

# Ordenación lineal

Clase 7

IIC 2133 - Sección 3

Prof. Eduardo Bustos

# Sumario

**Introducción**

Ordenación lineal

Cierre

# Construcción de un heap

La inserción que vimos permite agregar un **único** elemento a un heap  $H$  preexistente

Si tenemos un arreglo  $A$  y queremos obtener un heap podemos usar una de dos estrategias

1. Iterar para cada elemento de  $A$ , insertando sobre un heap originalmente vacío
2. Utilizar `SiftDown` para ciertos elementos de  $A$

Esta última forma es *in place* y sencilla

# Construcción de un heap

**input** : arreglo  $A[0 \dots n-1]$

BuildHeap( $A$ ):

**for**  $i = \lfloor n/2 \rfloor - 1 \dots 0$  :      ▷ loop decreciente

        SiftDown( $A, i$ )

**Observación:** los elementos de  $A$  en los cuales no se llama directamente SiftDown son hojas del último nivel del árbol

Respecto a su complejidad

- La complejidad asintótica *directa* es  $\mathcal{O}(n \log(n))$
- Se puede demostrar que una mejor cota es  $\mathcal{O}(n)$

BuildHeap deja  $A$  como un heap en tiempo  $\mathcal{O}(n)$

# Heaps para ordenar

Ya sabemos crear un heap a partir de un arreglo cualquiera

Y sabemos la propiedad de heap: cada nodo es estrictamente mayor que sus descendientes

¿Podemos aprovechar estos hechos para ordenar un arreglo  $A$ ?

# Ordenando con heaps

Dado un heap  $H$

- Su raíz es estrictamente mayor a todos los otros nodos
- Debe ser el último elemento del arreglo ordenado

Si sabemos que el último elemento del arreglo luego del intercambio **está ordenado**

- No queremos moverlo más
- Es decir, reducimos el **tamaño del heap**
- A este parámetro le llamamos  $A.heap\_size$

Cambiamos el tamaño del heap para que SiftDown sepa hasta dónde llegar moviendo elementos

# Ordenando con heaps

**input** : arreglo  $A[0 \dots n-1]$

HeapSort( $A$ ):

    BuildHeap( $A$ )

**for**  $i = n-1 \dots 1$  :      ▷ loop decreciente

$A[0] \rightleftharpoons A[i]$

$A.\text{heap\_size} = A.\text{heap\_size} - 1$

        ShiftDown( $A, 0$ )

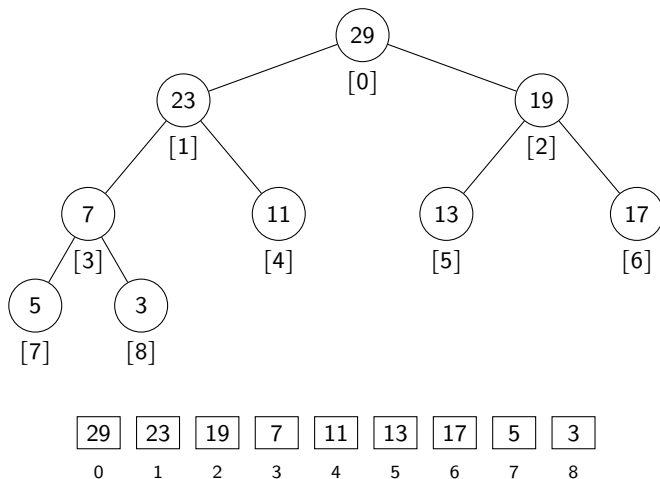
Respecto a su complejidad

- BuildHeap  $\mathcal{O}(n)$
- SiftDown se repite  $\mathcal{O}(n)$  veces  $\mathcal{O}(n \log(n))$
- Total  $\mathcal{O}(n + n \log(n)) = \mathcal{O}(n \log(n))$

