Ordenamiento sobre Arreglos

Jesús Ravelo
Kelwin Fernández
Universidad Simón Bolívar
Dpto. de Computación y Tecnología de la Información

*** ***Observación:*** *esta versión es un borrador y posiblemente puede contener errores. Debe ser utilizada prudentemente.* ***

Resumen

Estas notas presentan el problema de ordenamiento sobre arreglos y sus soluciones clásicas más conocidas: los algoritmos de ordenamiento llamados inserción, selección, burbuja, mezcla, "quicksort" y "heapsort".

Presentaciones de todas estas soluciones abundan en la literatura sobre algoritmos, y pueden ser encontradas en [**Cormen et al. 09**] y [**Hume et al. 99**], entre muchas otras referencias. La diferencia entre estas notas y la típica presentación hallada en la literatura es que acá se utiliza la filosofía de métodos formales de programación, con especificaciones pre/post-condición para problemas y subrutias, invariantes de iteraciones y cotas de terminación para iteraciones y recursiones. En ese sentido, estas notas siguen la filosofía de textos como [**Kaldewaij 90**], [**Gries 81**] y [**Backhouse 03**], aunque no se hace énfasis en derivación formal de programas como en la primera de estas tres referencias, sino en construcción intuitiva de los algoritmos como en los textos estándares de programación.

1. Introducción

Supongamos que queremos ordenar el siguiente arreglo de enteros ascendentemente según la relación $\leqslant$:

6	1	5	2	6	12	3	3	7	8

Una posible solución sería el siguiente arreglo:

1	2	3	3	5	6	6	7	8	12

Nótese que en el arreglo final todos los elementos preservan el número de ocurrencias que tenían en el arreglo inicial.

A continuación supongamos que deseamos ordenar el siguiente arreglo de conjuntos ascendentemente según su cardinalidad:

$\{true, false\}$	$\{1, 2, 4, 8, 16, 32\}$	$\{a, k, f\}$	ϕ	$\{\clubsuit, \spadesuit, \bigstar\}$

Podríamos tener como posible solución el arreglo:

ϕ	$\{true, false\}$	$\{a, k, f\}$	$\{\clubsuit, \spadesuit, \bigstar\}$	$\{1, 2, 4, 8, 16, 32\}$

O, dado que la cardinalidad de $\{a, k, f\}$ es igual a la cardinalidad de $\{\clubsuit, \spadesuit, \bigstar\}$, podríamos tener el siguiente:

ϕ	$\{true, false\}$	$\{\clubsuit, \spadesuit, \bigstar\}$	$\{a, k, f\}$	$\{1, 2, 4, 8, 16, 32\}$

Generalizando, podemos decir que el problema de ordenar un arreglo se sintetiza en: dado un arreglo cuyos elementos son de algún tipo T y dada una relación de orden total[1] $\sqsubseteq$ sobre T, devolver una permutación del arreglo ordenada (ascendente o descendentemente) según dicha relación.

[1]Definiciones formales de varios tipos de relaciones de orden, y en particular de relaciones de orden total, pueden ser conseguidos en textos de matemáticas discretas. Por ejemplo, *D. Gries & Schneider, A Logical Approach to Discrete Math, Springer-Verlag, 1993* y *D.F. Stanat & D.F. McAllister, Discrete Mathematics in Computer Science, Prentice Hall, 1977.* A efectos de lo que necesitamos en estas notas, basta que nuestra relación sea reflexiva, transitiva y conexa.

2. Algoritmos Clásicos de Ordenamiento

Los algoritmos clásicos de ordenamiento sobre arreglos se dividen principalmente en dos grandes familias según la complejidad algorítmica de sus casos promedios[2].

Familia $\Theta(n^2)$

Los algoritmos más representativos de esta familia son:

- Inserción.
- Selección.
- Burbuja.

Familia $\Theta(n \cdot lg\ n)$

Los algoritmos más representativos de esta familia son:

- Mezcla.
- "Quicksort".
- "Heapsort".

3. Especificación del Problema de Ordenamiento

En esta sección utilizaremos comúnmente las siguientes definiciones de funciones auxiliares:

- $bag.a[x..y) = \lbag\ i \mid x \leqslant i < y : a[i]\ \rbag$
- $ord.a[x..y) \equiv (\forall i, j \mid x \leqslant i < j < y : a[i] \sqsubseteq a[j])$
- $a[x..y) \sqsubseteq b[w..z) \equiv (\forall i, j \mid x \leqslant i < y\ \wedge\ w \leqslant j < z : a[i] \sqsubseteq b[j])$

[2] Aún sin tener conocimientos previos acerca del estudio de la complejidad algorítmica considérese que se corresponde con tener un tiempo de ejecución "cercano" a múltiplo, en nuestro caso, de n^2 o a múltiplo de $n \cdot lg\ n$

donde *bag* devuelve el multiconjunto de elementos contenidos en algún segmento de un arreglo, *ord* indica si un segmento de un arreglo está ordenado, y la última abreviación da un nuevo uso al símbolo "$\sqsubseteq$" para indicar que todos los elementos de un segmento son no-mayores a todos los elementos de otro segmento. Esta última abreviación también nos permitiremos usarla con la relación estricta "$\sqsubset$" (esto es, "menor") asociada a la original "$\sqsubseteq$" ("menor o igual").

Para el último, utilizaremos comúnmente un pequeño abuso de notación y lo ampliaremos no sólo a segmentos de arreglos sino a elementos simples. Por ejemplo, nos referiremos a $e \sqsubseteq b[x..y)$ para indicar que e es "menor o igual" a todos los pertenecientes al arreglo b entre los indices x e y, esto sería equivalente a $a[0..1) \sqsubseteq b[x..y)$ donde $a[0] = e$.

A estas alturas ya podemos dar una especificación formal del problema de ordenamiento de arreglos:

$$
\begin{array}{|l}
|[\\
\quad \textbf{const}\ \ n\text{: } \mathit{int}; \\
\quad \textbf{var}\ \ a\text{: } \mathit{array[0..n)\ of}\ T; \\
\quad \{\ \textbf{Pre:}\quad n \geq 0\ \wedge \mathit{bag}.a[0..n) = B\ \} \\
\quad\quad \mathit{ORDENAR} \\
\quad \{\ \textbf{Post:}\quad \mathit{ord}.a[0..n)\ \wedge\ \mathit{bag}.a[0..n) = B\ \} \\
]|
\end{array}
$$

Cuando ordenamos cosas diariamente (libros en una estantería, un mazo de cartas, etc), solemos tratar de ir ordenando un segmento del espacio de elementos y vamos agregando uno a uno los restantes en orden. Inicialmente tenemos que el segmento ordenado es vacío, es decir, el segmento $[0..0)$, luego procedemos a agregar un elemento, con lo cual tendremos ordenado el segmento $[0..1)$, luego el $[0..2)$ y así sucesivamente hasta haber agregado la totalidad de los elementos (n), con lo cual tendremos ordenado el total, segmento $[0..n)$, de los elementos.

Podemos proponer el siguiente invariante que cumple con la idea antes presentada:

$$
\left\{
\begin{array}{ll}
\textbf{Inv:} & 0 \leqslant k \leqslant n\ \wedge\ (\forall i, j \mid 0 \leqslant i < j < k : a[i] \sqsubseteq a[j]) \\
& \wedge\ \mathit{bag}.a[0..n) = B
\end{array}
\right\}
$$

el cual es equivalente, según la definición de ord, al siguiente invariante:

$$\{\ \textbf{Inv:}\quad 0 \leqslant k \leqslant n\ \wedge\ ord.a[0..k)\ \wedge\ bag.a[0..n) = B\ \}$$

Como función de cota podemos proponer $n - k$, y dado que el segmento ordenado inicialmente es vacío, la inicialización de k debe ser en cero.

Tenemos entonces hasta ahora lo siguiente:

```
|[
  const n: int;
  var a: array[0..n) of T;
  { Pre:  n ≥ 0 ∧ bag.a[0..n) = B }

  var k: int;
  k := 0;

  { Inv:  0 ⩽ k ⩽ n ∧ ord.a[0..k) ∧ bag.a[0..n) = B
    Cota: n − k                                      }
  do k ≠ n ⟶
     [ ? ]
     ; k := k + 1
  od

  { Post: ord.a[0..n) ∧ bag.a[0..n) = B }
]|
```

Detengámonos un momento a ver el siguiente arreglo en un momento de ejecución del procedimiento anterior en el cual k = 4:

1	2	4	14	0	3

Observación: Casillas sombreadas se refieren al segmento ya ordenado.

Dicho estado del arreglo cumple con el invariante dado que el segmento $[0..k)$ se encuentra ordenado, sin embargo, nótese que aún quedan elementos en el segmento de la derecha que finalmente desplazarán a los elementos ya ordenados de su posición actual.

Podemos entonces ser un poco más estrictos y exigir en el invariante que los elementos del segmento $[0..k)$ estén en su posición final, para esto se

utilizaría el predicado $a[0..k) \sqsubseteq a[k..n)$ dado que si todos los elementos en el segmento inicial son menores que aquellos pertenecientes al segmento final, no es posible que estos últimos desplacen a los primeros.

Nuestra segunda versión de invariante quedaría:

$$\left\{ \begin{array}{ll} \textbf{Inv:} & 0 \leqslant k \leqslant n \ \wedge \ ord.a[0..k) \ \wedge \ a[0..k) \sqsubseteq a[k..n) \\ & \wedge \ bag.a[0..n) = B \end{array} \right\}$$

y nótese que el bosquejo de iteración principal quedaría como antes

$$\begin{array}{l} |[\\ \quad \textbf{const} \ \ n\text{: } int; \\ \quad \textbf{var} \ \ a\text{: } array[0..n) \ of \ T; \\ \quad \{ \ \textbf{Pre:} \ \ n \geq 0 \ \wedge bag.a[0..n) = B \ \} \\ \\ \quad \textbf{var} \ \ k\text{: } int; \\ \quad k := 0; \\ \\ \quad \left\{ \begin{array}{ll} \textbf{Inv:} & 0 \leqslant k \leqslant n \ \wedge \ ord.a[0..k) \ \wedge \ a[0..k) \sqsubseteq a[k..n) \\ & \wedge \ bag.a[0..n) = B \\ \textbf{Cota:} & n - k \end{array} \right\} \\ \quad \underline{\textbf{do}} \ k \neq n \longrightarrow \\ \quad\quad \boxed{?} \\ \quad\quad ; k := k + 1 \\ \quad \underline{\textbf{od}} \\ \\ \quad \{ \ \textbf{Post:} \ \ ord.a[0..n) \ \wedge \ bag.a[0..n) = B \ \} \\ ]| \end{array}$$

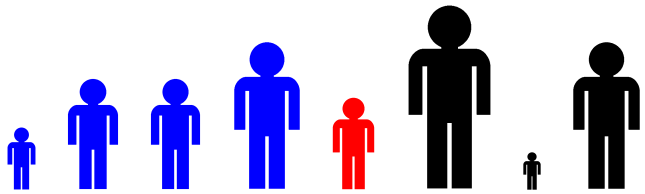
4. Inserción

El ordenamiento por inserción puede que sea el más natural de los ordenamientos. Lo aplicamos casi por instinto cuando queremos ordenar un mazo de cartas, al ordenar libros en una biblioteca por tamaño o por orden lexicográfico, entre otros.

