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.

```
procedure Insertion-Sort(A)
    for j ← 1 to n − 1 do
        key ← A[j]
        i ← j − 1
        while i ≥ 0 and A[i] > key do
            A[i + 1] ↔ A[i]
            i ← 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.

procedure *Insertion-Sort*(A)
  for $j \leftarrow 1$ **to** $n-1$ **do**
    $key \leftarrow A[j]$
    $i \leftarrow j-1$
    while $i \geq 0$ *and* $A[i] > key$ **do**
      $A[i+1] \leftrightarrow A[i]$
      $i \leftarrow i-1$

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.

```
procedure Merge-Sort(A, p, r)
    if p < r then
        q ← ⌊(p + r)/2⌋
        Merge-Sort(A, p, q)
        Merge-Sort(A, q + 1, r)
        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.

procedure *Merge-Sort*(A, p, r)
    if $p < r$ **then**
        $q \leftarrow \lfloor (p+r)/2 \rfloor$
        Merge-Sort(A, p, q)
        Merge-Sort$(A, q+1, r)$
        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.

```
procedure Merge-Sort(A, p, r)
    if p < r then
        q ← ⌊(p + r)/2⌋
        Merge-Sort(A, p, q)
        Merge-Sort(A, q + 1, r)
        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

procedure *Merge*(A, p, q, r)

  $L \leftarrow$ nuevo arreglo de tamaño $q - p + 2$

  $R \leftarrow$ nuevo arreglo de tamaño $r - q + 1$

  $L[0..q - p] \leftarrow A[p..q]$

  $R[0..r - q - 1] \leftarrow A[q + 1..r]$

  $L[q - p + 1] \leftarrow R[r - q] \leftarrow \infty$

  $i \leftarrow j \leftarrow 0$

  for $k \leftarrow p$ *to* r **do**

    if $L[i] \leq R[j]$ **then**

      $A[k] \leftarrow L[i]$

      $i \leftarrow i + 1$

    else

      $A[k] \leftarrow R[j]$

      $j \leftarrow j + 1$

Heap Sort

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

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

Heaps Binarios

- 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* 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:

$$\mathit{Parent}(i) = \lfloor i/2 \rfloor$$
$$\mathit{Left}(i) = 2i$$
$$\mathit{Right}(i) = 2i + 1$$

- 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

procedure *Decrease-Key*(A, i, key)
  if $A[i] < key$ **then** error "solo para disminuir clave"
  $A[i] \leftarrow key$
  while $i > 1$ *and* $A[\textit{Parent}(i)] > A[i]$ **do**
    $A[\textit{Parent}(i)] \leftrightarrow A[i]$
    $i \leftarrow \textit{Parent}(i)$

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

procedure *Decrease-Key*(A, i, key)
    if $A[i] < key$ **then** error "solo para disminuir clave"
    $A[i] \leftarrow key$
    while $i > 1$ *and* $A[\mathit{Parent}(i)] > A[i]$ **do**
        $A[\mathit{Parent}(i)] \leftrightarrow A[i]$
        $i \leftarrow \mathit{Parent}(i)$

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.

procedure *Min-Heap-Insert*(A, key)
  heap-size$[A] \leftarrow$ *heap-size*$[A] + 1$
  $A[$*heap-size*$[A]] = \infty$
  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.

procedure *Min-Heap-Insert*(A, key)
    heap-size$[A] \leftarrow$ *heap-size*$[A] + 1$
    $A[$*heap-size*$[A]] = \infty$
    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

```
procedure Min-Heapify(A, i)
    l ← Left(i)
    r ← Right(i)
    min ← i
    if l ≤ heap-size[A] and A[l] < A[min] then min ← l
    if r ≤ heap-size[A] and A[r] < A[min] then min ← r
    if min ≠ i then
        A[min] ↔ A[i]
        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

procedure *Min-Heapify*(A, i)
    $l \leftarrow$ *Left*(i)
    $r \leftarrow$ *Right*(i)
    $min \leftarrow i$
    if $l \leq$ *heap-size*$[A]$ *and* $A[l] < A[min]$ **then** $min \leftarrow l$
    if $r \leq$ *heap-size*$[A]$ *and* $A[r] < A[min]$ **then** $min \leftarrow r$
    if $min \neq i$ **then**
        $A[min] \leftrightarrow A[i]$
        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[*heap-size*[A]]) en la raíz y luego reparamos la propiedad de heap hacia abajo.