HeapSort ordena en tiempo  $\mathcal{O}(n \log(n))$

# Heapsort en acción

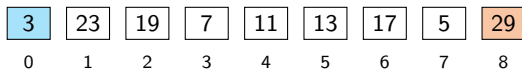
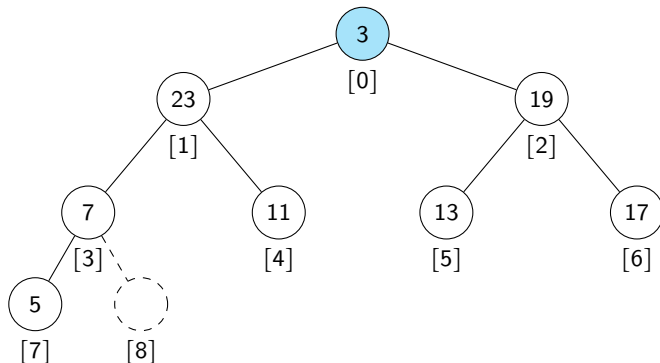
Supongamos que ya contamos con el heap resultante de BuildHeap





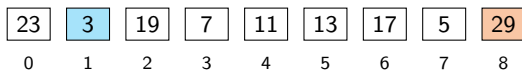
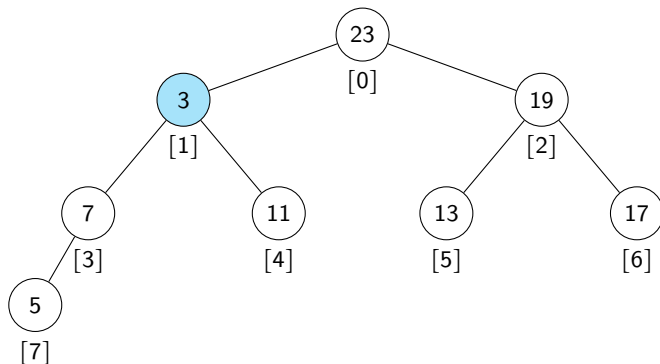
# Heapsort en acción

Movemos el primer elemento y reducimos el tamaño del heap en 1



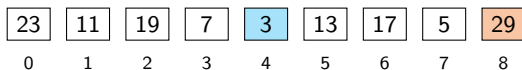
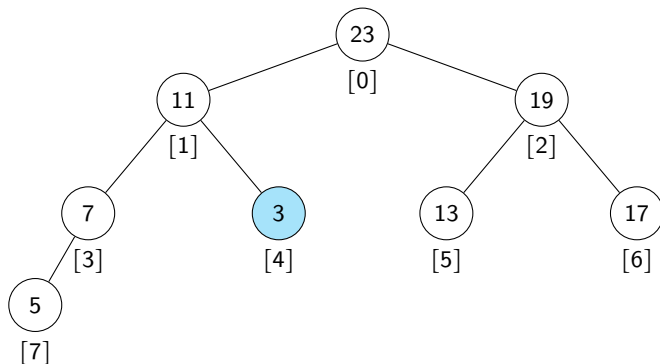
# Heapsort en acción

Aplicamos  $\text{SiftDown}(A, 0)$  (el heap es  $A[0 \dots 7]$ )



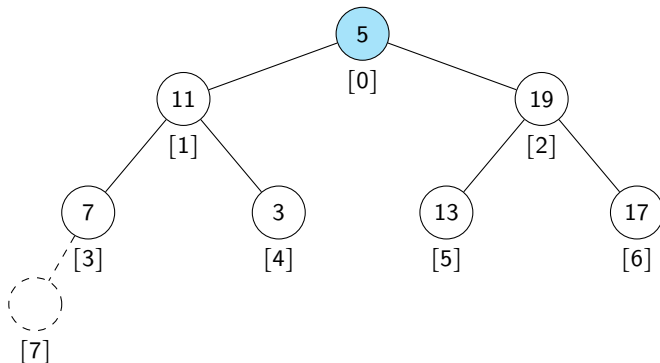
# Heapsort en acción

Aplicamos  $\text{SiftDown}(A, 1)$  (el heap es  $A[0 \dots 7]$ )



# Heapsort en acción

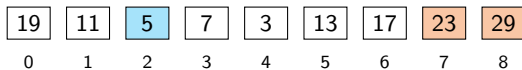
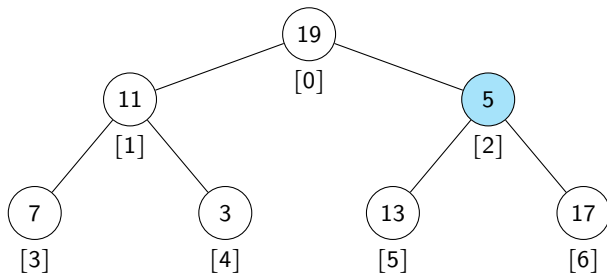
Repetimos el proceso con la nueva raíz



