

IIC-2133 — Estructuras de Datos y Algoritmos

Algoritmos de Ordenación (y Heaps)

Jorge A. Baier

Departamento de Ciencia de la Computación
Pontificia Universidad Católica de Chile

Santiago, Chile



Insertion Sort

Tarea: Ordenar un arreglo $A[0..n - 1]$

Idea Principal: A la izquierda del índice j , todo está ordenado. j . En cada iteración, “movemos” el valor en $A[j]$ hasta una posición tal que $A[0..j]$ quede ordenado.



Insertion Sort

Tarea: Ordenar un arreglo $A[0..n - 1]$

Idea Principal: A la izquierda del índice j , todo está ordenado. j . En cada iteración, “movemos” el valor en $A[j]$ hasta una posición tal que $A[0..j]$ quede ordenado.

```
1 procedure Insertion-Sort( $A$ )
2   for  $j \leftarrow 1$  to  $n - 1$  do
3      $key \leftarrow A[j]$ 
4      $i \leftarrow j - 1$ 
5     while  $i \geq 0$  and  $A[i] > key$  do
6        $A[i + 1] \leftrightarrow A[i]$ 
7        $i \leftarrow i - 1$ 
```

Tiempo en el peor caso:



Insertion Sort

Tarea: Ordenar un arreglo $A[0..n - 1]$

Idea Principal: A la izquierda del índice j , todo está ordenado. j . En cada iteración, “movemos” el valor en $A[j]$ hasta una posición tal que $A[0..j]$ quede ordenado.

```
1 procedure Insertion-Sort( $A$ )
2   for  $j \leftarrow 1$  to  $n - 1$  do
3      $key \leftarrow A[j]$ 
4      $i \leftarrow j - 1$ 
5     while  $i \geq 0$  and  $A[i] > key$  do
6        $A[i + 1] \leftrightarrow A[i]$ 
7        $i \leftarrow i - 1$ 
```

Tiempo en el peor caso: $O(n^2)$



Merge Sort

Tarea: Ordenar un arreglo $A[0..n - 1]$

Idea Principal: Ordenamos $A[1..m - 1]$ y luego $A[m..n - 1]$, donde $m = \lfloor n/2 \rfloor$, y luego hacemos la “mezcla ordenada” de los subarreglos.



Merge Sort

Tarea: Ordenar un arreglo $A[0..n - 1]$

Idea Principal: Ordenamos $A[1..m - 1]$ y luego $A[m..n - 1]$, donde $m = \lfloor n/2 \rfloor$, y luego hacemos la “mezcla ordenada” de los subarreglos.

```
1 procedure Merge-Sort( $A, p, r$ )
2   if  $p < r$  then
3      $q \leftarrow \lfloor (p + r)/2 \rfloor$ 
4     Merge-Sort( $A, p, q$ )
5     Merge-Sort( $A, q + 1, r$ )
6     Merge( $A, p, q, r$ )
```

Tiempo en el peor caso:



Merge Sort

Tarea: Ordenar un arreglo $A[0..n - 1]$

Idea Principal: Ordenamos $A[1..m - 1]$ y luego $A[m..n - 1]$, donde $m = \lfloor n/2 \rfloor$, y luego hacemos la “mezcla ordenada” de los subarreglos.

```
1 procedure Merge-Sort( $A, p, r$ )
2   if  $p < r$  then
3      $q \leftarrow \lfloor (p + r)/2 \rfloor$ 
4     Merge-Sort( $A, p, q$ )
5     Merge-Sort( $A, q + 1, r$ )
6     Merge( $A, p, q, r$ )
```

Tiempo en el peor caso: $O(n \log n)$

Memoria:



Merge Sort

Tarea: Ordenar un arreglo $A[0..n - 1]$

Idea Principal: Ordenamos $A[1..m - 1]$ y luego $A[m..n - 1]$, donde $m = \lfloor n/2 \rfloor$, y luego hacemos la “mezcla ordenada” de los subarreglos.

```
1 procedure Merge-Sort( $A, p, r$ )
2   if  $p < r$  then
3      $q \leftarrow \lfloor (p + r)/2 \rfloor$ 
4     Merge-Sort( $A, p, q$ )
5     Merge-Sort( $A, q + 1, r$ )
6     Merge( $A, p, q, r$ )
```

Tiempo en el peor caso: $O(n \log n)$

Memoria: Necesitamos un arreglo adicional del tamaño de A .