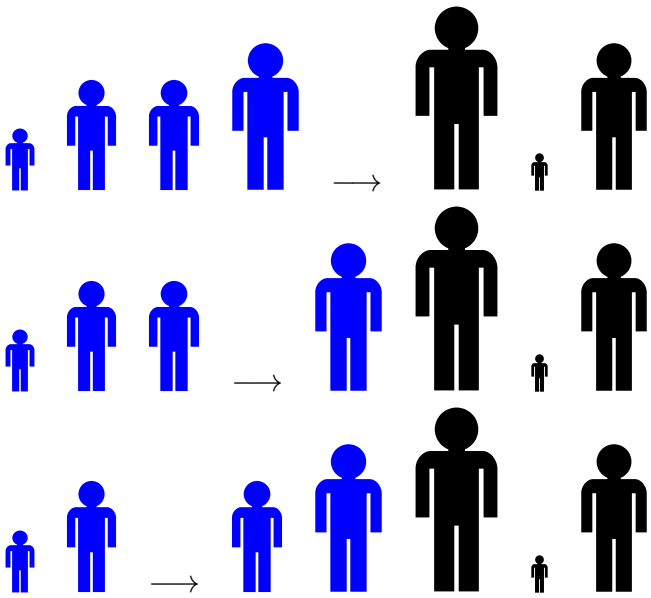
Supongamos que somos profesores de primaria y debemos ordenar a nuestros alumnos por orden de tamaño. Les pedimos que formen una fila, luego vamos pasando por cada uno de ellos y cada vez que avancemos adelantamos a cada niño, en caso de ser necesario, a su posición correcta, esto es, a la

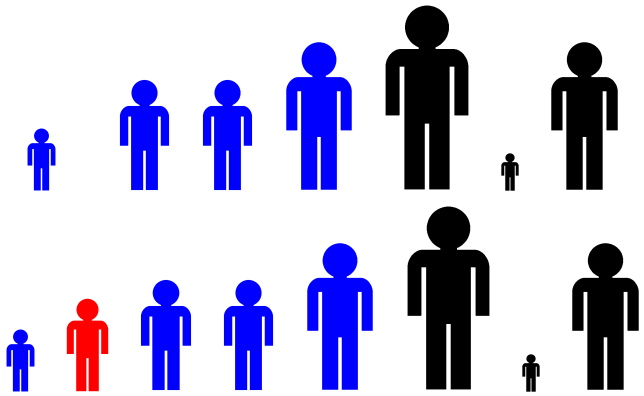
posición donde el anterior es de menor o igual tamaño y el siguiente es de mayor tamaño (es indiferente en que sentido queda la igualdad). Para esto, será necesario desplazar a todos los que queden detrás de él. Esto induce que, en cada paso, el conjunto de niños por los que ya hemos pasado estén ordenados mediante la relación de tamaño; al haber pasado por todos los niños, la fila estará ordenada.

Veamos por ejemplo el siguiente gráfico, donde los niños azules se refieren al segmento ya ordenado y el niño en rojo se trata del que colocaremos en su posición a continuación:

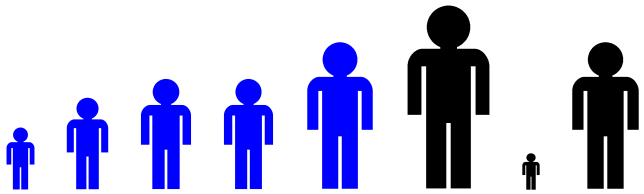


Digamos que el arreglo de niños es llamado a y que cada posición equivale a una casilla en el arreglo. Para este punto tenemos que el segmento ya ordenado es $a[0..k)$, lo cual podemos escribirlo como $ord.a[0..4)$, con $k = 4$. A continuación insertaremos al niño rojo en su posición correcta; para esto desplazaremos a todos aquellos que sean necesarios para poder insertarlo:





Ya habiendo hecho esto, contamos con que el segmento ordenado ha sido incrementado en uno y, por tanto, tenemos que $ord.a[0..k+1)$ se cumple (con $k = 4$).



Nótese que esto concuerda perfectamente con nuestra primera versión de invariante.

De esta manera funciona el ordenamiento por inserción: pasa por cada uno de los elementos del arreglo, lo *inserta* en su posición correcta a la vez que desplaza a todos los demás. Por ello, funciona con nuestra primera versión (más débil) de invariante, ya que los elementos del segmento ordenado no necesariamente están en su posición definitiva.

Con base en la primera propuesta de solución al problema de ordenamiento, basada en la primera propuesta de invariante, el contrato a ser satisfecho por lo que falta por ser desarrollado (código $\boxed{?}$) es:

$$\{0 \leqslant k < n \ \wedge \ ord.a[0..k) \ \wedge \ bag.a[0..n) = B\}$$

$$\boxed{?}$$

$$\{0 \leqslant k \leqslant n \ \wedge \ ord.a[0..k+1) \ \wedge \ bag.a[0..n) = B\}$$

Según tal contrato debemos en cada iteración colocar el elemento $a[k]$ en su posición correcta. Para esto, utilizaremos una variable *temp* que mantendrá el valor inicial de $a[k]$, puesto que debemos desplazar hacia la derecha tantos elementos como sean necesarios. Tenemos entonces lo siguiente:

$$
\begin{array}{l}
|[\ \ \textbf{var}\ \ temp\colon T; \\
\quad\ \ l\colon int; \\
\quad temp, l := a[k], k; \\
\quad \\
\quad \underline{\textbf{do}}\ l \neq 0\ \wedge\ temp \sqsubset a[l-1] \longrightarrow \\
\quad\quad a[l] := a[l-1]; \\
\quad\quad l := l-1 \\
\quad \underline{\textbf{od}} \\
\quad \\
\quad ;\ a[l] := temp \\
]|
\end{array}
$$

Claramente como cota para la iteración podemos colocar a l.

La construcción del invariante es un tanto más delicada. Podemos comenzar enunciando el rango en el que varía l, el cual está comprendido entre 0 y k, ambos extremos incluídos. Recordemos entonces que estamos desplazando a los elementos mayores a *temp* para poder insertar a este último en su posición. Por lo tanto, los segmentos $a[0..l)$ y $a[l+1..k+1)$ siguen estando ordenados. Además, todos los elementos que hemos ido dejando a la derecha, es decir, todos los $a[i]$ con $l < i \leqslant k$ son mayores que *temp*. Hasta ahora tenemos como invariante lo siguiente:

$$
\left\{
\begin{array}{ll}
\textbf{Inv:} & 0 \leqslant l \leqslant k\ \wedge\ ord.a[0..l)\ \wedge\ ord.a[l+1..k+1) \\
 & \wedge\ temp \sqsubset a[l+1..k+1)
\end{array}
\right\}
$$

Nos queda sólo documentar cómo el multiconjunto de elementos está siendo mantenido. Sin embargo, dado que estamos desplazando elementos sin hacer intercambios directos no podemos decir que $bag.a[0..n) = B$.

Podemos notar que, salvo el elemento que originalmente estaba en $a[k]$, el cual inicialmente almacenamos en *temp*, sólo hemos hecho desplazamientos en el arreglo. En cualquiera de las vueltas, los desplazamientos que hemos realizado han hecho que el segmento *a[l+1..k+1)* sea el resultado de haber movido cada uno de ellos una posición a la derecha. Nótese que, de haber

hecho al menos un movimiento ya, resulta que el elemento $a[l]$ está duplicado (en $a[l+1]$) y debemos ignorarlo. Este "hueco" en el arreglo debe ser completado con el elemento que tenemos en *temp*. Podemos deducir que:

$$bag.a[0..l) \ \uplus \ bag.a[l+1..n) \ \uplus \ [\![temp]\!] = B$$

donde el operador $\uplus$ se refiere a la unión de multiconjuntos, la cual suma la multiplicidad de sus elementos, y $[\![x]\!]$ denota un multiconjunto unitario.