5	11	19	7	3	13	17	23	29
0	1	2	3	4	5	6	7	8

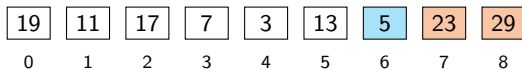
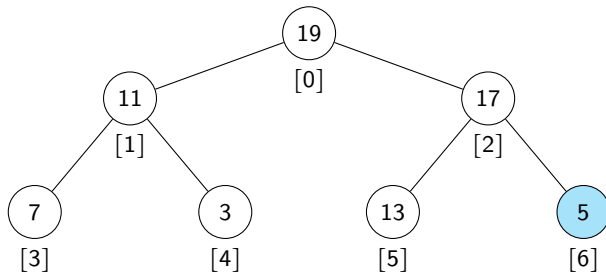
# Heapsort en acción

Aplicamos `SiftDown(A, 0)` (el heap es  $A[0 \dots 6]$ )



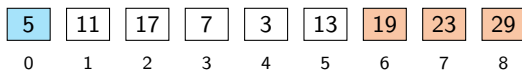
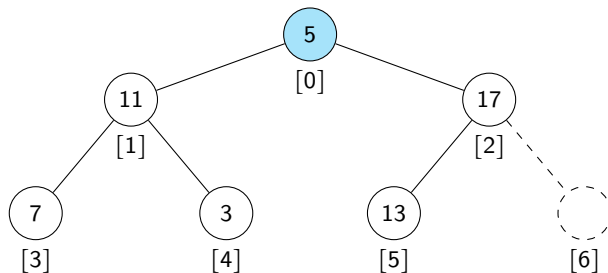
# Heapsort en acción

Aplicamos  $\text{SiftDown}(A, 2)$  (el heap es  $A[0 \dots 6]$ )



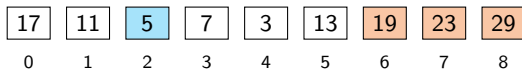
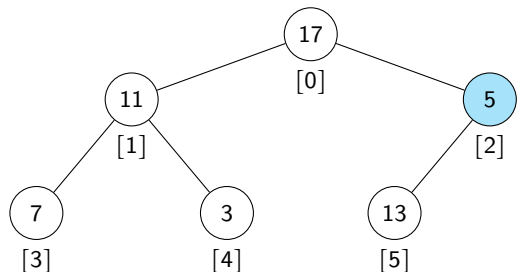
# Heapsort en acción

Repetimos el proceso con la nueva raíz



# Heapsort en acción

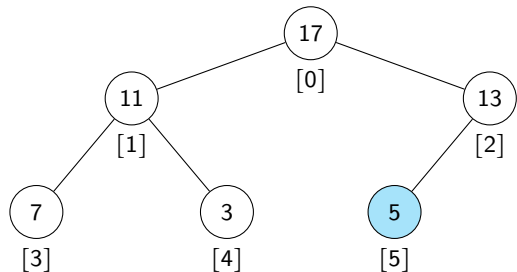
Aplicamos  $\text{SiftDown}(A, 0)$  (el heap es  $A[0 \dots 5]$ )





# Heapsort en acción

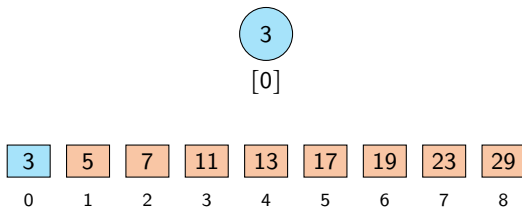
Aplicamos  $\text{SiftDown}(A, 2)$  (el heap es  $A[0 \dots 5]$ )



17	11	13	7	3	5	19	23	29
0	1	2	3	4	5	6	7	8

# Heapsort en acción

El proceso termina cuando queda solo un nodo en el heap: es el mínimo



# Complejidad de algoritmos de ordenación

Resumimos los resultados de complejidad por caso hasta el momento

Algoritmo	Mejor caso	Caso promedio	Peor caso	Memoria adicional
Selection Sort	$\mathcal{O}(n^2)$	$\mathcal{O}(n^2)$	$\mathcal{O}(n^2)$	$\mathcal{O}(1)$
Insertion Sort	$\mathcal{O}(n)$	$\mathcal{O}(n^2)$	$\mathcal{O}(n^2)$	$\mathcal{O}(1)$
Merge Sort	$\mathcal{O}(n \log(n))$	$\mathcal{O}(n \log(n))$	$\mathcal{O}(n \log(n))$	$\mathcal{O}(n)$
Quick Sort	$\mathcal{O}(n \log(n))$	$\mathcal{O}(n \log(n))$	$\mathcal{O}(n^2)$	$\mathcal{O}(1)$
Heap Sort	$\mathcal{O}(n \log(n))$	$\mathcal{O}(n \log(n))$	$\mathcal{O}(n \log(n))$	$\mathcal{O}(1)$