Merge de MergeSort

```
1 procedure Merge( $A, p, q, r$ )
2    $L \leftarrow$  nuevo arreglo de tamaño  $q - p + 2$ 
3    $R \leftarrow$  nuevo arreglo de tamaño  $r - q + 1$ 
4    $L[0..q - p] \leftarrow A[p..q]$ 
5    $R[0..r - q - 1] \leftarrow A[q + 1..r]$ 
6    $L[q - p + 1] \leftarrow R[r - q] \leftarrow \infty$ 
7    $i \leftarrow j \leftarrow 0$ 
8   for  $k \leftarrow p$  to  $r$  do
9     if  $L[i] \leq R[j]$  then
10       $A[k] \leftarrow L[i]$ 
11       $i \leftarrow i + 1$ 
12     else
13       $A[k] \leftarrow R[j]$ 
14       $j \leftarrow j + 1$ 
```



Heap Sort

Nuestro primer algoritmo $O(n)$ *in-place*.

Usa una cola de prioridades (*Heap*) como estructura de datos principal.



- Un *min/max heap binario* es un árbol binario que cumple la siguiente propiedad

Propiedad de Min-Heap: La clave del elemento almacenado en un nodo es *mayor o igual* a la clave de sus hijos.

Propiedad de Max-Heap: La clave del elemento almacenado en un nodo es *menor o igual* a la clave de sus hijos.

- El heap es también un árbol balanceado:
 - 1 si en un nivel un nodo n no tiene hijos, todos los nodos que están a la derecha de n en el mismo nivel tampoco los tienen
 - 2 un nodo no puede tener un hijo derecho si no tiene un hijo izquierdo



Implementación de Min Heaps

- Los elementos de un heap de n elementos se ubican en las posiciones $A[1..n]$ del arreglo ($A[0]$ no se usa).
- Dado un nodo i definimos:



Implementación de Min Heaps

- Los elementos de un heap de n elementos se ubican en las posiciones $A[1..n]$ del arreglo ($A[0]$ no se usa).
- Dado un nodo i definimos:

$$Parent(i) = \lfloor i/2 \rfloor$$

$$Left(i) = 2i$$

$$Right(i) = 2i + 1$$

- Además si el arreglo A implementa a un heap, $heap-size[A]$ es el número de elementos en el heap.
- Observación: todo arreglo ordenado ascendentemente es un *min-heap*.



Subrutina *Decrease-Key*

Objetivo: Disminuir la prioridad a un elemento en el heap en A

Supuesto: A era un heap antes de cambiar la prioridad

Idea: Percolamos el elemento hacia arriba

```
1 procedure Decrease-Key( $A, i, key$ )  
2   if  $A[i] < key$  then error “solo para disminuir clave”  
3    $A[i] \leftarrow key$   
4   while  $i > 1$  and  $A[Parent(i)] > A[i]$  do  
5      $A[Parent(i)] \leftrightarrow A[i]$   
6      $i \leftarrow Parent(i)$ 
```

Propiedad: *Decrease-Key* mantiene la propiedad de Heap

Tiempo peor caso:



Subrutina *Decrease-Key*

Objetivo: Disminuir la prioridad a un elemento en el heap en A

Supuesto: A era un heap antes de cambiar la prioridad

Idea: Percolamos el elemento hacia arriba

```
1 procedure Decrease-Key( $A, i, key$ )  
2   if  $A[i] < key$  then error “solo para disminuir clave”  
3    $A[i] \leftarrow key$   
4   while  $i > 1$  and  $A[Parent(i)] > A[i]$  do  
5      $A[Parent(i)] \leftrightarrow A[i]$   
6      $i \leftarrow Parent(i)$ 
```

Propiedad: *Decrease-Key* mantiene la propiedad de Heap

Tiempo peor caso: $O(\log n)$ (donde n es el tamaño del heap)



Inserción de un elemento

Idea: Ubicamos el elemento al final del arreglo y luego lo “percolamos” hacia arriba mientras sea necesario.

```
1 procedure Min-Heap-Insert( $A, key$ )
2    $heap-size[A] \leftarrow heap-size[A] + 1$ 
3    $A[heap-size[A]] = \infty$ 
4    $Decrease-Key(A, heap-size[A], key)$ 
```

Propiedad: *Min-Heap-Insert* mantiene la propiedad de Heap
Tiempo peor caso:



Inserción de un elemento

Idea: Ubicamos el elemento al final del arreglo y luego lo “percolamos” hacia arriba mientras sea necesario.

```
1 procedure Min-Heap-Insert( $A, key$ )
2    $heap-size[A] \leftarrow heap-size[A] + 1$ 
3    $A[heap-size[A]] = \infty$ 
4   Decrease-Key( $A, heap-size[A], key$ )
```

