort		
Quick Sort		
Merge Sort		
Heap Sort		

Pequeño repaso

Algoritmo	¿Complejidad en peor caso?😯	Estable
Insertion Sort	$O(n^2)$	
Selection Sort		
Quick Sort		
Merge Sort		
Heap Sort		

Pequeño repaso

Algoritmo	¿Complejidad en peor caso?😯	Estable
Insertion Sort	$O(n^2)$	
Selection Sort	$O(n^2)$	
Quick Sort		
Merge Sort		
Heap Sort		

Pequeño repaso

Algoritmo	¿Complejidad en peor caso?😯	Estable
Insertion Sort	$O(n^2)$	
Selection Sort	$O(n^2)$	
Quick Sort	$O(n^2)$	
Merge Sort		
Heap Sort		

Pequeño repaso

Algoritmo	¿Complejidad en peor caso?😯	Estable
Insertion Sort	$O(n^2)$	
Selection Sort	$O(n^2)$	
Quick Sort	$O(n^2)$	
Merge Sort	$O(n.\log n)$	
Heap Sort		

Pequeño repaso

Algoritmo	Complejidad en peor caso	¿Estable?😯
Insertion Sort	$O(n^2)$	
Selection Sort	$O(n^2)$	
Quick Sort	$O(n^2)$	
Merge Sort	$O(n.\log n)$	
Heap Sort	$O(n.\log n)$	

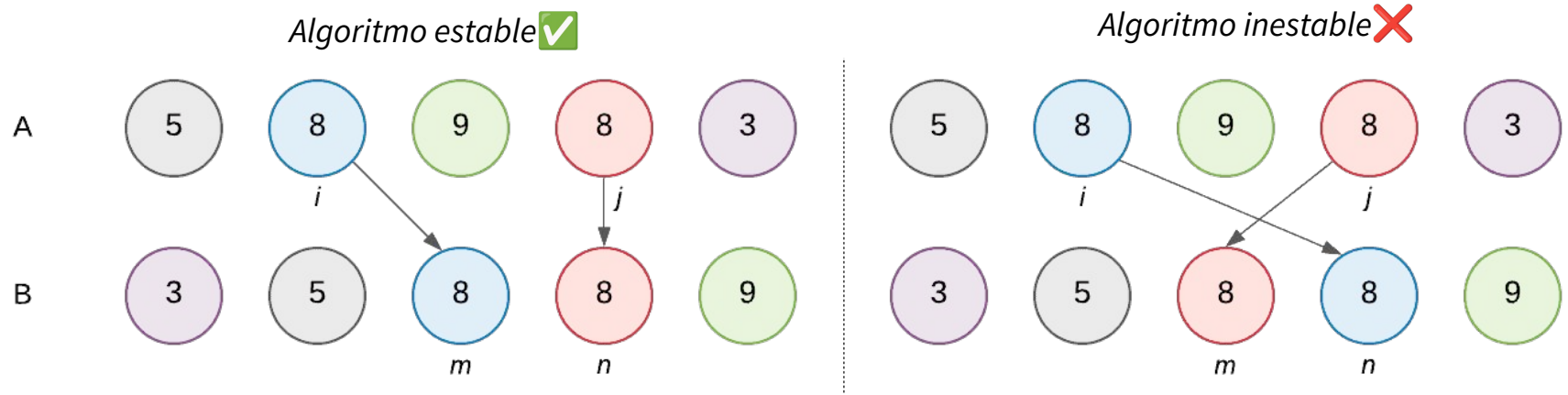
Pequeño repaso

¿Qué significa que un algoritmo sea estable?

Pequeño repaso

¿Qué significa que un algoritmo sea estable?

Un algoritmo de ordenamiento es estable si dos registros i y j con claves iguales mantienen su orden relativo luego del ordenamiento.