function *Extract-Min*(A)
  if *heap-size*$[A] < 1$ **then** error "heap vacío"
  $minkey \leftarrow A[1]$
  $A[1] \leftarrow A[$*heap-size*$[A]]$
  heap-size$[A] \leftarrow$ *heap-size*$[A] - 1$
  Min-Heapify$(A, 1)$
  return $minkey$

Heap-Min

Objetivo: Obtener la prioridad del elemento de mejor prioridad

function *Heap-Minimum*(A)
  return $A[0]$

Tiempo: $O(1)$

Build Heap

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

```
procedure Build-Min-Heap(A, n)
    heap-size[A] ← n
    i ← Parent(heap-size[A])
    while i ≥ 1 do
        Min-Heapify(A, i)
        i ← 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

procedure *Build-Min-Heap*(A, n)
    heap-size$[A] \leftarrow n$
    $i \leftarrow Parent($*heap-size*$[A])$
    while $i \geq 1$ **do**
        Min-Heapify(A, i)
        $i \leftarrow i - 1$

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 ascendetemente los elementos de A

Observación: Usamos un MAX-Heap

```
procedure HeapSort(A,n)
    Build-Max-Heap(A, n)
    for i ← n downto 2 do
        A[i] ↔ A[1]
        heap-size[A] ← heap-size[A] − 1
        Max-Heapify(A, 1)
```

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

Quick Sort

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

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

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

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

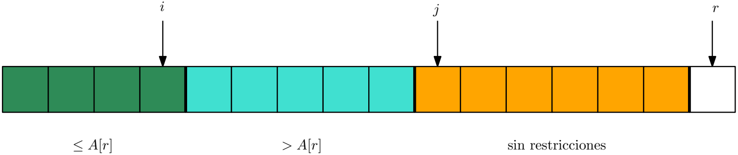
Observación: paso 2 debe ser $O(n)$

Pseudocódigo para Quick Sort

procedure *Quick-Sort*(A, p, r)
  if $p < r$ **then**
    $q \leftarrow$ *Partition*(A, p, r)
    Quick-Sort$(A, p, q-1)$
    Quick-Sort$(A, q+1, r)$

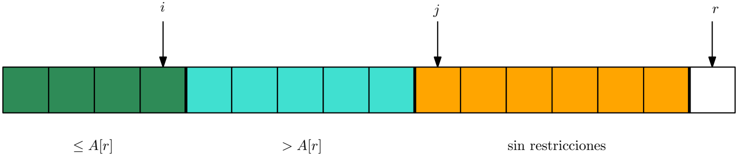
Partition

- 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 $k \in \{p, \ldots, i\}$
  2. $A[r] < A[k]$ para todo $k \in \{i+1, \ldots j-1\}$



Pseudo-Código para Partition

```
function Partition(A, p, r)
    i ← p − 1
    j ← p
    while j ≤ r do
        if A[j] ≤ A[r] then
            i ← i + 1
            A[i] ↔ A[j]
        j ← j + 1
    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:

```
function Random-Partition(A, p, r)
    i ← número aleatorio en {p, ..., r}
    A[r] ↔ A[i]
    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 $\{0, \ldots, k\}$
- Pregunta que inspira al algoritmo:

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

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

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

- **Objetivo:** construir arreglo $C[0 \ldots k]$ 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 $\{0, \ldots, k\}$)