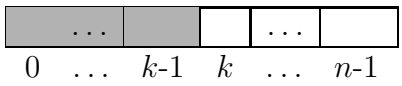
Finalmente contamos con la siguiente implementación de Ordenamiento por Inserción:

$$
\begin{array}{l}
|[\\
\quad \textbf{const}\ \ n\colon\ int; \\
\quad \textbf{var}\ \ a\colon\ array[0..n)\ of\ T; \\
\quad \{\ \textbf{Pre:}\quad n \geq 0\ \wedge bag.a[0..n) = B\ \} \\
\\
\quad \textbf{var}\ \ k\colon int; \\
\quad k := 0; \\
\\
\quad \left\{ \begin{array}{ll} \textbf{Inv:} & 0 \leqslant k \leqslant n\ \wedge\ ord.a[0..k)\ \wedge\ bag.a[0..n) = B \\ \textbf{Cota:} & n - k \end{array} \right\} \\
\quad \underline{\textbf{do}}\ k \neq n \longrightarrow \\
\quad\quad \textbf{var}\ \ temp\colon T; \\
\quad\quad\quad\ \ l\colon int; \\
\quad\quad temp, l := a[k], k; \\
\\
\quad\quad \left\{ \begin{array}{ll} \textbf{Inv:} & 0 \leqslant l \leqslant k\ \wedge\ ord.a[0..l)\ \wedge\ ord.a[l+1..k+1) \\ & \wedge\ bag.a[0..l)\ \uplus\ bag.a[l+1..n)\ \uplus\ [\![temp]\!] = B \\ & \wedge\ temp \sqsubset a[l+1..k+1) \\ \textbf{Cota:} & l \end{array} \right\} \\
\quad\quad \underline{\textbf{do}}\ l \neq 0\ \wedge\ temp \sqsubset a[l-1] \longrightarrow \\
\quad\quad\quad\ a[l] := a[l-1]; \\
\quad\quad\quad\ l := l - 1 \\
\quad\quad \underline{\textbf{od}} \\
\\
\quad\quad ; a[l] := temp \\
\quad\quad ; k := k + 1 \\
\quad \underline{\textbf{od}} \\
\\
\quad \{\ \textbf{Post:}\quad ord.a[0..n)\ \wedge\ bag.a[0..n) = B\ \} \\
]|
\end{array}
$$

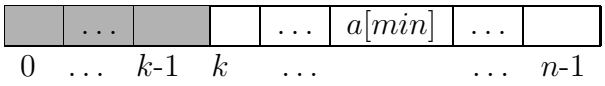
5. Selección

Para el desarrollo de este algoritmo utilizaremos la segunda versión de invariante más fuerte que propusimos anteriormente. Esta exige adicionalmente a que el segmento $a[0..k)$ se encuentre ordenado, que cada elemento

se encuentre en su posición final. Digamos que el siguiente es un momento cualquiera en la ejecución del algoritmo:



Dado nuestro invariante para la iteración externa, sabemos que justo antes de ejecutarse el cuerpo de la iteración se tiene que $ord.a[0..k)$ y $a[0..k) \sqsubseteq a[k..n)$. De esto se puede inferir que el segmento $a[0..k)$ tiene los k menores elementos de $bag.a[0..n)$. Para poder conservar el invariante, debemos asegurar que en la próxima vuelta el segmento $a[0..k+1)$ tiene los $k+1$ menores elementos del multiconjunto. Por lo tanto, debemos conseguir el menor elemento del segmento $a[k..n)$ y añadirlo a nuestro segmento ya ordenado. ¿En qué posición debemos colocarlo? Dado que debemos garantizar que el segmento $a[0..k+1)$ se mantenga ordenado para así cumplir el invariante en la próxima iteración, se debe colocar este elemento en la posición k del arreglo. Sea $a[min]$ el menor elemento del segmento $a[k..n)$:



pudiendo min ser igual a k o a $n-1$. Debemos intercambiar las casillas $a[k]$ y $a[min]$, así tendremos $ord.a[0..k+1)$, $a[0..k+1) \sqsubseteq a[k+1..n)$ y adicionalmente, dado que sólo se realizó un intercambio de elementos, se garantiza que $bag.a[0..n) = B$.

Veamos la siguiente implementación para conseguir el menor elemento en el intervalo $a[k..n)$:

$$
\begin{array}{l}
|[\ \ \textbf{var}\ \ l, min\colon int; \\
\quad | \\
\quad l, min := k+1, k; \\
\quad | \\
\quad \left\{ \begin{array}{ll} \textbf{Inv:} & k+1 \leqslant l \leqslant n \ \wedge\ k \leqslant min < n \ \wedge\ a[min] \sqsubseteq a[k..l) \\ \textbf{Cota:} & n-l \end{array} \right\} \\
\quad \underline{\textbf{do}}\ l \neq n \longrightarrow \\
\quad\quad | \\
\quad\quad \underline{\textbf{if}}\ a[l] \sqsubseteq a[min] \longrightarrow \\
\quad\quad\quad |\quad min := l \\
\quad\quad\quad [] \ a[min] \sqsubseteq a[l] \longrightarrow \\
\quad\quad\quad |\quad skip \\
\quad\quad \underline{\textbf{fi}} \\
\quad\quad ; l := l+1 \\
\quad\quad | \\
\quad \underline{\textbf{od}} \\
]|
\end{array}
$$

Como modificamos únicamente las variables locales l y min, se garantiza además el invariante externo en cada iteración del ciclo interno. Podemos armar el algoritmo completo teniendo como resultado:

```
|[ const n: int;
  var a: array[0..n) of T;
  { Pre:  n ≥ 0 ∧ bag.a[0..n) = B }

  var k: int;
  k := 0;

  { Inv:  0 ⩽ k ⩽ n ∧ ord.a[0..k) ∧ a[0..k) ⊑ a[k..n)
          ∧ bag.a[0..n) = B
    Cota: n − k                                          }
  do k ≠ N ⟶

    var l, min: int;
    l, min := k + 1, k;

    { Inv:  k + 1 ⩽ l ⩽ n ∧ k ⩽ min < n ∧ a[min] ⊑ a[k..l) }
    { Cota: n − l                                          }
    do l ≠ n ⟶

      if a[l] ⊑ a[min] ⟶
      |    min := l
      [] a[min] ⊑ a[l] ⟶
      |    skip
      fi
      ; l := l + 1

    od

    ; a[k], a[min] := a[min], a[k]
    ; k := k + 1
  od

  { Post:  ord.a[0..n) ∧ bag.a[0..n) = B }
]|
```

6. Burbuja

El Ordenamiento de Burbuja (*Bubble Sort*), o de Intercambio Directo trabaja semejantemente al de Selección: en cada paso lleva al menor elemento del segmento sin ordenar a su posición final, esto es, al extremo izquierdo de ese segmento. La diferencia recae en el mecanismo con el cual lo hace: mientras que el ordenamiento por selección escoge al menor elemento del segmento no ordenado y lo lleva a su posición final en un solo movimiento, el ordenamiento por burbuja realiza intercambios de pares de elementos adyacentes en caso de que estos se encuentren en orden incorrecto hasta que se logra el objetivo.

Su nombre deriva del aspecto que toman los elementos más livianos del arreglo al subir al tope como si fueran *burbujas*, si visualizamos el arreglo de forma vertical descendente. Como consecuencia, el arreglo se irá ordenando desde las posiciones inferiores. Veamos por ejemplo el siguiente arreglo al cual le aplicaremos una pasada del algoritmo. Nótese como el menor elemento, señalado con sombreado, asciende mediante intercambio con su vecino inmediato hasta su posición final. Los elementos entre los cuales se realiza el intercambio se presentan resaltados en negritas.

5	5	5	**5**	1
8	8	**8**	**1**	5
4	**4**	**1**	8	8
1	**1**	4	4	4
3	3	3	3	3

Sea l una variable que irá recorriendo los índices del arreglo para ejecutar los intercambios. En cada paso, debemos intercambiar los elementos de las posiciones l y $l-1$ en caso de que estos se encuentren en orden incorrecto, es decir, si $a[l-1] \sqsupset a[l]$. Tenemos entonces que para el intercambio podemos dar la siguiente solución:

$$
\begin{array}{l}
\underline{\textbf{if}}\ a[l-1] \sqsupset a[l] \longrightarrow \\
\quad\ \ a[l], a[l-1] := a[l-1], a[l] \\
[]\ a[l-1] \sqsubseteq a[l] \longrightarrow \\
\quad\ \ \mathit{skip} \\
\underline{\textbf{fi}}
\end{array}
$$

Las guardas de este condicional inducen que la variable l debe encontrarse en el rango $k < l < n$. Por lo tanto tenemos que el recorrido en el arreglo debe ser de la siguiente forma:

$$
\begin{array}{l}
|[\\
\quad \textbf{var}\ \ l\colon int; \\
\quad l := n - 1; \\
\quad \\
\quad \left\{ \begin{array}{ll} \textbf{Inv:} & k \leqslant l < n \ \wedge\ a[l] \sqsubseteq a[l+1..n) \\ \textbf{Cota:} & l \end{array} \right\} \\
\quad \underline{\textbf{do}}\ l \neq k \longrightarrow \\
\quad\quad \\
\quad\quad \underline{\textbf{if}}\ a[l-1] \sqsupset a[l] \longrightarrow \\
\quad\quad\quad\ \ a[l], a[l-1] := a[l-1], a[l] \\
\quad\quad [] \ a[l-1] \sqsubseteq a[l] \longrightarrow \\
\quad\quad\quad\ \ skip \\
\quad\quad \underline{\textbf{fi}} \\
\quad\quad \\
\quad\quad ; l := l - 1 \\
\quad \textbf{od} \\
]|
\end{array}
$$

Adicionalmente, en cada iteración del ciclo interno, se cumple el invariante del ciclo externo. Por lo tanto, debemos fortalecer este como:

$$
\left\{ \begin{array}{ll} \textbf{Inv:} & 0 \leqslant k \leqslant l < n \ \wedge\ ord.a[0..k) \ \wedge\ a[0..k) \sqsubseteq a[k..n) \\ & \wedge\ bag.a[0..n) = B \ \wedge\ a[l] \sqsubseteq a[l+1..n) \end{array} \right\}
$$

Finalmente tenemos como implementación para el Ordenamiento de Burbuja lo siguiente:

```
|[ const n: int;
 | var a: array[0..n) of T;
 |
 | { Pre:  n ≥ 0 ∧ bag.a[0..n) = B }
 |
 | var k: int;
 | k := 0;
 |
 | { Inv:  0 ⩽ k ⩽ n ∧ ord.a[0..k) ∧ a[0..k) ⊑ a[k..n)
 | |       ∧ bag.a[0..n) = B
 | | Cota: n − k }
 | do k ≠ n ⟶
 |  | var l: int;
 |  | l := n − 1;
 |  |
 |  | { Inv:  0 ⩽ k ⩽ l < n ∧ ord.a[0..k) ∧ a[0..k) ⊑ a[k..n)
 |  | |       ∧ bag.a[0..n) = B ∧ a[l] ⊑ a[l + 1..n)
 |  | | Cota: l }
 |  | do l ≠ k ⟶
 |  |  |
 |  |  | if a[l − 1] ⊐ a[l] ⟶
 |  |  | |    a[l], a[l − 1] := a[l − 1], a[l]
 |  |  | [] a[l − 1] ⊑ a[l] ⟶
 |  |  | |    skip
 |  |  | fi
 |  |  |
 |  |  | ; l := l − 1
 |  | od
 |  | ; k := k + 1
 | od
 |
 | { Post:  ord.a[0..n) ∧ bag.a[0..n) = B }
]|
```

La práctica muestra que en el caso promedio resulta que no es necesario ejecutar todas las pasadas para que el arreglo quede finalmente ordenado. Analice a qué puede deberse el que no se realice ningún intercambio en la iteración interna.

