Ordenación lineal

Clase 7

IIC 2133 - Sección 1

Prof. Yadran Eterovic

Sumario

Construcción de un heap

Construcción de un heap

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

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

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

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

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 $i = \lfloor n/2 \rfloor - 1 \dots 0$ **:** $\triangleright$ loop decreciente
    `SiftDown`(A, i)

Construcción de un heap

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

`BuildHeap`(A)**:**
  for $i = \lfloor n/2 \rfloor - 1 \dots 0$ **:** $\triangleright$ 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

Construcción de un heap

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

`BuildHeap`(A)**:**
  for $i = \lfloor n/2 \rfloor - 1 \ldots 0$ **:** $\triangleright$ 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

Construcción de un heap

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

`BuildHeap`(A)**:**
  for $i = \lfloor n/2 \rfloor - 1 \ldots 0$ **:** $\triangleright$ 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 $\mathcal{O}(n \log(n))$

Construcción de un heap

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

`BuildHeap`(A)**:**
  for $i = \lfloor n/2 \rfloor - 1 \ldots 0$ **:** $\triangleright$ 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 $\mathcal{O}(n)$

Construcción de un heap

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

`BuildHeap`(A)**:**
  for $i = \lfloor n/2 \rfloor - 1 \ldots 0$ **:** ▷ 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

Heaps para ordenar

Ya sabemos crear un heap a partir de un arreglo cualquiera

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

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

Ordenando con heaps

Dado un heap H

- Su raíz es estrictamente mayor a todos los otros nodos

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

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**

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

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**

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`

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 $A[0 \ldots n-1]$

```
HeapSort(A):
    BuildHeap(A)
    for i = n − 1 ... 1 :        ▷ loop decreciente
        A[0] ⇋ A[i]
        A.heap_size = A.heap_size − 1
        ShiftDown(A, 0)
```

Respecto a su complejidad

Ordenando con heaps

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

$\texttt{HeapSort}(A)$**:**

  $\texttt{BuildHeap}(A)$

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

    $A[0] \leftrightharpoons A[i]$

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

    $\texttt{ShiftDown}(A, 0)$

Respecto a su complejidad

- $\texttt{BuildHeap}$ $\mathcal{O}(n)$

Ordenando con heaps

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

```
HeapSort(A):
    BuildHeap(A)
    for i = n − 1 … 1 :        ▷ loop decreciente
        A[0] ⇋ A[i]
        A.heap_size = A.heap_size − 1
        ShiftDown(A, 0)
```

Respecto a su complejidad

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

Ordenando con heaps

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

$\texttt{HeapSort}(A)$**:**

$\quad\texttt{BuildHeap}(A)$

$\quad$**for** $i = n-1 \dots 1$ **:** $\qquad \triangleright$ loop decreciente

$\qquad A[0] \leftrightharpoons A[i]$

$\qquad A.\texttt{heap\_size} = A.\texttt{heap\_size} - 1$

$\qquad \texttt{ShiftDown}(A, 0)$

Respecto a su complejidad

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

Ordenando con heads

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

```
HeapSort(A):
    BuildHeap(A)
    for i = n − 1 ... 1 :          ▷ loop decreciente
        A[0] ⇋ A[i]
        A.heap_size = A.heap_size − 1
        ShiftDown(A, 0)
```