# Objetivos de la clase

- ☐ Conocer algoritmos de ordenación en tiempo lineal
- ☐ Comprender la limitación de dominio para tener tales algoritmos

# Sumario

Introducción

**Ordenación lineal**

Cierre

# Ordenación en tiempo lineal

Consideremos una  $A$  secuencia de  $n$  naturales entre 0 y  $k$

- Para todo  $a_i \in A$ , se tiene que  $0 \leq a_i \leq k$
- Notemos que no necesariamente  $n = k - 1$
- La secuencia  $A$  puede tener elementos repetidos

Propondremos un algoritmo de ordenación que **no compara**

- Para cada dato, contaremos cuántos datos son menores que él
- Esto nos indica la posición final de cada elemento

Si  $k \in \mathcal{O}(n)$ , entonces este algoritmo será  $\Theta(n)$

## El algoritmo CountingSort()

**input** : Arreglo  $A[0 \dots n-1]$ , natural  $k$

**output**: Arreglo  $B[0 \dots n-1]$

CountingSort ( $A, k$ ):

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

# El algoritmo CountingSort()

CountingSort ( $A, k$ ):

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

- La complejidad del algoritmo es  $\Theta(n + k)$
- Si  $k \in \mathcal{O}(n)$ , entonces CountingSort() es  $\Theta(n)$

¡Este es un mejor tiempo que  $\mathcal{O}(n \log(n))$ !



## Ejemplo de ejecución

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

$A$

7	1	1	3	0	7	5	5	7	3
0	1	2	3	4	5	6	7	8	9

Hacemos el llamado CountingSort( $A, 7$ )

# Ejemplo de ejecución

CountingSort ( $A, k$ ):

- 1  $B[0 \dots n-1] \leftarrow$  arreglo vacío
- 2  $C[0 \dots k] \leftarrow$  arreglo vacío
- 3 **for**  $i = 0 \dots k$  :
- 4      $C[i] \leftarrow 0$

$A$

7	1	1	3	0	7	5	5	7	3
0	1	2	3	4	5	6	7	8	9

$C$

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

## Ejemplo de ejecución

CountingSort ( $A, k$ ):

```
1   $B[0 \dots n-1] \leftarrow$  arreglo vacío  
2   $C[0 \dots k] \leftarrow$  arreglo vacío  
3  for  $i = 0 \dots k$  :  
4       $C[i] \leftarrow 0$   
5  for  $j = 0 \dots n-1$  :  
6       $C[A[j]] \leftarrow C[A[j]] + 1$ 
```

$A$

7	1	1	3	0	7	5	5	7	3
0	1	2	3	4	5	6	7	8	9

$C$

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

$C$

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

Hasta aquí,  $C[x]$  contiene el número de copias de  $x$  en  $A$

## Ejemplo de ejecución

CountingSort ( $A, k$ ):

```
1   $B[0 \dots n-1] \leftarrow$  arreglo vacío
2   $C[0 \dots k] \leftarrow$  arreglo vacío
3  for  $i = 0 \dots k$  :
4       $C[i] \leftarrow 0$ 
5  for  $j = 0 \dots n-1$  :
6       $C[A[j]] \leftarrow C[A[j]] + 1$ 
7  for  $p = 1 \dots k$  :
8       $C[p] \leftarrow C[p] + C[p-1]$ 
```

$A$

7	1	1	3	0	7	5	5	7	3
0	1	2	3	4	5	6	7	8	9

$C$

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

$C$

1	3	3	5	5	7	7	10
0	1	2	3	4	5	6	7

Hasta aquí,  $C[x]$  contiene cuántos elementos  
menores o iguales a  $x$  hay en  $A$