Pequeño repaso

Algoritmo	**Complejidad en peor caso**	**¿Estable?😲**
Insertion Sort	$O(n^2)$	
Selection Sort	$O(n^2)$	
Quick Sort	$O(n^2)$	
Merge Sort	O(n.log n)	
Heap Sort	O(n.log n)	

Pequeño repaso

Algoritmo	**Complejidad en peor caso**	**¿Estable?😲**
Insertion Sort	$O(n^2)$	Sí
Selection Sort	$O(n^2)$	
Quick Sort	$O(n^2)$	
Merge Sort	$O(n.\log n)$	
Heap Sort	$O(n.\log n)$	

Pequeño repaso

Algoritmo	**Complejidad en peor caso**	**¿Estable?😲**
Insertion Sort	$O(n^2)$	Sí
Selection Sort	$O(n^2)$	No
Quick Sort	$O(n^2)$	
Merge Sort	$O(n.\log n)$	
Heap Sort	$O(n.\log n)$	

Pequeño repaso

Algoritmo	**Complejidad en peor caso**	**¿Estable?😲**
Insertion Sort	$O(n^2)$	Sí
Selection Sort	$O(n^2)$	No
Quick Sort	$O(n^2)$	No
Merge Sort	$O(n.\log n)$	
Heap Sort	$O(n.\log n)$	

Pequeño repaso

Algoritmo	**Complejidad en peor caso**	**¿Estable?😲**
Insertion Sort	$O(n^2)$	Sí
Selection Sort	$O(n^2)$	No
Quick Sort	$O(n^2)$	No
Merge Sort	O(n.log n)	Sí
Heap Sort	O(n.log n)	

Pequeño repaso

Algoritmo	Complejidad en peor caso	Estable
Insertion Sort	$O(n^2)$	Sí
Selection Sort	$O(n^2)$	No
Quick Sort	$O(n^2)$	No
Merge Sort	$O(n.\log n)$	Sí
Heap Sort	$O(n.\log n)$	No

Sobre los algoritmos

Estos algoritmos ordenan arreglos comparando los elementos, es decir, son algoritmos de sorting por comparación.

Vieron en la teórica que en el peor caso son $\Omega(n \log n)$.

¿Puede haber algo mejor?

Sí😁, por ejemplo, si tenemos **información** de los elementos.

Bucket Sort

¿Qué información tenemos?

Que los elementos pueden separarse (según algún criterio) en **M** categorías ordenables. Es decir, para $i < j$, todo elemento de la categoría i es menor que toda elemento de la categoría j.

Idea:

1) Construir un arreglo de M listas y guardar los elementos de la categoría i en la i-ésima lista.
2) Ordenar las M listas por separado (en caso que se requiera).
3) Reconstruir el arreglo A, concatenando las listas en orden.

Bucket Sort : Ejemplo

input:

101	95	8	52	82	15	38	125	5	11	75	3

M = 4

Bucket Sort : Ejemplo

input:

101	95	8	52	82	15	38	125	5	11	75	3

M = 4

Paso 1:

8	5	3

0-10

15	38	11

11-50

95	52	82	75

51-100

101	125

101-150

Bucket Sort : Ejemplo

Paso 2:

3	5	8

0-10

11	15	38

11-50

52	75	82	95

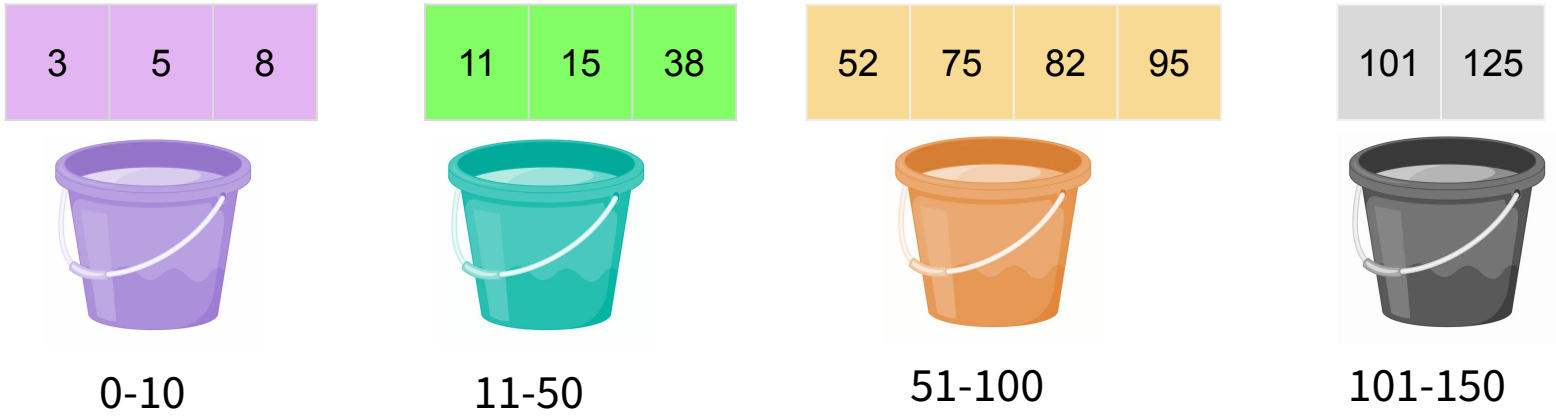
51-100

101	125

101-150

Bucket Sort : Ejemplo

Paso 2:



Paso 3:

output:

3	5	8	11	15	38	52	75	82	95	101	125

Bucket Sort

Algorithm 1 BUCKET-SORT(A, M)

$n \leftarrow length(A)$
$B \leftarrow$ arreglo de M listas vacías
for $i \leftarrow [0..n-1]$ **do**
    insertar $A[i]$ en la lista $B[h(A[i])]$
end for
for $j \leftarrow [0..M-1]$ **do**
    ordenar lista $B[j]$
end for
concatenar listas $B[0], B[1], \ldots, B[M-1]$

h(x) es una función que indica la categoría de x (de 0 a M − 1). Suponemos que se ejecuta en O(1).

Bucket Sort

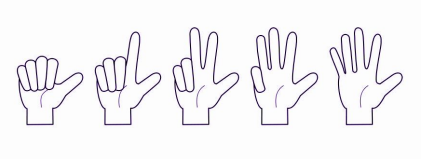
¿Complejidades?

Algorithm 1 BUCKET-SORT(A, M)

$n \leftarrow length(A)$
$B \leftarrow$ arreglo de M listas vacías
for $i \leftarrow [0..n-1]$ **do**
    insertar $A[i]$ en la lista $B[h(A[i])]$
end for
for $j \leftarrow [0..M-1]$ **do**
    ordenar lista $B[j]$
end for
concatenar listas $B[0], B[1], \ldots, B[M-1]$

Complejidad : O (n + M) + O(ordenar los buckets)

Counting Sort



¿Qué información tenemos?

Asume que los elementos de un arreglo A de naturales son **menores** que un cierto valor **k**.

Idea:

1) Construir un arreglo B de tamaño k y guardar en la posición i, la cantidad de apariciones de i en A.
2) Reconstruir el arreglo A, poniendo tantas copias de cada valor $j \in \{0..k\}$, como lo indique B[j].

Counting Sort : Ejemplo

A =

7	1	4	1	4	2	12	1

k = 12

Counting Sort : Ejemplo

A =

7	1	4	1	4	2	12	1

k = 12

Paso 1 :