```
procedure Counting-Sort(A, k)
    B ← copia del arreglo A
    C ← arreglo de tamaño k + 1 lleno de ceros
    for j ← 0 to len[A] − 1 do  C[A[j]] ← C[A[j]] + 1
```

Pseudo código para Counting Sort

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

procedure *Counting-Sort*(A, k)
    $B \leftarrow$ copia del arreglo A
    $C \leftarrow$ arreglo de tamaño $k+1$ lleno de ceros
    for $j \leftarrow 0$ **to** $len[A] - 1$ **do** $C[A[j]] \leftarrow C[A[j]] + 1$
    for $i \leftarrow 1$ **to** k **do** $C[i] \leftarrow C[i] + C[i-1]$
    for $j \leftarrow len[B] - 1$ **downto** 0 **do**
        $A[C[j] - 1] \leftarrow B[j]$
        $C[j] \leftarrow C[j] - 1$

Propiedad: Counting sort es correcto y **estable**.
Tiempo: Es $\Theta(k+n)$. En la práctica lo usamos si k es $O(n)$.

Radix Sort

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

- Un ejemplo de ejecución:

  329
  457
  839
  436
  720
  355

Radix Sort

- **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

Radix Sort

- **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

Radix Sort

- **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:

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

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 ← 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 $\Theta(d(n + k))$.

Otra Propiedad de Radix Sort

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

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 $b \geq \lfloor \log_2 n \rfloor$, hacemos $r = \lfloor \log_2 n \rfloor$

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 $b \geq \lfloor \log_2 n \rfloor$, 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

Bucket Sort

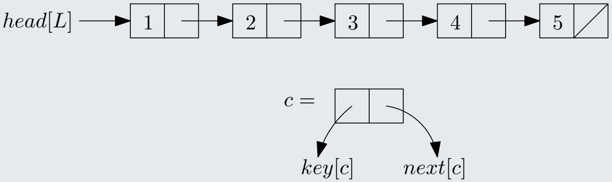
- **Input:** Arreglo A
- **Supuesto:** Las claves distribuyen uniformemente en $[0, 1)$
- **Idea fundadora:**

> Los datos se pueden separar en m cubetas (*buckets*)
>
> - En $B[0]$ ubicamos los elementos con clave en $[0, \frac{1}{m})$
> - En $B[1]$ ubicamos los elementos con clave en $[\frac{1}{m}, \frac{2}{m})$
> - En $B[k]$ ubicamos los elementos con clave en $[\frac{k}{m}, \frac{k+1}{m})$
>
> Cada bucket es una lista ligada **ordenada**.

- Ahora, en una pasada por A ubicamos, ordenadamente, a todos los números en su bucket
- Luego, recorremos los buckets copiando los números en el arreglo final

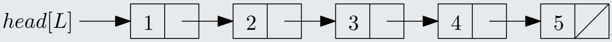
Listas Ligadas Simples con Sentinelas

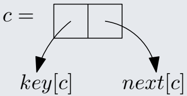
Lista ligada simple (sin sentinela):



(L está vacía ssi $head[L] = \text{NIL}$)

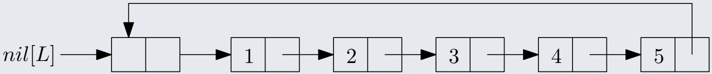
Lista ligada simple (sin sentinela):





(L está vacía ssi $head[L] = \text{NIL}$)

Lista ligada simple con sentinela:



(L está vacía ssi $next[nil[L]] = nil[L]$)

Pseudo Código para Bucket Sort

```
procedure Bucket-Sort(A, m)
    B ← arreglo de m listas vacías
    for i ← 0 to len[A] − 1 do
```

Pseudo Código para Bucket Sort

procedure *Bucket-Sort*(A, m)

  $B \leftarrow$ arreglo de m listas vacías

  for $i \leftarrow 0$ **to** $len[A] - 1$ **do**

    Ordered-Insert$(A[i], B[\lfloor mA[i] \rfloor])$

  $i \leftarrow 0$

  for $k \leftarrow 0$ **to** $m - 1$ **do**

    $\ell \leftarrow next[nil[B[k]]]$

    while $\ell \neq nil[B[k]]$ **do**

      $A[i] \leftarrow key[\ell]$

      $i \leftarrow i + 1$

      $\ell \leftarrow next[\ell]$

Pseudo Código Ordered Insert

(suponemos que L es una lista ligada simple con sentinelas)

```
procedure Ordered-Insert(x, L)
    new ← nueva celda con clave x
    c ← nil[L]
    while next[c] ≠ nil[L] and key[next[c]] ≥ x do
        c ← next[c]
    next[new] ← next[c]
    next[c] ← new
```

Análisis de Bucket Sort

Teorema: Si $m = n/C$ y C es una constante, el tiempo esperado de la ejecución de Bucket-Sort es $\Theta(n)$.

Demostración: pizarra

Estadísticas de Orden

- A veces es necesario obtener estadísticas de un arreglo A desordenado.
- Ejemplos: k-ésimo elemento más pequeño, mediana
- Existen formas de usar ideas detrás de algoritmos de ordenación para esto.

Idea de Quick Sort para i-ésimo Elemento Menor

(suponemos que $i \in \{1, \ldots, r - p + 1\}$)

procedure *Randomized-Select*(A, p, r, i)
    if $p = r$ **then return** $A[p]$
    $q \leftarrow$ *Randomized-Partition*(A, p, r)
    $k \leftarrow q - p + 1$
    if $i = k$ **then return** $A[q]$
    else if $i < k$ **then**
        Randomized-Select$(A, p, q - 1, i)$
    else if $i < k$ **then**
        Randomized-Select$(A, q + 1, r, i - k)$

Propiedad: *Randomized-Select* es correcto.
Propiedad: Tiempo esperado de ejecución es $O(n)$