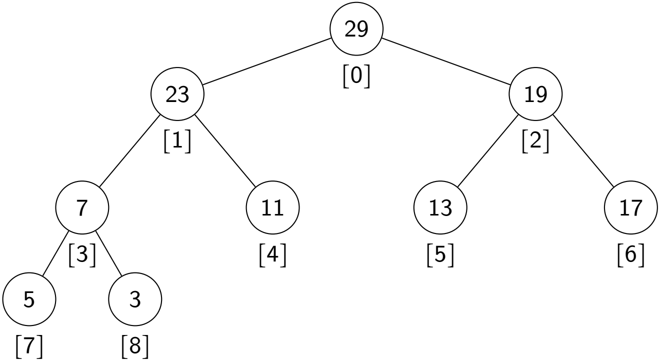
Respecto a su complejidad

- 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

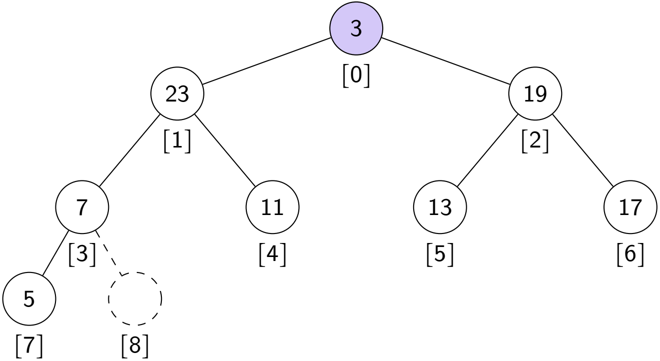
Supongamos que ya contamos con el heap resultante de `BuildHeap`



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

Heapsort en acción

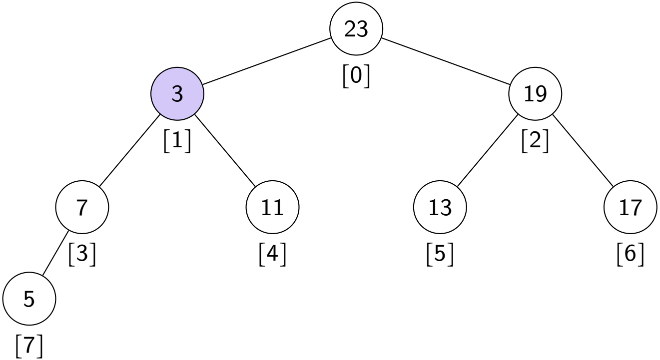
Movemos el primer elemento y reducimos el tamaño del heap en 1



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

Heapsort en acción

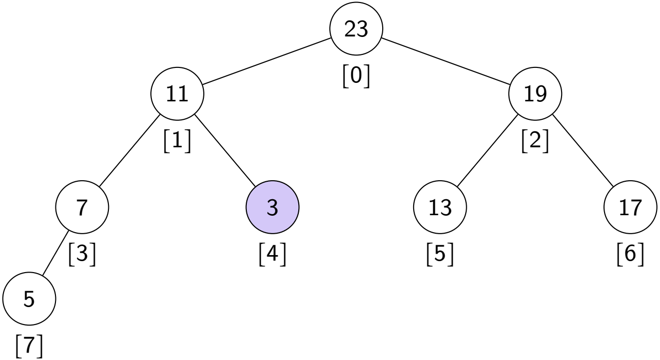
Aplicamos `SiftDown`$(A, 0)$ (el heap es $A[0 \ldots 7]$)



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

Heapsort en acción

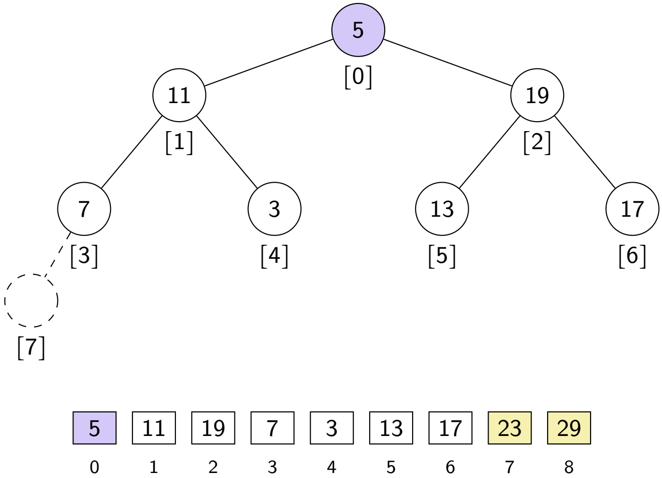
Aplicamos $\texttt{SiftDown}(A, 1)$ (el heap es $A[0 \dots 7]$)