En respuesta a esto, suele agregarse una variable booleana que indica si se efectuaron o no intercambios a lo largo de la pasada interna, para de esta manera interrumpir el ciclo dando por ordenado el arreglo. Para esto podemos hacer lo siguiente:

$$
\begin{array}{l}
|[\\
\quad \vdots \\
\quad huboCambios := false; \\
\quad \left\{ \begin{array}{ll} \textbf{Inv:} & k \leqslant l < n \ \wedge \ a[l] \sqsubseteq a[l+1..n) \\ & \wedge \neg huboCambios \Rightarrow \ ord.a[l..n) \\ \textbf{Cota:} & l \end{array} \right\} \\
\quad \underline{\textbf{do}}\ l \neq k \longrightarrow \\
\quad\quad \underline{\textbf{if}}\ a[l-1] \sqsupset a[l] \longrightarrow \\
\quad\quad\quad\quad a[l], a[l-1] := a[l-1], a[l]; \\
\quad\quad\quad\quad huboCambios := true \\
\quad\quad [] \ a[l-1] \sqsubseteq a[l] \longrightarrow \\
\quad\quad\quad\quad skip \\
\quad\quad \underline{\textbf{fi}} \\
\quad\quad ; l := l - 1 \\
\quad \underline{\textbf{od}} \\
]|
\end{array}
$$

Finalmente tenemos como implementación para el Ordenamiento de Burbuja "Mejorado" lo siguiente:

```
|[ const n: int;
  var a: array[0..n) of T;

  { Pre:  n ≥ 0 ∧ bag.a[0..n) = B }

  var k: int;
  k := 0;

  { Inv:  0 ⩽ k ⩽ n ∧ ord.a[0..k) ∧ a[0..k) ⊑ a[k..n)
          ∧ bag.a[0..n) = B
    Cota: n − k }
  do k ≠ n ⟶

    var l: int;
        huboCambios: boolean;
    l := n − 1;
    huboCambios := false;

    { Inv:  0 ⩽ k ⩽ l < n ∧ ord.a[0..k) ∧ a[0..k) ⊑ a[k..n)
            ∧ bag.a[0..n) = B ∧ a[l] ⊑ a[l + 1..n)
            ∧ ¬huboCambios ⇒ ord.a[l..n)
      Cota: l }
    do l ≠ k ⟶

      if a[l − 1] ⊐ a[l] ⟶
          a[l], a[l − 1] := a[l − 1], a[l];
          huboCambios := true
      [] a[l − 1] ⊑ a[l] ⟶
          skip
      fi

      ; l := l − 1
    od

    ; k := k + 1
  od

  { Post:  ord.a[0..n) ∧ bag.a[0..n) = B }
]|
```

7. Divide y vencerás

Ya hemos tratado los tres primeros algoritmos de ordenamiento, todos ellos con complejidad $\Theta(n^2)$ en peor caso. A continuación veremos otra perspectiva mediante la cual implementaremos los algoritmos de ordenamiento con complejidad $\Theta(n \cdot lg\ n)$.

Supongamos que tenemos que ordenar una pila inmensa de facturas por fecha para poder archivarlas. Claramente intentaremos invertir la menor cantidad de tiempo posible en ello. Por lo tanto, le pedimos ayuda a un amigo, cada uno ordenará un lote de facturas y luego reordenaremos ambos grupos en uno solo. Pero resulta que los bloques a ordenar siguen siendo demasiado grandes y aún así no tendremos tiempo de terminar, entonces cada uno le pide ayuda a dos amigos más para ordenar los lotes y así sucesivamente; luego, cada dúo deberá únicamente reordenar ambos bloques para entregárselos a su amigo.

Otra posible solución sería entregarle a un amigo las facturas más viejas y quedarnos nosotros con las más nuevas, luego cada uno de nosotros dividirá de nuevo cada grupo en las más viejas y más nuevas para pedirle ayuda a otros dos amigos y así sucesivamente. La acción de ordenar se va ejecutando a medida que separamos los grupos de facturas según su fecha.

Entonces para facilitar el trabajo de ordenación podemos dividir el espacio, trabajar cada segmento y luego construir la solución. Esta forma de atacar el problema de ordenar un arreglo presenta naturaleza recursiva. La división del problema finaliza cuando el segmento se encuentra trivialmente ordenado, esto podemos hacerlo para segmentos de tamaño menores a dos.

Sea el siguiente procedimiento encargado de la ordenación:

```
proc ordenar (in n: int; in-out a: array[0..n) of T; in izq,der: int)
  { Pre:  0 ⩽ izq ⩽ der ⩽ n ∧ bag.a[izq..der) = C
          ∧ a[0..izq) = A0 ∧ a[der..n) = A1 }
  { Post: ord.a[izq..der) ∧ bag.a[izq..der) = C
          ∧ a[0..izq) = A0 ∧ a[der..n) = A1 }
  { Cota: der − izq }
|[
  if der − izq < 2 ⟶
     skip
  [] der − izq ≥ 2 ⟶
     var x: int;

     ?
     ordenar(n, a, izq, x);
     ordenar(n, a, x, der);
     ?
  fi
]|
```

donde los parámetros *izq* y *der* delimitan el segmento a ordenar, la variable x indica el punto en el que se dividirá el espacio, y los signos de interrogación se corresponden respectivamente con la *división* y *recombinación* del espacio a ordenar. Claramente, si vamos a dividir el trabajo buscaremos que ambos segmentos sean más o menos del mismo tamaño para evitar que una persona quede ociosa esperando a que la otra termine de ordenar el otro segmento.

Nótese que la precondición exige que nadie pueda alterar el segmento de elementos de otra persona. Cada instancia de la recursión debe devolver el segmento $a[izq..der)$ ordenado y, los otros segmentos $a[0..izq)$ y $a[der..n)$ deben permanecer inalterados.

8. Mezcla

El Ordenamiento por Mezcla (*Merge Sort*) fue desarrollado por John von Neumann en el año 1945. Veamos la siguiente implementación para el algoritmo de ordenamiento por mezcla:

proc ordenar(**in** n: *int*; **in-out** a: *array[0..n) of T*; **in** izq, der: *int*)

$\left\{ \begin{array}{l} \textbf{Pre:} \quad 0 \leqslant izq \leqslant der \leqslant n \ \wedge \ bag.a[izq..der) = C \\ \qquad\quad \wedge \ a[0..izq) = A_0 \ \wedge \ a[der..n) = A_1 \end{array} \right\}$

$\left\{ \begin{array}{l} \textbf{Post:} \quad ord.a[izq..der) \ \wedge \ bag.a[izq..der) = C \\ \qquad\quad \wedge \ a[0..izq) = A_0 \ \wedge \ a[der..n) = A_1 \end{array} \right\}$

$\{$ **Cota:** $\quad der - izq \ \}$

```
|[
  if der − izq < 2 −→
       skip
  [] der − izq ≥ 2 −→
       var med: int;
       med := (izq + der) div 2;

       {izq < med < der}

       ordenar(n, a, izq, med);
       ordenar(n, a, med, der);
       mezclar(n, a, izq, med, der)
  fi
]|
```

También podríamos asignar a *med* el valor obtenido de $(izq + der + 1)div\ 2$ y seguir cumpliendo con el contrato. Queda como ejercicio para el lector demostrar esto.

A continuación debemos definir el procedimiento *mezclar*. Éste se encargará de componer la solución al problema completo ordenando los dos segmentos en uno solo. Este procedimiento deberá seguir la restricción de que sólo puede alterar el espacio a ordenar que le toca; además deberá ordenar y juntar ambos segmentos en uno solo. Presentaremos la siguiente especificación para el procedimiento:

proc mezclar(**in** n: *int*; **in-out** a: *array[0..n) of T*;
in izq, med, der: *int*)

$\left\{ \begin{array}{l} \textbf{Pre:} \quad 0 \leqslant izq < med < der \leqslant n \ \wedge \ bag.a[0..n) = D \\ \qquad\quad \wedge \ a[0..izq) = S_0 \ \wedge \ a[der..n) = S_1 \end{array} \right\}$

$\left\{ \begin{array}{l} \textbf{Post:} \quad bag.a[0..n) = D \ \wedge \ ord.a[izq..der) \\ \qquad\quad \wedge \ a[0..izq) = S_0 \ \wedge \ a[der..n) = S_1 \end{array} \right\}$

Para mezclar los dos segmentos utilizaremos un arreglo auxiliar que irá recibiendo en orden los elementos de ambos segmentos. Podemos entonces establecer tres variables i, j y k que indicarán el elemento por el que vamos desplazándonos en el primer y segundo segmento, y en el arreglo auxiliar, respectivamente. Sería posible eliminar la variable k puesto que ésta en cada instante de la iteración tendrá el valor $(i\text{-}izq) + (j\text{-}med)$, pero por cuestiones de legibilidad y para reducir el número de operaciones, la usaremos.

En cada paso de la iteración debemos escoger el siguiente menor elemento de ambos segmentos y pasarlo al arreglo auxiliar. Dado que ambos segmentos están ordenados, tenemos que el siguiente menor elemento del segmento de la izquierda será el $a[i]$ y el menor de la derecha el $a[j]$ (siempre que ambos segmentos cuenten con elementos). Una vez alguno de los dos segmentos se haya acabado, podemos proceder a rellenar con los elementos del otro segmento.