B =

0	3	1	0	2	0	0	1	0	0	0	0	1

Counting Sort : Ejemplo

A =

7	1	4	1	4	2	12	1

k = 12

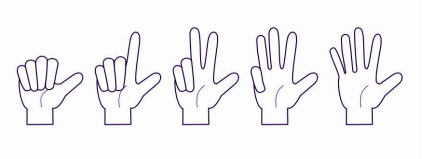
Paso 1 : **B =**

0	3	1	0	2	0	0	1	0	0	0	0	1

Paso 2 : **output =**

1	1	1	2	4	4	7	12

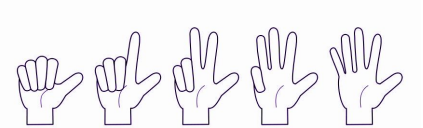
Counting Sort



Algorithm 2 COUNTING-SORT(A: arreglo(nat), k: nat)

```
1:  B ← arreglo de tamaño k
2:  for i ← [0..k − 1] do
3:      B[i] ← 0
4:  end for
5:  //cuento la cantidad de apariciones de cada elemento.
6:  for j ← [0..length(A) − 1] do
7:      B[A[j]] ← B[A[j]] + 1
8:  end for
9:  //inserto en A la cantidad de apariciones de cada elemento.
10: indexA ← 0
11: for i ← [0..k − 1] do
12:     for j ← [1..B[i]] do
13:         A[indexA] ← i
14:         indexA ← indexA + 1
15:     end for
16: end for
```

Counting Sort



¿Complejidades?

Algorithm 2 COUNTING-SORT(A: arreglo(nat), k: nat)

```
1:  B ← arreglo de tamaño k
2:  for i ← [0..k − 1] do
3:      B[i] ← 0
4:  end for
5:  //cuento la cantidad de apariciones de cada elemento.
6:  for j ← [0..length(A) − 1] do
7:      B[A[j]] ← B[A[j]] + 1
8:  end for
9:  //inserto en A la cantidad de apariciones de cada elemento.
10: indexA ← 0
11: for i ← [0..k − 1] do
12:     for j ← [1..B[i]] do
13:         A[indexA] ← i
14:         indexA ← indexA + 1
15:     end for
16: end for
```

Complejidad : O(n + k)

Radix Sort (versión naturales)

¿Qué información tenemos?

Que son números naturales, en una base *b* y que se pueden descomponer en sus cifras significativas.

Idea:

1) Ordenar los valores mirando solo el dígito 1
2) Ordenar los valores mirando solo el dígito 2 (manteniendo el orden en caso de empate)
3) Ordenar los valores mirando solo el dígito 3 (manteniendo el orden en caso de empate)
4) …

Radix Sort (versión naturales) : Ejemplo

input:

325	19	426	17	445	123	175

Radix Sort (versión naturales) : Ejemplo

input:

325	19	426	17	445	123	175

12**3**
32**5**
44**5**
17**5** ⟶
42**6**
1**7**
1**9**

paso 1

Radix Sort (versión naturales) : Ejemplo

input:

325	19	426	17	445	123	175

123
325
445
175
426
17
19

paso 1

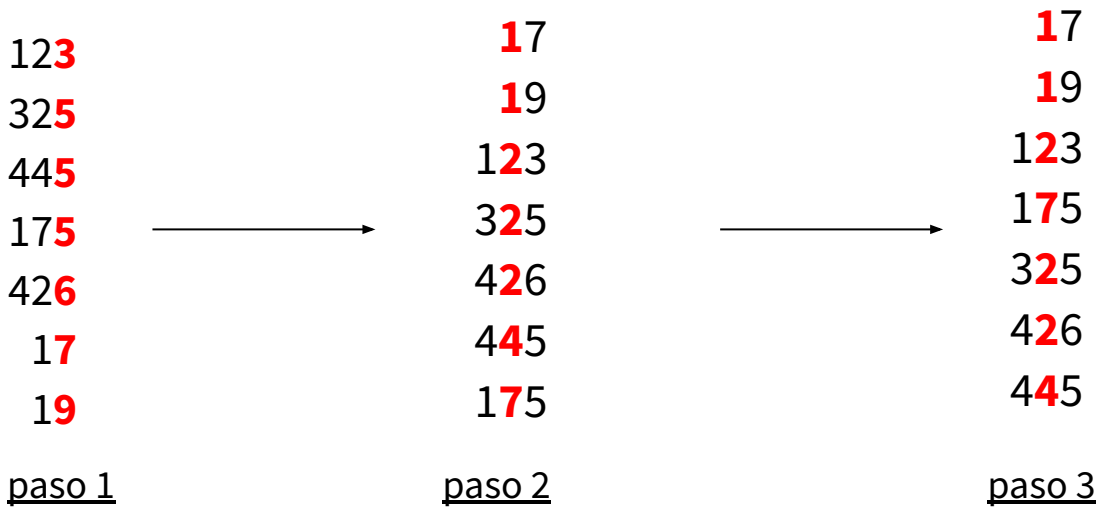
17
19
123
325
426
445
175

paso 2

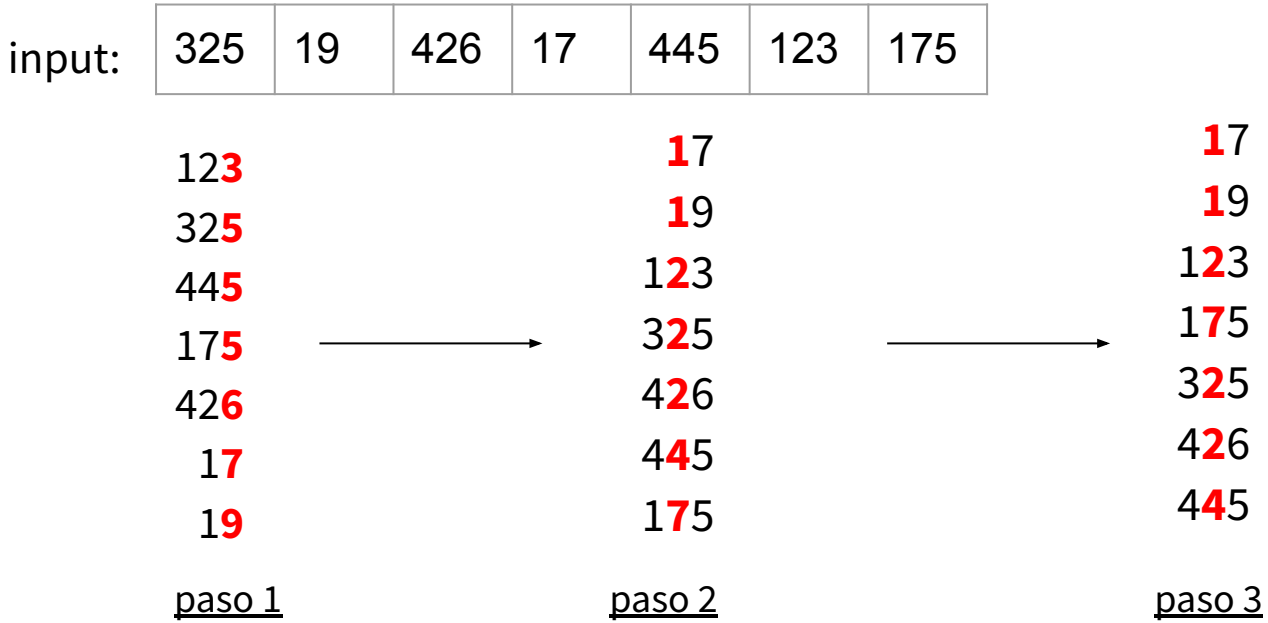
Radix Sort (versión naturales) : Ejemplo

input:

325	19	426	17	445	123	175



Radix Sort (versión naturales) : Ejemplo



Radix Sort (versión naturales)

Algorithm 3 RADIX-SORT(A, d)

```
for i ← [1..d] // del menos significativo al más significativo do
    Ordenar el arreglo A según el dígito i, en forma estable
end for
```

Llamamos *d* a la cantidad máxima de dígitos de los elementos y suponemos que el dígito 1 es el menos significativo de cada valor.