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

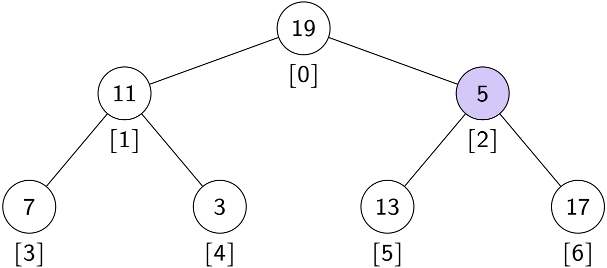
Heapsort en acción

Repetimos el proceso con la nueva raíz



Heapsort en acción

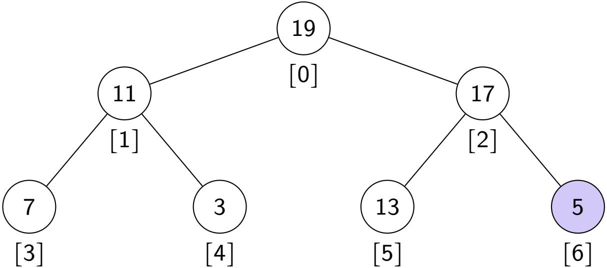
Aplicamos $\texttt{SiftDown}(A, 0)$ (el heap es $A[0 \ldots 6]$)



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

Heapsort en acción

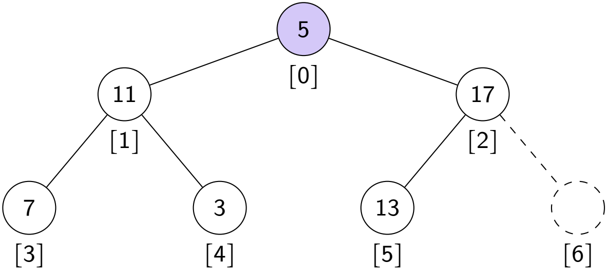
Aplicamos `SiftDown`$(A, 2)$ (el heap es $A[0\ldots 6]$)



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

Heapsort en acción

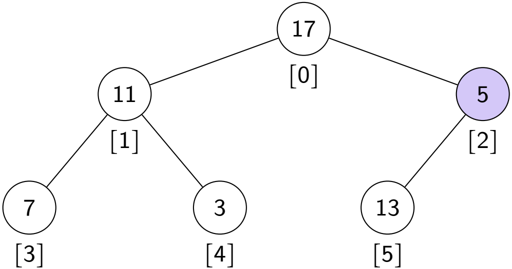
Repetimos el proceso con la nueva raíz



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

Heapsort en acción

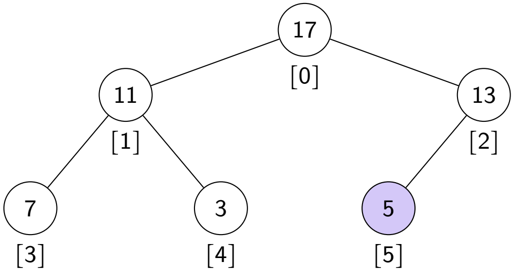
Aplicamos `SiftDown`$(A, 0)$ (el heap es $A[0 \ldots 5]$)



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

Heapsort en acción

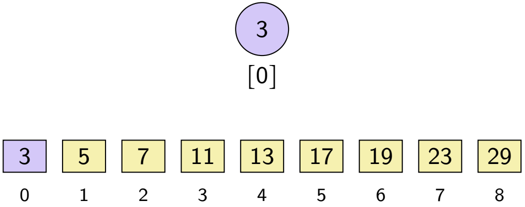
Aplicamos `SiftDown`$(A, 2)$ (el heap es $A[0 \ldots 5]$)