Podemos establecer entonces mientras ambos segmentos cuenten aún con elementos la siguiente iteración:

$$\left\{\begin{array}{ll} \textbf{Inv:} & izq \leqslant i \leqslant med \leqslant j \leqslant der \ \wedge \ k = (j - med) + (i - izq) \\ & \wedge \ bag.a[izq..i) \uplus bag.a[med..j) = bag.b[0..k) \\ & \wedge \ bag.a[0..n) = D \ \wedge \ ord.b[0..k) \\ \textbf{Cota:} & (der - izq) - k \end{array}\right\}$$

$$\begin{array}{l} \underline{\textbf{do}}\ i \neq med \ \wedge \ j \neq der \longrightarrow \\ \quad \underline{\textbf{if}}\ a[i] \sqsubseteq a[j] \longrightarrow \\ \quad\quad b[k], i := a[i], i + 1 \\ \quad [\,]\ a[j] \sqsupset a[i] \longrightarrow \\ \quad\quad b[k], j := a[j], j + 1 \\ \quad \underline{\textbf{fi}} \\ \quad ; k := k + 1 \\ \underline{\textbf{od}} \end{array}$$

Luego debemos proceder a rellenar los elementos restantes con aquellos del segmento que aún no se encuentre vacío. Estableceremos para ello lo siguiente:

$$
\begin{array}{l}
\underline{\textbf{if}}\ i \neq med\ \wedge\ j = der \longrightarrow \\
\quad \left\{ \begin{array}{ll} \textbf{Inv:} & izq \leqslant i \leqslant med\ \wedge\ k = (der - med) + (i - izq) \\ & \wedge\ bag.a[izq..i) \uplus bag.a[med..der) = bag.b[0..k) \\ & \wedge\ bag.a[0..n) = D\ \wedge\ ord.b[0..k) \\ \textbf{Cota:} & med - i \end{array} \right\} \\
\quad \underline{\textbf{do}}\ i \neq med \longrightarrow \\
\quad\quad b[k], i := a[i], i + 1; \\
\quad\quad k := k + 1 \\
\quad \underline{\textbf{od}} \\
[]\ i = med\ \wedge\ j \neq der \longrightarrow \\
\quad \left\{ \begin{array}{ll} \textbf{Inv:} & med \leqslant j \leqslant der\ \wedge\ k = (der - j) + (med - izq) \\ & \wedge\ bag.a[izq..med) \uplus bag.a[med..j) = bag.b[0..k) \\ & \wedge\ bag.a[0..n) = D\ \wedge\ ord.b[0..k) \\ \textbf{Cota:} & der - j \end{array} \right\} \\
\quad \underline{\textbf{do}}\ j \neq der \longrightarrow \\
\quad\quad b[k], j := a[j], j + 1; \\
\quad\quad k := k + 1 \\
\quad \underline{\textbf{od}} \\
\underline{\textbf{fi}}
\end{array}
$$

Fácilmente podemos ver que los condicionales no son necesarios puesto que las mismas guardas de los ciclos demarcan en cuál de los dos se debe entrar. Si ya se transcribieron todos los elementos del primer segmento entonces $i = med$ y la guarda del primer ciclo no se cumpliría, análogamente con el segundo.

Sólo nos queda transcribir el arreglo ordenado al segmento original $[izq..der)$. Para ello utilizaremos la misma variable k y simplemente iremos iterando sobre el arreglo auxiliar mientras se realizan las respectivas asignaciones.

$$
\begin{array}{l}
k := 0; \\
\left\{ \begin{array}{ll} \textbf{Inv:} & 0 \leqslant k \leqslant der - izq\ \wedge\ a[izq..izq + k) = b[0..k) \\ \textbf{Cota:} & der - izq - k \end{array} \right\} \\
\underline{\textbf{do}}\ k < der - izq \longrightarrow \\
\quad a[izq + k] := b[k]; \\
\quad k := k + 1 \\
\underline{\textbf{od}}
\end{array}
$$

Finalmente contamos con la siguiente implementación para el procedimiento *mezclar*:

proc mezclar(**in** n: *int*; **in-out** a: *array[0..n) of* T;
in izq, med, der: *int*)

$$\left\{ \begin{array}{ll} \textbf{Pre:} & 0 \leqslant izq < med < der \leqslant n \ \wedge \ bag.a[0..n) = D \\ & \wedge \ a[0..izq) = S_0 \ \wedge \ a[der..n) = S_1 \end{array} \right\}$$

$$\left\{ \begin{array}{ll} \textbf{Post:} & bag.a[0..n) = D \ \wedge \ ord.a[izq..der) \\ & \wedge \ a[0..izq) = S_0 \ \wedge \ a[der..n) = S_1 \end{array} \right\}$$

|[

var i, j, k: *int*;
var b: *array[0..(der-izq+1)] of* T;
$i,\ j,\ k := izq,\ med,\ 0;$

$$\left\{ \begin{array}{ll} \textbf{Inv:} & izq \leqslant i \leqslant med \leqslant j \leqslant der \ \wedge \ k = (j - med) + (i - izq) \\ & \wedge \ bag.a[izq..i) \uplus bag.a[med..j) = bag.b[0..k) \\ & \wedge \ bag.a[0..n) = D \ \wedge \ ord.b[0..k) \\ \textbf{Cota:} & (der - izq) - k \end{array} \right\}$$

do $i \neq med \ \wedge \ j \neq der \longrightarrow$

if $a[i] \sqsubseteq a[j] \longrightarrow$
$\quad b[k], i := a[i], i + 1$
[] $a[j] \sqsupset a[i] \longrightarrow$
$\quad b[k], j := a[j], j + 1$
fi

$; k := k + 1$

od;

$$\left\{ \begin{array}{ll} \textbf{Inv:} & izq \leqslant i \leqslant med \leqslant j \leqslant der \\ & \wedge \ k = (j - med) + (i - izq) \\ & \wedge \ bag.a[izq..i) \uplus bag.a[med..j) = bag.b[0..k) \\ & \wedge \ bag.a[0..n) = D \ \wedge \ ord.b[0..k) \\ \textbf{Cota:} & med - i \end{array} \right\}$$

do $i \neq med \longrightarrow$
$\quad b[k], i := a[i], i + 1;$
$\quad k := k + 1$
od

$$
\left\{
\begin{array}{ll}
\textbf{Inv:} & izq \leqslant i \leqslant med \leqslant j \leqslant der \\
& \wedge\ k = (der - j) + (i - izq) \\
& \wedge\ bag.a[izq..i) \uplus bag.a[med..j) = bag.b[0..k) \\
& \wedge\ bag.a[0..n) = D\ \wedge\ ord.b[0..k) \\
\textbf{Cota:} & der - j
\end{array}
\right\}
$$

do $j \neq der \longrightarrow$
$\quad b[k], j := a[j], j + 1;$
$\quad k := k + 1$
od

$k := 0;$

$$
\left\{
\begin{array}{ll}
\textbf{Inv:} & 0 \leqslant k \leqslant der - izq\ \wedge\ a[izq..izq + k) = b[0..k) \\
\textbf{Cota:} & der - izq - k
\end{array}
\right\}
$$

do $k < der - izq \longrightarrow$
$\quad a[izq + k] := b[k];$
$\quad k := k + 1$
od

]||

Este algoritmo presenta la leve desventaja de que necesita memoria extra para ejecutarse puesto que mezclar ambos segmentos requiere de la creación de un arreglo temporal.

9. Quicksort

El Ordenamiento Rápido (*Quicksort*) fue desarrollado por C. A. R. Hoare en el año 1962, se trata igualmente de un algoritmo que utiliza la técnica divide y vencerás.

Para la descomposición, el algoritmo considera un elemento pivote. Ésta se realiza de tal forma que todos los elementos menores o iguales al pivote quedan a su izquierda y los mayores a su derecha.

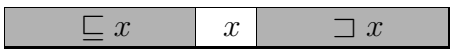
Dado que no estamos dividiendo el espacio a ordenar necesariamente en dos partes iguales o aproximadamente iguales, podría estar sucediento, en el peor de los casos, que el pivote sea el mayor elemento del segmento, con lo cual tendremos un subproblema prácticamente de igual tamaño que el problema inicial. Por lo tanto, es fundamental la escogencia de un buen pivote.

Una de las técnicas para la escogencia indica escoger tres elementos del arreglo y tomar el elemento medio, pero esto no aporta una mejora sustancial, puesto que podríamos haber seleccionado el segundo o penúltimo elemento en la secuencia a ordenar, que no difiere mucho de agarrar el primero o el último. Otra idea sería pasearnos cada vez por el arreglo para escoger el elemento medio, pero esto implicaría un cálculo costoso, por lo que afectaría en gran parte el rendimiento del algoritmo. Por lo tanto, podríamos de hecho escoger el primero, el último o cualquier elemento del segmento como pivote, y en promedio esto resultará una buena escogencia. Sin embargo, para casos específicos, en donde se tenga un preconocimiento acerca de la distribución de los elementos en el arreglo se podría escoger algún otro elemento específico del arreglo.

El procedimiento principal del algoritmo es semejante al ordenamiento por mezcla, sólo que altera el orden de las instrucciones. Igualmente los parámetros *izq* y *der* delimitan el espacio a ordenar.

proc ordenar(**in** n: *int*; **in-out** a: *array[0..n) of T*; **in** izq, der: *int*)
{ **Pre:** $0 \leqslant izq \leqslant der \leqslant n \ \wedge \ bag.a[izq..der) = C$
$\wedge \ a[0..izq) = A_0 \ \wedge \ a[der..n) = A_1$ }
{ **Post:** $ord.a[izq..der) \ \wedge \ bag.a[izq..der) = C$
$\wedge \ a[0..izq) = A_0 \ \wedge \ a[der..n) = A_1$ }
{ **Cota:** $der - izq$ }
|[
if $der - izq < 2 \longrightarrow$
skip
[] $der - izq \geq 2 \longrightarrow$
var *pivote*: *int*;
particionar$(n, a, izq, der, pivote)$;
ordenar$(n, a, izq, pivote)$;
ordenar$(n, a, pivote{+}1, der)$;
fi
]|

El procedimiento *particionar* es el encargado de distribuir los elementos del arreglo de tal forma que los elementos queden organizados de la siguiente manera:



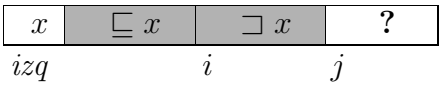
Para cumplir con esto, tenemos entonces la siguiente especificación para el problema:

$$\begin{array}{l}\textbf{proc}\ \text{particionar}(\underline{\textbf{in}}\ \text{n: } int;\ \underline{\textbf{in-out}}\ \text{a: } array[0..n)\ of\ T;\\ \qquad\qquad \underline{\textbf{in}}\ \text{izq, der: } int,\ \underline{\textbf{out}}\ \text{pivote: } int)\\ \left\{\begin{array}{ll}\textbf{Pre:} & 0 \leqslant izq < der \leqslant n\ \wedge\ bag.a[0..n) = D\\ & \wedge\ a[0..izq) = S_0\ \wedge\ a[der..n) = S_1\end{array}\right\}\\ \left\{\begin{array}{ll}\textbf{Post:} & bag.a[0..n) = D\ \wedge\ a[0..izq) = S_0\ \wedge\ a[der..n) = S_1\\ & \wedge\ izq \leqslant pivote < der\\ & \wedge\ a[izq..pivote) \sqsubseteq a[pivote] \sqsubset a[pivote+1..der)\end{array}\right\}\end{array}$$

Digamos que nuestro pivote, x, es el primer elemento del segmento por particionar. Al comienzo del procedimiento tenemos lo siguiente:

Dado que estamos buscando separar los elementos entre los menores o iguales y los elementos mayores al pivote, consideraremos dos intervalos contiguos que contendrán estos elementos respectivamente. En un instante cualquiera del recorrido, tendremos que se ha de cumplir:

Estos intervalos estarán delimitados de la siguiente manera por las variables izq, i, j y der:



En la siguiente iteración, debemos decidir a qué intervalo pertenece $a[j]$, suponiendo que $a[j] \sqsubseteq x$, debemos colocarlo en el primer segmento. Para esto, intercambiaremos el elemento que ocupa actualmente la posición i del arreglo con $a[j]$, luego incrementaremos i y j. En caso contrario basta con incrementar j.

Una vez finalizado el recorrido, nuestro arreglo tendrá la siguiente apariencia:

Ya en este punto basta hacer un simple intercambio entre x y $a[i]$ para satisfacer la postcondición del procedimiento. Veamos finalmente la implementación del procedimiento *particionar*:

proc particionar(**in** n: *int*; **in-out** a: *array[0..n) of T*;
in izq, der: *int*, **out** pivote: *int*)

$$\left\{ \begin{array}{l} \textbf{Pre:} \quad 0 \leqslant izq < der \leqslant n \ \wedge \ bag.a[0..n) = D \\ \qquad\quad \wedge \ a[0..izq) = S_0 \ \wedge \ a[der..n) = S_1 \end{array} \right\}$$

$$\left\{ \begin{array}{l} \textbf{Post:} \quad bag.a[0..n) = D \ \wedge \ a[0..izq) = S_0 \ \wedge \ a[der..n) = S_1 \\ \qquad\quad \wedge \ izq \leqslant pivote < der \\ \qquad\quad \wedge \ a[izq..pivote) \sqsubseteq a[pivote] \sqsubset a[pivote+1..der) \end{array} \right\}$$

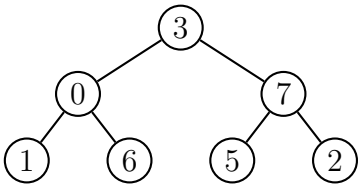
```
|[
  var i, j: int;

  i, j := izq + 1, izq + 1;

  { Inv:  izq < i ⩽ j ⩽ der ∧ a[izq + 1..i) ⊑ a[izq] ⊏ a[i..j)
          ∧ bag.a[izq..der) = C
    Cota: der − j }
  do j ≠ der ⟶
     if a[j] ⊑ a[izq] ⟶
        a[i], a[j] := a[j], a[i];
        i := i + 1
     [] a[j] ⊐ a[izq] ⟶
        skip
     fi
     ; j := j + 1
  od

  ; a[izq], a[i − 1] := a[i − 1], a[izq]
  ; pivote := i − 1
]|
```

10. Heapsort

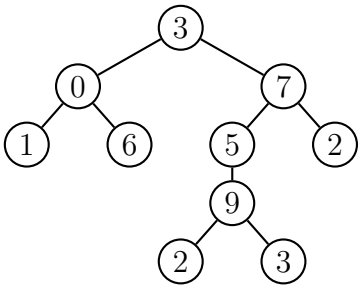
El algoritmo Heapsort o de *Ordenamiento por Montículos* es un algoritmo de complejidad $\Theta(n \cdot lg\, n)$. En su desarrollo involucra una estructura de datos que se denomina *heaps* o *montículos*.

10.1. Heaps

Un *heap* es una estructura de datos que puede ser vista como un árbol binario, donde cada elemento tiene hasta dos hijos asociados y que adicionalmente cumple con otras restricciones. Veamos un ejemplo de árbol binario:



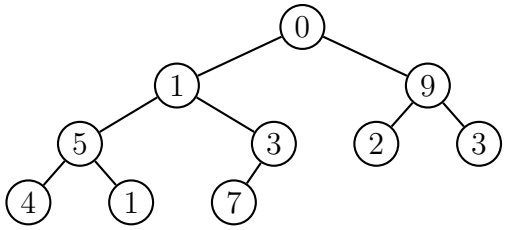
La primera restricción que impondremos será que cada nivel debe tener tantos elementos como sea posible. Por ejemplo, en el árbol anterior podemos ver que cada nivel se encuentra copado en todas sus posiciones válidas. Un árbol que no satisfaga esa restricción sería por ejemplo el siguiente:



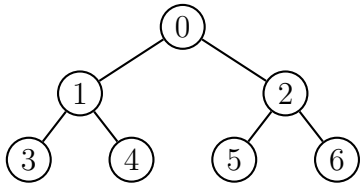
No cumple pues el cuarto y quinto nivel no se encuentran completos. Si somos estrictos en esto, podríamos tener heaps con un número restringido de vértices. Se deja como ejercicio al lector calcular la cantidad de posibles nodos de un árbol estrictamente completo.

Relajaremos entonces un poco esta condición de tal forma que todos los niveles se encuentren completamente llenos, a excepción del último, el cual

podrá tener ciertos espacios vacíos. ¿Qué espacios deben ocupar los elementos del último nivel? Debemos ir llenando de izquierda a derecha, es decir, los espacios que posiblemente quedasen vacíos estarían situados en la parte derecha del último nivel. Veamos por ejemplo el siguiente árbol de tamaño 10:



Es posible enumerar los elementos pertenecientes a un cierto árbol por niveles; es decir, el elemento raíz ocupará la posición 0, sus hijos ocuparán las posiciones 1 y 2 respectivamente, luego los elementos en el tercer nivel llevarán las etiquetas 3, 4, 5 y 6 y así sucesivamente. Veamos por ejemplo la numeración (representada con el valor correspondiente en cada círculo) del primer árbol que manejamos.



El elemento 1 por ejemplo, tiene como hijos los nodos etiquetados como 3 y 4 o también $2 \cdot 1 + 1$ y $2 \cdot 1 + 2$; el cero tiene a su vez como hijos los elementos $1 = 2 \cdot 0 + 1$ y $2 = 2 \cdot 0 + 2$. Se puede verificar que, en caso de existir, los hijos del elemento que alberga la posición i son el $2i+1$ y el $2i+2$.

Definiremos las funciones *hijoIzq* e *hijoDer* que determinarán la posición en el árbol del hijo izquierdo y derecho de un cierto elemento:

$$hijoIzq : \mathbb{N} \longrightarrow \mathbb{N} \qquad hijoDer : \mathbb{N} \longrightarrow \mathbb{N}$$
$$hijoIzq(i) = 2i + 1 \qquad hijoDer(i) = 2i + 2$$

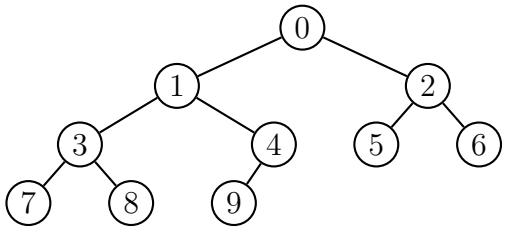
El valor obtenido al aplicar cada una de estas funciones a un índice i será congruente con nuestro árbol si y sólo si el hijo, izquierdo o derecho según sea el caso, del elemento i existe.

¿Bajo que condiciones se puede decir que los hijos de un árbol existen? Para esto definiremos el tamaño de un *Heap* (*heapSize*) como el número de elementos que lo integran. Por ejemplo, el heap pasado era de tamaño 7.

Podemos asegurar que un cierto hijo del elemento i existe si se encuentra en el intervalo correspondiente a las posibles posiciones que ocupan los elementos en el *heap*, es decir,

$$0 \leqslant hijoIzq(i), hijoDer(i) < heapSize$$

esta notación ha sido consistente con nuestro árbol completo que presentamos anteriormente, pero, ¿será también consistente si el árbol no se encuentra lleno en su último nivel? Veamos por ejemplo qué sucede con el árbol presentado anteriormente:



Al considerar el árbol completo asociado al eliminar los elementos 7, 8 y 9 tenemos que se cumple para todos los elementos en el intervalo [0..7).

Para el elemento 3, se tiene que sus hijos serían el $2 \cdot 3 + 1 = 7$ y $2 \cdot 3 + 2 = 8$, lo cual es cierto. En el caso del 4 tenemos que sus hijos serían respectivamente el 9 y 10, dado que el 10 es igual a *heapSize*, no existe; para los 5 y 6 se tiene que al aplicar las funciones a cada uno de estos el valor obtenido excede el rango de valores aceptados en el *heap*, por lo tanto, sus hijos no existen. Esto concuerda perfectamente con la noción que hemos presentado.

Hasta ahora hemos presentado esto de forma intuitiva, pero no hemos dado una demostración de que esto es siempre cierto. Daremos una demostración basándonos en las características que cumplen los árboles semejantes a los que hemos definido.

Cuando un árbol está balanceado, se pueden verificar cada uno de los siguientes items:

- Si un elemento ocupa la posición i en la numeración, este se encuentra en el nivel $\lfloor log_2(i+1) \rfloor$ del *heap*.
- El número de elementos del nivel m, si está completo es de 2^m.
- El elemento izquierdo del nivel m tiene índice $2^m - 1$.