¿Complejidades?

Complejidad : O(d × ordenar A por un dígito).

Con Bucket Sort (sin ordenar los buckets): O(n · d).

Radix Sort Generalizado

- Radix Sort puede usarse también para ordenar **cadenas de caracteres** (la primera letra es la más significativa).
- El mismo esquema sirve para ordenar **tuplas**, donde el orden que se quiera dar depende principalmente de un “dígito” (i.e., un campo de la tupla) y en el caso de empate se tengan que revisar los otros.
- La idea en el fondo es la misma: usar un algoritmo estable e ir ordenado por “dígito” (del menos al más significativo).

Algorithm 4 RADIX-SORT(A, d)

```
for i ← [1..d] // del menos significativo al más significativo do
    Ordenar el arreglo A según el dígito i, en forma estable
end for
```

¡Ejercicios!

Ejercicio 18. Dado un arreglo de pares `<estudiante, nota>`, queremos devolver un arreglo de pares `<estudiante, promedio>`, ordenado en forma descendente por promedio y, en caso de empate, por número de libreta.

Los estudiantes se representan con un número de libreta (puede considerar que es un número de a lo sumo 10 dígitos), y la nota es un número entre 1 y 100. La cantidad de estudiantes se considera no acotada.

Se pide dar un algoritmo que resuelva el problema con complejidad $O(n + m \log m)$, donde n es la cantidad total de notas (es decir, el tamaño del arreglo de entrada), y m es la cantidad total de estudiantes.

Ejemplo:

```
Entrada: <12395, 72>, <45615, 81>, <12395, 94>, <45615, 100>, <78920, 81>,
         <12395, 90>, <45615, 71>, <78920, 50>
Salida: <12395, 85.3333>, <45615, 84>, <78920, 65.5>
```

a) Describa cada etapa del algoritmo con sus palabras, y justifique por qué cumple con las complejidades pedidas.

b) Escriba el algoritmo en pseudocódigo. Puede utilizar (sin reescribir) todos los algoritmos y estructuras de datos vistos en clase.

¡¡Recreo!!

Ejercicio de Parcial 🐕

b) Se está por realizar un gran censo de perros en la Ciudad de Buenos Aires. La información recaudada llegará en forma de un arreglo de tuplas `<Nombre, Raza, Edad, Peso>`. Las razas son `strings` de largo acotado L, pero no sabemos cuántas razas pueden existir, aunque sí sabemos que entre ellas está la mejor raza de todas: raza Perro. Las edades serán enteros positivos entre 1 y 20 y los pesos serán de tipo `float`.

Proponga un algoritmo que organice la información en dos arreglos:

- Uno con los perros raza Perro ordenados por edad y, en caso de empate, por peso, ambos en forma ascendente
- Otro arreglo con la información de las razas y la cantidad de perros de cada una en forma de tuplas `<Raza, int>`, ordenadas en forma decreciente por la cantidad y, en caso de empate, alfabéticamente por la raza

La complejidad temporal del algoritmo debe ser $O(n + p.log(p))$, donde n es el total de perros y p el total de perros raza Perro.

Ejemplo

Si recibimos:

```
[ <Nulus, Perro, 4, 15.3>, <Joe, Cocker, 14, 8.0>, <Lady, Caniche, 10, 6.1>, <Dobby, Perro, 7, 18.3>
<Res, Dogo, 3, 50.0>, <Haru, Perro, 7, 30.0>, <Clementina, Perro, 14, 10.0>, <Choqui, Caniche, 16, 5.0>]
```

Deberíamos devolver:

```
P: [ <Nulus, 4, 15.3>, <Dobby, 7, 18.3>, <Haru, 7, 30.0>, <Clementina, 14, 10.0>]

R: [ <Caniche, 2>, <Cocker, 1>, <Dogo, 1>]
```