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

Objetivos de la clase

- ☐ Conocer algoritmos de ordenación en tiempo lineal

Objetivos de la clase

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

Sumario

Ordenación en tiempo lineal

Ordenación en tiempo lineal

Consideremos una A secuencia de n naturales entre 0 y k

Ordenación en tiempo lineal

Consideremos una A secuencia de n naturales entre 0 y k

- Para todo $a_i \in A$, se tiene que $0 \leq a_i \leq k$

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 $n = k - 1$

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

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**

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

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

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)**:**

$B[0 \dots n-1] \leftarrow$ arreglo vacío de n celdas

$C[0 \dots k] \leftarrow$ arreglo vacío de $k+1$ celdas

for $i = 0 \dots k$ **:**

  $C[i] \leftarrow 0$

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

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

for $p = 1 \dots k$ **:**

  $C[p] \leftarrow C[p] + C[p-1]$

for $r = n-1 \dots 0$ **:**

  $B[C[A[r]] - 1] \leftarrow A[r]$

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

return B

El algoritmo `CountingSort()`

`CountingSort` (A, k):

$\quad B[0 \ldots n-1] \leftarrow$ arreglo vacío

$\quad C[0 \ldots k] \leftarrow$ arreglo vacío

$\quad$ **for** $i = 0 \ldots k$ **:**

$\quad\quad C[i] \leftarrow 0$

$\quad$ **for** $j = 0 \ldots n-1$ **:**

$\quad\quad C[A[j]] \leftarrow C[A[j]] + 1$

$\quad$ **for** $p = 1 \ldots k$ **:**

$\quad\quad C[p] \leftarrow C[p] + C[p-1]$

$\quad$ **for** $r = n-1 \ldots 0$ **:**

$\quad\quad B[C[A[r]] - 1] \leftarrow A[r]$

$\quad\quad C[A[r]] \leftarrow C[A[r]] - 1$

$\quad$ **return** B

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

El algoritmo `CountingSort()`

`CountingSort` (A, k)**:**

$B[0 \dots n-1] \leftarrow$ arreglo vacío
$C[0 \dots k] \leftarrow$ arreglo vacío
for $i = 0 \dots k$ **:**
    $C[i] \leftarrow 0$
for $j = 0 \dots n-1$ **:**
    $C[A[j]] \leftarrow C[A[j]] + 1$
for $p = 1 \dots k$ **:**
    $C[p] \leftarrow C[p] + C[p-1]$
for $r = n-1 \dots 0$ **:**
    $B[C[A[r]] - 1] \leftarrow A[r]$
    $C[A[r]] \leftarrow C[A[r]] - 1$
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):

$B[0 \dots n-1] \leftarrow$ arreglo vacío
$C[0 \dots k] \leftarrow$ arreglo vacío
for $i = 0 \dots k$ **:**
    $C[i] \leftarrow 0$
for $j = 0 \dots n-1$ **:**
    $C[A[j]] \leftarrow C[A[j]] + 1$
for $p = 1 \dots k$ **:**
    $C[p] \leftarrow C[p] + C[p-1]$
for $r = n-1 \dots 0$ **:**
    $B[C[A[r]] - 1] \leftarrow A[r]$
    $C[A[r]] \leftarrow C[A[r]] - 1$
return B

A

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

Ejemplo de ejecución

```
CountingSort (A, k):
    B[0 ... n − 1] ← arreglo vacío
    C[0 ... k] ← arreglo vacío
    for i = 0 ... k :
        C[i] ← 0
    for j = 0 ... n − 1 :
        C[A[j]] ← C[A[j]] + 1
    for p = 1 ... k :
        C[p] ← C[p] + C[p − 1]
    for r = n − 1 ... 0 :
        B[C[A[r]] − 1] ← A[r]
        C[A[r]] ← C[A[r]] − 1
    return B
```