Teniendo esto, podemos verificar que el nodo i tiene a su izquierda un total de

$$i - (2^{\lfloor log_2(i+1) \rfloor} - 1)$$

elementos y tiene a su derecha

$$\begin{array}{ll} & 2^{\lfloor log_2(i+1) \rfloor} - 1 - (i - (2^{\lfloor log_2(i+1) \rfloor} - 1)) \\ = & 2^{\lfloor log_2(i+1) \rfloor} - 1 - i + 2^{\lfloor log_2(i+1) \rfloor} - 1 \\ = & 2 \cdot 2^{\lfloor log_2(i+1) \rfloor} - i - 2 \end{array}$$

elementos. Cada uno de los elementos a la izquierda aportarán dos unidades a la posición del hijo izquierdo de i, y los elementos a su derecha aportarán una unidad. Adicionalmente, el elemento i estará aportando con una unidad más. Por lo tanto, el hijo izquierdo del elemento i se encuentra desplazado con respecto a su padre en:

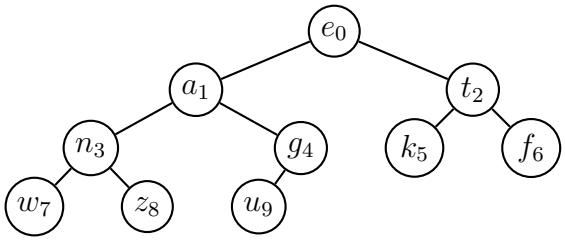
$$\begin{array}{ll} & 2 \cdot (i - (2^{\lfloor log_2(i+1) \rfloor} - 1)) + 2 \cdot 2^{\lfloor log_2(i+1) \rfloor} - i - 2 + 1 \\ = & 2i - 2 \cdot 2^{\lfloor log_2(i+1) \rfloor} + 2 + 2 \cdot 2^{\lfloor log_2(i+1) \rfloor} - i - 2 + 1 \\ = & i + 1 \end{array}$$

Luego, si el elemento se encuentra en la posición i, su hijo izquierdo se encontrará en la $i + i + 1 = 2i + 1$.

Para el cálculo de la posición del hijo derecho basta con desplazar en una unidad al hijo izquierdo correspondiente a este último. Es decir, el hijo derecho se encontrará en la posición $2i + 2$.

Es importante hacer notar que la demostración anterior tiene sentido siempre y cuando el hijo izquierdo/derecho exista, en caso contrario no tiene sentido hablar de la posición que ocupa alguno de sus hijos en el *heap*. Se deja como ejercicio al lector demostrar las tres propiedades enunciadas anteriormente: nivel de un nodo en el *heap*, número de elementos por nivel y elemento izquierdo de un nivel.

Recordemos que estamos buscando con todo esto implementar un algoritmo de ordenamiento sobre arreglos. Sin embargo, nótese que podemos llevar fácilmente un *heap* a un arreglo con la numeración que realizamos; simplemente tenemos que colocar cada nodo i del *heap* en la casilla número i del arreglo. De esta forma se seguirá manteniendo la relación padre-hijos que hemos indicado anteriormente. Por ejemplo, un ejemplo de *heap* y su arreglo asociado se encuentra a continuación; hemos colocado como subíndice de cada elemento su etiqueta asignada en la numeración para mayor claridad:



e	a	t	n	g	k	f	w	z	u

¿De qué tamaño debe ser el arreglo? No es necesario ser demasiado estrictos, exigiendo que el arreglo deba ser de tamaño igual al *heap*. Utilizaremos como convención que el arreglo puede ser de cualquier tamaño (n) que no cause conflictos con el del *heap*, esto es

$$0 \leqslant heapSize \leqslant n$$

Y que los elementos del heap se encontrarán en el intervalo de casillas [0..*heapSize*) del arreglo. Por lo tanto, podríamos haber colocado el *heap* anterior en el siguiente arreglo:

e	a	t	n	g	k	f	w	z	u	...	
0	1	2	3	4	5	6	7	8	9	...	n-1

Desde este momento nos referiremos comúnmente con la notación $a[i]$ como el nodo del *heap* almacenado en la casilla i del arreglo a.

10.2. Predicados Auxiliares

Teniendo n: *int* y a: *array[0..n) of T*, definimos para l, m: *int* tales que $0 \leqslant l \leqslant m \leqslant n$ el predicado

$$okHeap(a, l, m) \equiv (\forall i, j \mid l \leqslant i, j < m \ \wedge \ i \ esDesc \ j : a[i] \sqsubseteq a[j])$$

y también definimos para l, m, exc: int tales que $0 \leqslant l \leqslant exc < m \leqslant n$ el predicado

$$casiHeap(a, l, m, exc) \equiv (\forall i, j \mid l \leqslant i, j < m \wedge i \ esDesc \ j \wedge j \neq exc : a[i] \sqsubseteq a[j])$$

donde

$$esDesc = (esHijo)^+$$

es decir, la relación *esDesc* viene siendo la clausura transitiva de la relación *esHijo*, donde

$$x \ esHijo \ y \equiv (x = 2y + 1) \ \vee \ (x = 2y + 2)$$

lo cual es equivalente a

$$x \ esHijo \ y \equiv y = (x - 1) \ div \ 2$$

De esta forma *okHeap* indica que el segmento $[l..m)$ de a está adecuadamente organizado como un *heap*, y decimos que un *casiHeap* tiene posiblemente un nodo "problemático" *exc* que le impide ser *heap*.

Tal como hemos definido el predicado *okHeap*, nos estamos refiriendo a *max-Heaps*, donde el progenitor es mayor que sus descendientes. Sin embargo, es importante saber que es posible definir la propiedad *okHeap* de forma análoga diciendo que los descendientes son mayores a los progenitores, refiriendonos entonces a *min-Heaps*. Sin embargo, para el *Ordenamiento Heapsort* estaremos utilizando *max-Heaps* y nos referiremos a ellos simplemente como *heaps*.

Un aspecto importante de usar *heaps* para ordenar arreglos, es que en un segmento no-vacío de arreglo, si éste aloja un heap enraizado en 0, tal posición 0 contiene el máximo:

$$x > 0 \ \wedge \ okHeap(a, 0, x) \Rightarrow (\forall i \mid 0 \leqslant i < x : a[0] \sqsupseteq a[i])$$

10.3. Procedimientos

10.3.1. acomodarHeap

Este procedimiento permitirá acomodar un *casiHeap* para que satisfaga el predicado *okHeap*. Veamos la firma del procedimiento *acomodarHeap*:

proc acomodarHeap (**in** n: *int*; **in-out** a: *array[0..n) of T*; **in** l, m: *int*)

$$\left\{\begin{array}{ll}\textbf{Pre:} & 0 \leqslant l < m \leqslant n \ \wedge \ casiHeap(a,l,m,l) \\ & \wedge\ bag.a[l..m) = B \ \wedge \ a[0..l) = A_0 \ \wedge \ a[m..n) = A_1\end{array}\right\}$$

$$\left\{\begin{array}{ll}\textbf{Post:} & okHeap(a,l,m) \\ & \wedge\ bag.a[l..m) = B \ \wedge \ a[0..l) = A_0 \ \wedge \ a[m..n) = A_1\end{array}\right\}$$

Las reorganizaciones de *heaps* conviene iniciarlas por la raíz, y es lo que se acostumbra hacer. Atacaremos el problema de acomodar el *heap* de esta manera gracias a la propiedad siguiente:

$$okHeap(a, x+1, y) \equiv casiHeap(a, x, y, x)$$

Dada nuestra precondición, se tiene que $casiHeap(a, l, m, l)$; por lo tanto, tenemos que el intervalo $[l+1, m)$ del arreglo representa un *heap*.

Adicionalmente, se cumple que los "casi-Heaps" terminan de acomodarse cuando localmente se satisfacen las condiciones de orden requeridas, entendiendo por *localmente* que basta que los hijos directos ya estén bien ordenados (si tales hijos existen dentro del rango en consideración):

$$\left(\begin{array}{l} casiHeap(a,x,y,z) \\ \wedge\ (2z+1 < y \Rightarrow a[2z+1] \sqsubseteq a[z]) \\ \wedge\ (2z+2 < y \Rightarrow a[2z+2] \sqsubseteq a[z]) \\ \ \end{array}\right) \equiv okHeap(a,x,y)$$

Teniendo esto, podemos proceder a arreglar la condición de orden requerida de forma local entre el nodo problemático y sus hijos. Así tendremos que se cumple $casiHeap(a, x, y, hijo(z))$ donde $hijo(z)$ viene siendo el descendiente directo del nodo z al cual pasamos el problema. Procediendo de igual manera, podemos descender verticalmente por el *heap* hasta solventar la situación. Esto se da cuando localmente el nodo satisfaga los requerimientos de orden, gracias a la propiedad anterior, o igualmente, cuando éste no tenga hijos gracias a que en rangos vacíos se cumple trivialmente la propiedad de ser heap:

$$okHeap(a, x, x)$$

Para verificar que localmente un nodo satisface el orden, basta con verificar que este sea el mayor entre él y sus hijos. Veamos cómo podemos verificar esto y, en caso de no satisfacerse el orden cómo podemos arreglarlo. Sea k el posible nodo problemático y m el límite superior del segmento que contiene al *"casi-Heap"* y *listo* una variable booleana que indica si se satisface localmente el orden. Para la verificación definiremos la siguiente notación equivalente para simplificar las guardas de los condicionales:

$$\begin{array}{l} \underline{\textbf{if}}\ B_0 \longrightarrow \\ \quad\ \ I_0 \\ []\ \neg B_0 \longrightarrow \\ \quad\ \ I_1 \\ \underline{\textbf{fi}} \end{array} \quad = \quad \begin{array}{l} \underline{\textbf{if}}\ B_0\ \textbf{then} \\ \quad\ \ I_0 \\ \underline{\textbf{else}} \\ \quad\ \ I_1 \\ \underline{\textbf{fi}} \end{array}$$

Tenemos entonces lo siguiente:

```
var mayor: int;
mayor := k;

if (2k + 1 < m ∧ a[2k + 1] ⊐ a[mayor]) then
|    mayor := 2k + 1
else
|    skip
fi;

if (2k + 2 < m ∧ a[2k + 2] ⊐ a[mayor]) then
|    mayor := 2k + 2
else
|    skip
fi;

if (mayor = k) then
|    listo := true
else
| a[mayor], a[k] := a[k], a[mayor]
| ; k :=mayor
fi
```