Propiedad: *Min-Heap-Insert* mantiene la propiedad de Heap

Tiempo peor caso: $O(\log n)$ (donde n es el tamaño del heap)



Subrutina Min-Heapify

Objetivo: Reestablecer propiedad de Heap con raíz en el índice i

Supuesto: Subárboles con raíz en $Left(i)$ y $Right(i)$ son Heaps

```
1 procedure Min-Heapify( $A, i$ )
2    $l \leftarrow Left(i)$ 
3    $r \leftarrow Right(i)$ 
4    $min \leftarrow i$ 
5   if  $l \leq heap-size[A]$  and  $A[l] < A[min]$  then  $min \leftarrow l$ 
6   if  $r \leq heap-size[A]$  and  $A[r] < A[min]$  then  $min \leftarrow r$ 
7   if  $min \neq i$  then
8      $A[min] \leftrightarrow A[i]$ 
9     Min-Heapify( $A, min$ )
```

Propiedad: Al terminar el árbol con raíz en i es un heap

Tiempo peor caso:



Subrutina Min-Heapify

Objetivo: Reestablecer propiedad de Heap con raíz en el índice i

Supuesto: Subárboles con raíz en $Left(i)$ y $Right(i)$ son Heaps

```
1 procedure Min-Heapify( $A, i$ )
2    $l \leftarrow Left(i)$ 
3    $r \leftarrow Right(i)$ 
4    $min \leftarrow i$ 
5   if  $l \leq heap-size[A]$  and  $A[l] < A[min]$  then  $min \leftarrow l$ 
6   if  $r \leq heap-size[A]$  and  $A[r] < A[min]$  then  $min \leftarrow r$ 
7   if  $min \neq i$  then
8      $A[min] \leftrightarrow A[i]$ 
9     Min-Heapify( $A, min$ )
```

Propiedad: Al terminar el árbol con raíz en i es un heap

Tiempo peor caso: $O(\log(n/i))$



Extracción del elemento de mejor (menor) prioridad

Idea: Ubicamos el último elemento ($A[\text{heap-size}[A]]$) en la raíz y luego reparamos la propiedad de heap hacia abajo.

```
1 function Extract-Min( $A$ )
2   if  $\text{heap-size}[A] < 1$  then error “heap vacío”
3    $\text{minkey} \leftarrow A[1]$ 
4    $A[1] \leftarrow A[\text{heap-size}[A]]$ 
5    $\text{heap-size}[A] \leftarrow \text{heap-size}[A] - 1$ 
6   Min-Heapify( $A, 1$ )
7   return  $\text{minkey}$ 
```



Objetivo: Obtener la prioridad del elemento de mejor prioridad

```
1 function Heap-Minimum(A)  
2   return A[0]
```

Tiempo: $O(1)$



Build Heap

Objetivo: Dado un arreglo de números, A de n elementos, convertirlo en un heap

```
1 procedure Build-Min-Heap( $A, n$ )  
2    $heap-size[A] \leftarrow n$   
3    $i \leftarrow Parent(heap-size[A])$   
4   while  $i \geq 1$  do  
5      $Min-Heapify(A, i)$   
6      $i \leftarrow i - 1$ 
```

Propiedad: A contendrá un heap al terminar la ejecución

Tiempo:



Build Heap

Objetivo: Dado un arreglo de números, A de n elementos, convertirlo en un heap

```
1 procedure Build-Min-Heap( $A, n$ )  
2    $heap-size[A] \leftarrow n$   
3    $i \leftarrow Parent(heap-size[A])$   
4   while  $i \geq 1$  do  
5      $Min-Heapify(A, i)$   
6      $i \leftarrow i - 1$ 
```

Propiedad: A contendrá un heap al terminar la ejecución

Tiempo: $O(n)$ en el peor caso [demostración: pizarra]



Heap Sort

Objetivo: Dado un arreglo de números, A , de n elementos ordenar ascendentemente los elementos de A

Observación: Usamos un MAX-Heap

```
1 procedure HeapSort( $A, n$ )
2   Build-Max-Heap( $A, n$ )
3   for  $i \leftarrow n$  downto 2 do
4      $A[i] \leftrightarrow A[1]$ 
5      $heap-size[A] \leftarrow heap-size[A] - 1$ 
6     Max-Heapify( $A, 1$ )
```