A

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

Hacemos el llamado `CountingSort`$(A, 7)$

Ejemplo de ejecución

```
CountingSort (A, k):
    B[0 ... n − 1] ← arreglo vacío
    C[0 ... k] ← arreglo vacío
    for i = 0 ... k :
        C[i] ← 0
```

A

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

C

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

Ejemplo de ejecución

```
CountingSort (A, k):
    B[0 ... n − 1] ← arreglo vacío
    C[0 ... k] ← arreglo vacío
    for i = 0 ... k :
        C[i] ← 0
    for j = 0 ... n − 1 :
        C[A[j]] ← C[A[j]] + 1
```

A

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

C

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

C

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

Ejemplo de ejecución

```
CountingSort (A, k):
    B[0 ... n − 1] ← arreglo vacío
    C[0 ... k] ← arreglo vacío
    for i = 0 ... k :
        C[i] ← 0
    for j = 0 ... n − 1 :
        C[A[j]] ← 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):

    $B[0 \ldots n-1] \leftarrow$ arreglo vacío
    $C[0 \ldots k] \leftarrow$ arreglo vacío
    for $i = 0 \ldots k$ **:**
        $C[i] \leftarrow 0$
    for $j = 0 \ldots n-1$ **:**
        $C[A[j]] \leftarrow C[A[j]] + 1$
    for $p = 1 \ldots k$ **:**
        $C[p] \leftarrow C[p] + C[p-1]$

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

Ejemplo de ejecución

`CountingSort` (A, k):

$B[0 \dots n-1] \leftarrow$ arreglo vacío
$C[0 \dots k] \leftarrow$ arreglo vacío
for $i = 0 \dots k$ **:**
  $C[i] \leftarrow 0$
for $j = 0 \dots n-1$ **:**
  $C[A[j]] \leftarrow C[A[j]] + 1$
for $p = 1 \dots k$ **:**
  $C[p] \leftarrow C[p] + C[p-1]$

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

for $r = n-1 \ldots 0$ **:**
  $B[C[A[r]]-1] \leftarrow A[r]$
  $C[A[r]] \leftarrow C[A[r]] - 1$

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

A

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

B

0	1	2	3	4	5	6	7	8	9	
				3						$r = 9$
				3					**7**	$r = 8$
				3		**5**			7	$r = 7$
				3	**5**	5			7	$r = 6$
				3	5	5		**7**	7	$r = 5$
0				3	5	5		7	7	$r = 4$
0			**3**	3	5	5		7	7	$r = 3$
0		**1**	3	3	5	5		7	7	$r = 2$
0	**1**	1	3	3	5	5		7	7	$r = 1$
0	1	1	3	3	5	5	**7**	7	7	$r = 0$

C

	0	1	2	3	4	5	6	7
$r = 9$	1	3	3	**5**	5	7	7	10
$r = 8$	1	3	3	4	5	7	7	**10**
$r = 7$	1	3	3	4	5	**7**	7	9
$r = 6$	1	3	3	4	5	**6**	7	9
$r = 5$	1	3	3	4	5	5	7	**9**
$r = 4$	**1**	3	3	4	5	5	7	8
$r = 3$	0	3	3	**4**	5	5	7	8
$r = 2$	0	**3**	3	3	5	5	7	8
$r = 1$	0	**2**	3	3	5	5	7	8
$r = 0$	0	1	3	3	5	5	7	**8**

RadixSort

Otro algoritmo de ordenación en tiempo lineal es `RadixSort`

RadixSort

Otro algoritmo de ordenación en tiempo lineal es `RadixSort`

- Usado por las máquinas que ordenaban tarjetas perforadas

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

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

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

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

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

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

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

Ordenando por dígito

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

Ordenando por dígito

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

- Podemos ordenar según el dígito más significativo n_0

Ordenando por dígito

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

- Podemos ordenar según el dígito más significativo n_0
- Según este dígito, separamos los números en *compartimientos*

Ordenando por dígito

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

- 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

Ordenando por dígito

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

- 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

Ordenando por dígito

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

- 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

Ordenación estable

Un ingrediente fundamental para el algoritmo que plantearemos es el siguiente

Definición

Dada una secuencia $A[0 \ldots n-1]$, sea $B[0 \ldots n-1]$ la secuencia resultante de ordenar A usando un algoritmo de ordenación $\mathcal{S}$.

Ordenación estable

Un ingrediente fundamental para el algoritmo que plantearemos es el siguiente

Definición

Dada una secuencia $A[0 \ldots n-1]$, sea $B[0 \ldots n-1]$ la secuencia resultante de ordenar A usando un algoritmo de ordenación $\mathcal{S}$. Sean a, a' elementos en A tales que para el algoritmo $\mathcal{A}$ son equivalentes y a aparece antes que a' en A.

Ordenación estable

Un ingrediente fundamental para el algoritmo que plantearemos es el siguiente

Definición

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

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 $\mathcal{S}$. Sean a, a' elementos en A tales que para el algoritmo $\mathcal{A}$ son equivalentes y a aparece antes que a' en A. Diremos que $\mathcal{S}$ 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**

RadixSort

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

- Ordena por dígito n_{d-1}

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**

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 $n_{d-k} \cdots n_{d-1}$

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

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

RadixSort