Teniendo esto, si *listo* es falso, tenemos en k el nuevo nodo problemático. Veamos finalmente la implementación del procedimiento *acomodarHeap*:

proc acomodarHeap (**in** n: *int*; **in-out** a: *array[0..n) of T*; **in** l, m: *int*)

$$\left\{\begin{array}{ll}\textbf{Pre:} & 0 \leqslant l < m \leqslant n \ \wedge \ casiHeap(a,l,m,l) \\ & \wedge\ bag.a[l..m) = B \ \wedge \ a[0..l) = A_0 \ \wedge \ a[m..n) = A_1\end{array}\right\}$$

$$\left\{\begin{array}{ll}\textbf{Post:} & okHeap(a,l,m) \\ & \wedge\ bag.a[l..m) = B \ \wedge \ a[0..l) = A_0 \ \wedge \ a[m..n) = A_1\end{array}\right\}$$

|[

var k: *int*;
$listo$: *boolean*;
$k, listo := l, false$;

$$\left\{\begin{array}{ll}\textbf{Inv:} & l \leqslant k < m \ \wedge \ casiHeap(a,l,m,k) \\ & \wedge\ listo \Rightarrow \left(\begin{array}{l}(2k+1 < m \Rightarrow a[2k+1] \sqsubseteq a[k]) \\ \wedge\ (2k+2 < m \Rightarrow a[2k+2] \sqsubseteq a[k])\end{array}\right) \\ & \wedge\ bag.a[l..m) = B \ \wedge \ a[0..l) = A_0 \ \wedge \ a[m..n) = A_1 \\ \textbf{Cota:} & (m-k) + (1 - \#listo)\end{array}\right\}$$

do $\neg listo \longrightarrow$

|[

var $mayor$: *int*;
$mayor := k$;

if $(2k+1 < m \ \wedge \ a[2k+1] \sqsupset a[mayor])$ **then**
  $mayor := 2k+1$
else
  skip
fi;
if $(2k+2 < m \ \wedge \ a[2k+2] \sqsupset a[mayor])$ **then**
  $mayor := 2k+2$
else
  skip
fi;
if $(mayor = k)$ **then**
  $listo := true$
else
  $a[mayor], a[k] := a[k], a[mayor]$
  $; k :=mayor$
fi

]|

od

]|

10.3.2. construirHeap

Hasta ahora hemos visto como convertir un *casiHeap* en un *heap* mediante el procedimiento anterior. Sin embargo, para eso necesitamos tener un *heap* y, hasta los momentos no tenemos las herramientas necesarias para construirlo. Gracias a la propiedad antes citada que dice

$$okHeap(a, x+1, y) \equiv casiHeap(a, x, y, x)$$

tenemos una forma inductiva de consturir *heaps*:

1. Tomar un segmento que trivialmente constituya un *heap*, podría ser por ejemplo el segmento vacío $[n..n)$ del arreglo a ordenar.

2. Adicionar al intervalo un nuevo elemento con lo cual tendremos un *casiHeap*.

3. Finalmente llamar a *acomodarHeap* para así tener un *heap* de tamaño mayor al inicial en uno.

De esta manera, iterando en los pasos 2 y 3 podemos agregar todos los elementos del arreglo a ordenar a un *heap*.

Veamos la siguiente implementación para construir el heap:

proc construirHeap (**in** n: *int*; **in-out** a: *array[0..n) of T*)
{ **Pre:** $n \geq 0 \ \wedge \ bag.a[0..n) = B$ }
{ **Post:** $okHeap(a, 0, n) \ \wedge \ bag.a[0..n) = B$ }
|[
var k: *int*;
$k :=$? ;

{ **Inv:** $0 \leqslant k \leqslant n \ \wedge \ okHeap(a, k, n) \ \wedge \ bag.a[0..n) = B$
Cota: k }
do $k \neq 0 \longrightarrow$
$k := k - 1$;
$acomodarHeap(n, a, k, n)$
od
]|

Nótese que dejamos la inicialización de k aún sin definir. Anteriormente dijimos que debíamos empezar con un segmento que trivialmente constituya

un *heap*. Podríamos simplemente inicializar k en n. Sin embargo, dado que en rangos no necesariamente vacíos se tiene un *heap* si en el rango considerado nadie es hijo de nadie:

$$okHeap(a, x, y) \Leftarrow x \geq \left\lfloor \frac{y}{2} \right\rfloor$$

podemos inicializar k en n *div* 2.

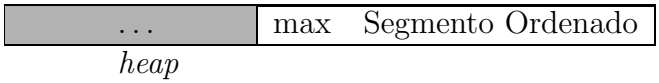
10.3.3. Heapsort

Finalmente podemos construir el ordenamiento. Para esto, debemos recordar que el mayor elemento de un *heap* se encuentra en su raíz, por lo tanto, para ordenar un arreglo basta:

1. Construir un *heap* con todos sus elementos.
2. Intercambiar la raíz del *heap* con el último elemento de este.
3. Decrementar el tamaño del *heap*.
4. Acomodar el *heap* resultante.

De esta forma, en cualquier instante justo antes de comenzar la iteración tendríamos un arreglo como el siguiente:

donde *max* es el mayor elemento del *heap* que pasará a estar en la siguiente posición:



formando parte del segmento ordenado.

Veamos la implementración final del *Ordenamiento Heapsort*:

proc heapSort(**in** n: *int*; **in-out** a: *array[0..n) of T*)
$\{$ **Pre:** $n \geq 0 \ \wedge \ bag.a[0..n) = B$ $\}$
$\{$ **Post:** $ord.a[0..n) \ \wedge \ bag.a[0..n) = B$ $\}$
|[
var k: *int*;
$construirHeap(n, a)$;
$k := n$;

$$\left\{ \begin{array}{ll} \textbf{Inv:} & 0 \leqslant k \leqslant n \ \wedge \ okHeap(a, 0, k) \ \wedge \ ord.a[k..n) \\ & \wedge \ a[0..k) \sqsubseteq a[k..n) \ \wedge \ bag.a[0..n) = B \\ \textbf{Cota:} & k \end{array} \right\}$$

do $k \geq 0 \longrightarrow$
$k := k - 1$
$; a[0], a[k] := a[k], a[0]$
$;\ acomodarHeap(n, a, 0, k)$
od
]|

11. Ejercicios

1. Especificar e implementar una versión de los ordenamientos por inserción, selección y burbuja tal que los elementos se vayan ordenando desde las posiciones más altas del arreglo. Es decir, al final de una iteración k, el segmento ordenado debe ser el $[k..n)$.

2. Implementar una versión de burbuja bidireccional.

3. Realizar una corrida en frío de cada uno de los algoritmos expuestos para el arreglo:

   1, 6, 3, 2, 8, 7, 4, 7, 3, 8, 10, 0, 3, 0, 9

   Determinar el número de iteraciones/recursiones para cada uno.

4. Suponga que desea calcular la moda entre los elementos de un arreglo, proponga una solución al problema que utilice algoritmos de ordenamiento. Se define la moda de un multiconjunto de elementos como el valor que más se repite, esto es, que no existe otro valor con más ocurrencias que él, nótese que más de un elemento puede ser la moda de un multiconjunto.

5. Suponga que usted posee un lunes todos los ciudadanos pertenecientes al registro electoral de una nación almacenados en un arreglo ordenados mediante su cédula de identidad. Al finalizar la semana se han agregado al final del arreglo todos los ciudadanos que se hayan inscrito en el transcurso de la semana. Se solicita su trabajo como Ingeniero de la Computación. ¿Qué algoritmo de ordenamiento escogería para realizar esta faena?. Suponga que la fracción de individuos añadidos en una semana es ínfima con respecto al total de personas que viven en el país.

6. Implemente una versión de quicksort donde se escoja como pivote el elemento medio entre tres elementos del segmento.

7. Dada la constante "oculta" en notación de complejidad asintótica, es común que para casos pequeños un algoritmo como Inserción se comporte mejor que otro asintóticamente más eficiente como mezcla o Quicksort. Modifique Quicksort para que cuando el tamaño del segmento sea menor o igual a k elementos se utilice Inserción como algoritmo de ordenamiento.

8. El problema de la *Bandera Holandesa* se define como:

   proc banderaHolandesa (
   in N: *int*;
   in-out a: *array[0..N) of [rojo,blanco,azul]*;
   in izq, der: *int*)

   $$\{\ \textbf{Pre:}\quad 0 \leqslant izq \leqslant der \leqslant N\ \wedge\ bag.a[0..N) = A\ \}$$

   $$\left\{\begin{array}{ll} \textbf{Post:} & bag.a[0..N) = A \\ & \wedge\ (\exists\ p, q, r\ |izq \leqslant p \leqslant q \leqslant der : \\ & \qquad (\forall i \mid izq \leqslant i < p : a[i] = rojo) \\ & \qquad \wedge\ (\forall i \mid p \leqslant i < q : a[i] = blanco) \\ & \qquad \wedge\ (\forall i \mid q \leqslant i < der : a[i] = azul) \end{array}\right\}$$

   Implemente este procedimiento utilizando un invariante de iteración semejante al del procedimiento de partición presentado en el algoritmo Quicksort. Desarrolle un invariante de iteración.

9. Indique cómo podría modificarse el problema de la bandera holandesa para que sea útil para el algoritmo Quicksort, realice las modificaciones necesarias e indique si en algún caso este algoritmo se estaría comportando mejor que el presentado anteriormente.

10. Implemente y especifique una versión del algoritmo de Heapsort que considere *min-Heaps*. Para esto, tiene que reespecificar el comportamiento de los heaps.

Referencias

[**Cormen et al. 09**] T.H. Cormen, C.E. Leiserson, R.L. Rivest & C. Stein, *Introduction to Algorithms*, The MIT Press, 3ra. edición, 2009.

[**Hume et al. 99**] J.N.P. Hume, T.L. West, R.C. Holt & D.T. Barnard, *Programming Data Structures in Java*, Holt Software Associates Inc. & The University of Toronto Press, 1999.

[**Kaldewaij 90**] A. Kaldewaij, *Programming: The Derivation of Algorithms*, Prentice Hall, 1990.

[**Gries 81**] David Gries, *The Science of Programming*, Springer-Verlag, 1981.

[**Backhouse 03**] Roland Backhouse, *Program Construction: Calculating Implementations from Specifications*, John Wiley & Sons, 2003.