## Ejemplo de ejecución

```
9  for  $r = n - 1 \dots 0$  :  
10      $B[C[A[r]] - 1] \leftarrow A[r]$   
11      $C[A[r]] \leftarrow C[A[r]] - 1$ 
```

Para  $r = 9$

$A$

7	1	1	3	0	7	5	5	7	3
0	1	2	3	4	5	6	7	8	9

$B$

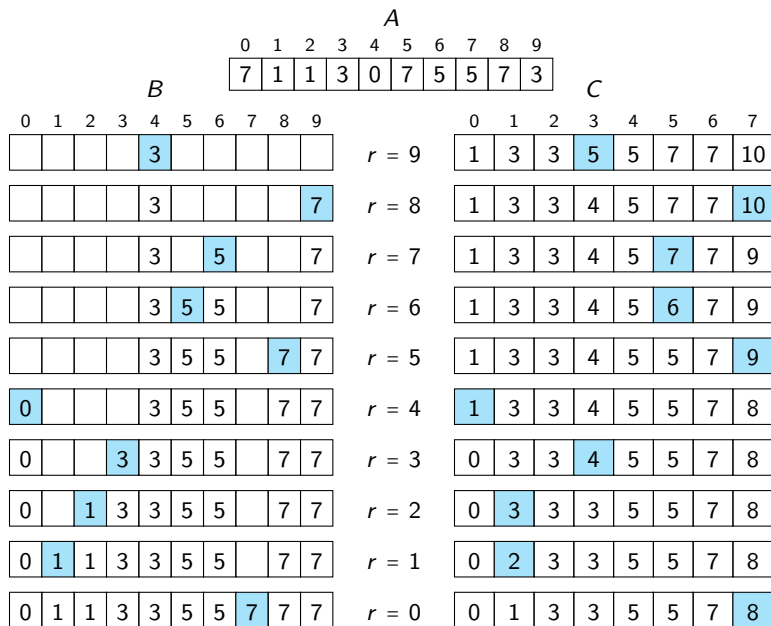
				3					
0	1	2	3	4	5	6	7	8	9

$C$

1	3	3	5	5	7	7	10
0	1	2	3	4	5	6	7

1	3	3	4	5	7	7	10
0	1	2	3	4	5	6	7

# Ejemplo de ejecución



# RadixSort

Otro algoritmo de ordenación en tiempo lineal es RadixSort

- Usado por las máquinas que ordenaban tarjetas perforadas
- Cada tarjeta tiene 80 columnas y 12 líneas
- Cada columna puede tener un agujero en una línea

# RadixSort

El algoritmo ordena las tarjetas revisando una columna determinada

- Si hay un agujero en la columna, se pone en uno de los 12 compartimientos
- Las tarjetas con la perforación en la primera columna quedan encima
- La misma idea funciona para  $d$  columnas

Podemos generalizarla para un número natural de  $d$  dígitos



# Ordenando por dígito

Consideremos un número natural de  $d$  dígitos  $n_0 n_1 \dots n_{d-1}$

- Podemos ordenar según el dígito más significativo  $n_0$
- Según este dígito, separamos los números en *compartimientos*
- Luego, ordenados recursivamente cada compartimiento por su segundo dígito más significativo  $n_1$
- Finalmente, combinamos los contenidos de cada compartimiento

Problema: posiblemente muchos llamados recursivos

# Ordenación estable

Un ingrediente fundamental para el algoritmo que plantearemos es el siguiente

## Definición

Dada una secuencia  $A[0 \dots n-1]$ , sea  $B[0 \dots n-1]$  la secuencia resultante de ordenar  $A$  usando un algoritmo de ordenación  $S$ . Sean  $a, a'$  elementos en  $A$  tales que para el algoritmo  $A$  son equivalentes y  $a$  aparece antes que  $a'$  en  $A$ . Diremos que  $S$  es **estable** si los elementos correspondientes  $b$  y  $b'$  aparecen en el mismo orden relativo en  $B$ .