Tiempo peor caso: $O(n \log n)$



Objetivo: Ordenar $A[p \dots r]$ in-place

- 1 Si $p \geq r$, retornamos (el arreglo está ordenado)
- 2 Reordenar elementos en $A[p \dots r]$ tales que, para algún q

$$\begin{aligned} A[i] &\leq A[q] && \text{si } i \in \{p, \dots, q-1\} \\ A[q] &\leq A[j] && \text{si } j \in \{q+1, \dots, r\} \end{aligned}$$

- 3 Ordenar $A[p \dots q-1]$ (usando QuickSort)
- 4 Ordenar $A[q+1 \dots r]$ (usando QuickSort)

Observación: paso 2 debe ser $O(n)$



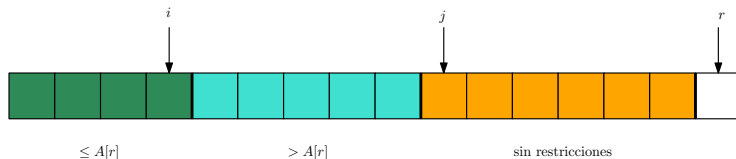
Pseudocódigo para Quick Sort

```
1 procedure Quick-Sort( $A, p, r$ )  
2   if  $p < r$  then  
3      $q \leftarrow \text{Partition}(A, p, r)$   
4     Quick-Sort( $A, p, q - 1$ )  
5     Quick-Sort( $A, q + 1, r$ )
```



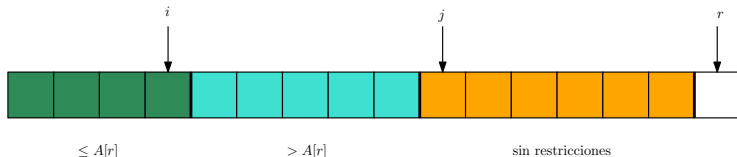
Partition

- Usamos las variables i , j
- $A[r]$ será el pivote (es decir, el último elemento)
- Perseguimos construir un loop que satisfaga la siguiente invariante:
 - 1 $A[k] \leq A[r]$ para todo $k \in \{p, \dots, i\}$
 - 2 $A[r] < A[k]$ para todo $k \in \{i + 1, \dots, j - 1\}$



Pseudo-Código para Partition

```
1 function Partition(A, p, r)  
2    $i \leftarrow p - 1$   
3    $j \leftarrow p$   
4   while  $j \leq r$  do  
5     if  $A[j] \leq A[r]$  then  
6        $i \leftarrow i + 1$   
7        $A[i] \leftrightarrow A[j]$   
8      $j \leftarrow j + 1$   
9   return  $i$ 
```



Tiempo en el Peor Caso

Teorema: Tiempo de ejecución de Quick Sort es $\Theta(n^2)$ en el peor caso.

Si, en vez de *Partition*, usamos:

```
1 function Random-Partition( $A, p, r$ )  
2    $i \leftarrow$  número aleatorio en  $\{p, \dots, r\}$   
3    $A[r] \leftrightarrow A[i]$   
4   return Partition( $A, p, r$ )
```

Teorema: El tiempo de ejecución de Quick Sort, usando *Random-Partition*, es $O(n \log n)$ **en el caso promedio**.

Demostración: se justifica y resuelve $\frac{T(n)}{n+1} = \frac{O(1)}{n+1} + \frac{T(n-1)}{n}$.



“Optimalidad” de Quick Sort, Merge Sort y Heap Sort

Teorema: Sea **Sort** un algoritmo de ordenación que sólo puede comparar e intercambiar elementos de un arreglo. Entonces, el tiempo de ejecución de **Sort** es $\Omega(n \log n)$.

Demostración: pizarra.

Este se conoce como un resultado de *optimalidad asintótica*.

Algoritmos mejores que **Sort**—que “leen” claves—existen!



Counting Sort

- Nuestro primer algoritmo de ordenación $O(n)$
- **Input:** Arreglo A
- **Supuesto:** Las claves a ordenar están en $\{0, \dots, k\}$
- Pregunta que inspira al algoritmo:



¿Podemos determinar la posición de cada elemento de A en el arreglo ordenado?



Counting Sort

- Nuestro primer algoritmo de ordenación $O(n)$
- **Input:** Arreglo A
- **Supuesto:** Las claves a ordenar están en $\{0, \dots, k\}$
- Pregunta que inspira al algoritmo:



¿Podemos determinar la posición de cada elemento de A en el arreglo ordenado?

- **Objetivo:** construir arreglo $C[0 \dots k]$ tal que $C[k]$ contiene el número de elementos en A que son *menores o iguales* a k .
- Usando C , podemos construir el arreglo ordenado recorriendo A una vez.



