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 $j \leftarrow 1$ **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:

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 ← ⌊(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:

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 ← nuevo arreglo de tamaño q − p + 2
    R ← nuevo arreglo de tamaño r − q + 1
    L[0..q − p] ← A[p..q]
    R[0..r − q − 1] ← A[q + 1..r]
    L[q − p + 1] ← R[r − q] ← ∞
    i ← j ← 0
    for k ← p to r do
        if L[i] ≤ R[j] then
            A[k] ← L[i]
            i ← i + 1
        else
            A[k] ← R[j]
            j ← 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-Insert*(A, i, key)
  if $A[i] < key$ **then** error "solo para disminuir clave"
  $A[i] \leftarrow key$
  while $i > 0$ *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-Insert(A, i, key)
    if A[i] < key then error "solo para disminuir clave"
    A[i] ← key
    while i > 0 and A[Parent(i)] > A[i] do
        A[Parent(i)] ↔ A[i]
        i ← 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] ← heap-size[A] + 1
    A[heap-size[A]] = −∞
    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)
  $l \leftarrow$ *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: Si i es el único nodo que no cumple la propiedad de heap en A, al terminar Min-Heapify(A, i), A contiene 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 ← 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: Si i es el único nodo que no cumple la propiedad de heap en A, al terminar Min-Heapify(A, i), A contiene 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, i)
  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)
```

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)

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)
    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

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

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

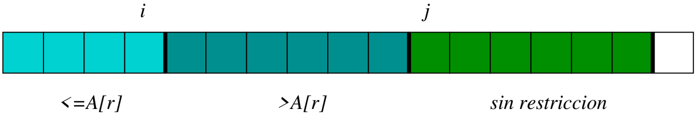
Observación: paso 2 debe ser $O(n)$

Pseudocódigo para Quick Sort

```
procedure Quick-Sort(A, p, q)
    if p < r then
        q ← Partition(A, p, q)
        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 $k \in \{i, \ldots j - 1\}$



Pseudo-Código para Paritition

procedure *Partition*(A, p, r)

    $i \leftarrow p - 1$

    $j \leftarrow p$

    $pivotkey \leftarrow A[r]$

    while $j \leq r$ **do**

        if $A[j] \leq pivotkey$ **then**

            $i \leftarrow i + 1$ $A[i] \leftrightarrow A[j]$

        $j \leftarrow j + 1$