Si ordenamos por el segundo dígito, un orden estable dejaría elementos que comparten segundo dígito en el mismo orden en que nos llegaron

# RadixSort

El algoritmo RadixSort ordena por dígito **menos significativo**

- Ordena por dígito  $n_{d-1}$
- Luego, usando el mismo arreglo, ordena por dígito  $n_{d-2}$ , **con un algoritmo estable**
- Luego de ordenar  $k$  dígitos, los datos están ordenados si solo miramos el fragmento  $n_{d-k} \cdots n_{d-1}$
- Se requieren solo  $d$  pasadas para ordenar la secuencia completa

RadixSort( $A, d$ ):

**for**  $j = 0 \dots d - 1$  :

    StableSort( $A, j$ )   ▷ algoritmo de ordenación estable por  
                           $j$ -ésimo dígito menos significativo

## Ejemplo de ejecución

	Arreglo inicial	Ordenado por unidad	Ordenado por decena	Ordenado por centena
0	0 6 4	0 0 <b>0</b>	0 <b>0</b> 0	<b>0</b> 0 0
1	0 0 8	0 0 <b>1</b>	0 <b>0</b> 1	<b>0</b> 0 1
2	2 1 6	5 1 <b>2</b>	0 <b>0</b> 8	<b>0</b> 0 8
3	5 1 2	3 4 <b>3</b>	5 <b>1</b> 2	<b>0</b> 2 7
4	0 2 7	0 6 <b>4</b>	2 <b>1</b> 6	<b>0</b> 6 4
5	7 2 9	1 2 <b>5</b>	1 <b>2</b> 5	<b>1</b> 2 5
6	0 0 0	2 1 <b>6</b>	0 <b>2</b> 7	<b>2</b> 1 6
7	0 0 1	0 2 <b>7</b>	7 <b>2</b> 9	<b>3</b> 4 3
8	3 4 3	0 0 <b>8</b>	3 <b>4</b> 3	<b>5</b> 1 2
9	1 2 5	7 2 <b>9</b>	0 <b>6</b> 4	<b>7</b> 2 9

# RadixSort

RadixSort( $A, d$ ):

**for**  $j = 0 \dots d - 1$  :

    StableSort( $A, j$ )   ▷ algoritmo de ordenación estable por  
                           $j$ -ésimo dígito menos significativo

Supongamos que  $A$  tiene  $n$  datos naturales con  $d$  dígitos y se usa algoritmo estable lineal

- Si cada dígito puede tomar  $k$  valores distintos...
- Entonces RadixSort toma tiempo  $\Theta(d \cdot (n + k))$
- Si  $d$  es constante y  $k \in \mathcal{O}(n)$ , entonces RadixSort es  $\Theta(n)$

# Dos implementaciones

Estas ideas tiene dos implementaciones

LSD string sort (*Least Significant Digit*)

- Si todos los strings son del mismo largo (patentes, IP's, teléfonos)
- Funciona bien si el largo es pequeño
- No recursivo

MSD string sort (*Most Significant Digit*)

- Si los strings tienen largo diferente
- Ordenamos con `CountingSort()` por primer caracter
- Recursivamente ordenamos subarreglos correspondientes a cada caracter (excluyendo el primero, que es común en cada subarreglo)
- Como Quicksort, puede ordenar de forma independiente
- **Pero** particiona en tantos grupos como valores del primer caracter

# MSD en acción

she  
sells  
seashells  
by  
the  
sea  
shore  
the  
shells  
she  
sells  
are  
surely  
seashells

are  
by  
she  
sells  
seashells  
sea  
shore  
shells  
she  
sells  
surely  
seashells  
the  
the

are  
by  
sells  
seashells  
sea  
sells  
seashells  
she  
shore  
shells  
she  
surely  
the  
the

are  
by  
seashells  
sea  
seashells  
sells  
sells  
she  
shore  
shells  
she  
surely  
the  
the

...

are  
by  
sea  
seashells  
seashells  
sells  
sells  
she  
she  
shells  
shore  
surely  
the  
the

# Cuidados de MSD

En la ejecución de MSD string sort se debe considerar

- Si un string  $s_1$  es prefijo de otro  $s_2$ ,  $s_1$  es menor que  $s_2$   
 $she \leq shells$
- Pueden usarse diferentes alfabetos
  - binario (2)
  - minúsculas (26)
  - minúsculas + mayúsculas + dígitos (64)
  - ASCII (128)
  - Unicode (65.536)
- Para subarreglos pequeños (e.g.  $|A| \leq 10$ )
  - cambiar a InsertionSort que *sepa* que los primeros  $k$  caracteres son iguales



# Sumario

Introducción

Ordenación lineal

Cierre

# Objetivos de la clase

- ☐ Conocer algoritmos de ordenación en tiempo lineal
- ☐ Comprender la limitación de dominio para tener tales algoritmos