Pseudo código para Counting Sort

(suponemos que las claves están en $\{0, \dots, k\}$)

```
1 procedure Counting-Sort( $A, k$ )  
2    $B \leftarrow$  copia del arreglo  $A$   
3    $C \leftarrow$  arreglo de tamaño  $k + 1$  lleno de ceros  
4   for  $j \leftarrow 0$  to  $\text{len}[A] - 1$  do  
      $C[A[j]] \leftarrow C[A[j]] + 1$ 
```



Pseudo código para Counting Sort

(suponemos que las claves están en $\{0, \dots, k\}$)

```
1 procedure Counting-Sort( $A, k$ )
2    $B \leftarrow$  copia del arreglo  $A$ 
3    $C \leftarrow$  arreglo de tamaño  $k + 1$  lleno de ceros
4   for  $j \leftarrow 0$  to  $\text{len}[A] - 1$  do
       $C[A[j]] \leftarrow C[A[j]] + 1$ 
5   for  $i \leftarrow 1$  to  $k$  do  $C[i] \leftarrow C[i] + C[i - 1]$ 
6   for  $j \leftarrow \text{len}[B] - 1$  downto  $0$  do
7      $A[C[j]] \leftarrow B[j]$ 
8      $C[j] \leftarrow C[j] - 1$ 
```

Propiedad: Counting sort es correcto y **estable**.

Tiempo: Es $\Theta(k + n)$. En la práctica lo usamos si k es $O(n)$.



- **Idea:** Ordenar, sucesiva y **establemente**, desde el dígito menos significativo, hasta el más significativo.
- Un ejemplo de ejecución:

329

457

839

436

720

355



- **Idea:** Ordenar, sucesiva y **establemente**, desde el dígito menos significativo, hasta el más significativo.
- Un ejemplo de ejecución:

329 720

457 355

839 457

436 657

720 329

355 839



- **Idea:** Ordenar, sucesiva y **establemente**, desde el dígito menos significativo, hasta el más significativo.
- Un ejemplo de ejecución:

329	720	720
457	355	329
839	457	839
436	657	355
720	329	457
355	839	657



- **Idea:** Ordenar, sucesiva y **establemente**, desde el dígito menos significativo, hasta el más significativo.
- Un ejemplo de ejecución:

329	720	720	329
457	355	329	355
839	457	839	457
436	657	355	657
720	329	457	720
355	839	657	839



Pseudo Código de Radix Sort

Suponiendo que cada elemento en el arreglo A tiene a lo más d dígitos, donde el dígito 1 es el menos significativo, obtenemos:

```
1 procedure Radix-Sort( $A, d$ )
2   for  $i \leftarrow 1$  to  $d$  do
3     | Ordene  $A$  sobre el dígito  $i$  con un algoritmo estable
```

Propiedad:

- 1 Si cada dígito puede tomar hasta k valores, y
 - 2 el algoritmo usado en la línea 3 es $\Theta(n + k)$
- entonces *Radix-Sort* ejecuta en $\Theta(d(n + k))$.



Teorema: Dado n números de b bits y cualquier entero positivo $r \leq b$, *Radix-Sort* ejecuta en tiempo $\Theta(\frac{b}{r}(n + 2^r))$.



Otra Propiedad de Radix Sort

Teorema: Dado n números de b bits y cualquier entero positivo $r \leq b$, *Radix-Sort* ejecuta en tiempo $\Theta(\frac{b}{r}(n + 2^r))$.

Corolario: *Radix-Sort* es $\Theta(n)$ si $b \leq C \log_2 n$, donde C es una constante.

Demostración: Si $b < \lfloor \log_2 n \rfloor$, elegimos $r = b$. Si $b > \lfloor \log_2 n \rfloor$, hacemos $r = \lfloor \log_2 n \rfloor$



Teorema: Dado n números de b bits y cualquier entero positivo $r \leq b$, *Radix-Sort* ejecuta en tiempo $\Theta(\frac{b}{r}(n + 2^r))$.

Corolario: *Radix-Sort* es $\Theta(n)$ si $b \leq C \log_2 n$, donde C es una constante.

Demostración: Si $b < \lfloor \log_2 n \rfloor$, elegimos $r = b$. Si $b > \lfloor \log_2 n \rfloor$, hacemos $r = \lfloor \log_2 n \rfloor$

Precaución! el corolario anterior:

- no dice nada de la memoria adicional requerida
- no dice cuál es el valor óptimo de r