```
RadixSort(A, d):
    for j = 0 ... d − 1 :
        StableSort(A, j)    ▷ algoritmo de ordenación estable por
                              j-ésimo dígito menos significativo
```

RadixSort

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

RadixSort

$\texttt{RadixSort}(A, d)$:
  for $j = 0 \ldots d - 1$ **:**
    $\texttt{StableSort}(A, j)$ $\triangleright$ 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...

RadixSort

```
RadixSort(A, d):
    for j = 0 ... 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 $\Theta(d \cdot (n + k))$

RadixSort

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

Dos implementaciones

Estas ideas tiene dos implementaciones

LSD string sort (*Least Significant Digit*)

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)

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

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

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*)

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

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

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)

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

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

Cuidados de MSD

En la ejecución de MSD string sort se debe considerar

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

$$\text{she} \preceq \text{shells}$$

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

$$\text{she} \leq \text{shells}$$

- Pueden usarse diferentes alfabetos

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

$$\text{she} \preceq \text{shells}$$

- Pueden usarse diferentes alfabetos
    - binario (2)

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

$$\text{she} \preceq \text{shells}$$

- Pueden usarse diferentes alfabetos
  - binario (2)
  - minúsculas (26)

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

$$\text{she} \leq \text{shells}$$

- Pueden usarse diferentes alfabetos
  - binario (2)
  - minúsculas (26)
  - minúsculas + mayúsculas + dígitos (64)

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

$$\text{she} \preceq \text{shells}$$

- Pueden usarse diferentes alfabetos
  - binario (2)
  - minúsculas (26)
  - minúsculas + mayúsculas + dígitos (64)
  - ASCII (128)

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

$$\text{she} \preceq \text{shells}$$

- Pueden usarse diferentes alfabetos
  - binario (2)
  - minúsculas (26)
  - minúsculas + mayúsculas + dígitos (64)
  - ASCII (128)
  - Unicode (65.536)

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

$$\text{she} \leq \text{shells}$$

- Pueden usarse diferentes alfabetos
  - binario (2)
  - minúsculas (26)
  - minúsculas + mayúsculas + dígitos (64)
  - ASCII (128)
  - Unicode (65.536)
- Para subarreglos pequeños (e.g. $|A| \leq 10$)

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

$$\text{she} \leq \text{shells}$$

- 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

Objetivos de la clase

Objetivos de la clase

- ☐ Conocer algoritmos de ordenación en tiempo lineal

Objetivos de la clase

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