Ayudantía 8 - Estructura de Datos y Algoritmos ELO320

Algoritmos de ordenamiento

Werner Creixell, Felipe Vera

6 de junio del 2013

1. Explicación

Ordenar un arreglo es un problema que, si se hiciera con el primer método que se nos viniera a la mente, no se lograría algo demasiado eficiente. Y es aquí cuando entra a la batalla el método *Divide et Impera*, repasado en la ayudantía pasada. Recordando el concepto aplicado al ordenamiento de arreglos, se sigue la siguiente estrategia:

- **Dividir**: Dependiendo del algoritmo de ordenamiento, el arreglo se divide de distintas maneras.
- **Resolver para elementos pequeños**: Las listas de 0 y 1 elementos ya se consideran ordenadas, las listas de 2 elementos pueden ordenarse, de modo que si no están ordenadas se pueden ordenar fácilmente intercambiando sus dos elementos.
- **Combinar**: Dependiendo del algoritmo, el ordenamiento se hace acá o se combinan las listas ya ordenadas de la manera adecuada.

Dichos algoritmos son de complejidad $\theta(n \log n)$ y están basados en comparaciones e intercambio de elementos. Estos algoritmos son:

- Merge Sort
- Heap Sort
- Quick Sort

2. Merge Sort

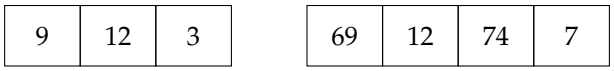
Es el ejemplo más fácil de comprender para entender el método *Divide et Impera*. Este método se basa en ORDENAMIENTO ASCENDENTE. Es decir, a medida que se empieza a combinar las listas luego de dividir, se comienza a ordenar las listas resultantes, asumiendo que se están combinando y ordenando dos listas ya ordenadas ascendentemente.

2.1. Ejemplo de funcionamiento

Si se tiene un arreglo de siete elementos...

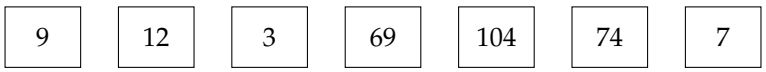
9	12	3	69	104	74	7

Se divide recursivamente en arreglos de la mitad de tamaño.



Merge Sort 1 Merge Sort 2

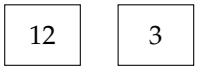
Hasta que se tienen listas pequeñas de un solo elemento cada una. (Se incluye también la recursión a la que corresponde cada una.



MS 1.1 MS 1.2.1 MS 1.2.2 MS 2.1.1 MS 2.1.2 MS 2.2.1 MS 2.2.2

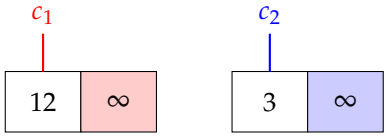
La recursión funciona de la siguiente manera:

- Se deben combinar el Merge Sort 1 y el Merge Sort 2. Como ninguno de los dos es una lista de un solo elemento, se aplica el Merge Sort a cada una de estas por separado.
- (Para el caso del Merge Sort 1) Se deben combinar el Merge Sort 1.1 y el 1.2. El Merge Sort 1.1 es una lista de un solo elemento, por lo que se considera ordenada. Sin embargo, el Merge Sort 1.2 se aplica a una lista de dos elementos. Se aplica Merge Sort a esta lista.
- (Para el caso del Merge Sort 1.2) Se divide en dos listas a las cuales se les aplica Merge Sort 1.2.1 y Merge Sort 1.2.2. Ambas son listas de un solo elemento, por lo que se procede a combinarlas.



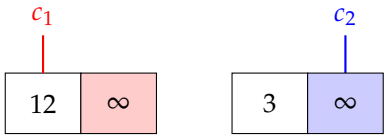
MS 1.2.1 MS 1.2.2

- Para combinarlas, se asume que ambas listas a combinar *(merge)* están ordenadas ascendentemente. Claramente esto se cumple para las listas de un solo elemento MS 1.2.1 y MS 1.2.2.

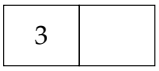


MS 1.2.1 MS 1.2.2

Como puede observarse se coloca un cursor c_1 y c_2 al principio de cada lista y un ∞ al final de cada una. [1]. Luego se comparan los elementos apuntados por c_1 y c_2. Como $A[c_2] = 3 < A[c_1] = 12$ se copia el elemento apuntado por c_2 (el 3) y este cursor avanza un espacio.



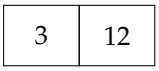
MS 1.2.1 MS 1.2.2



Merge Sort 1.2

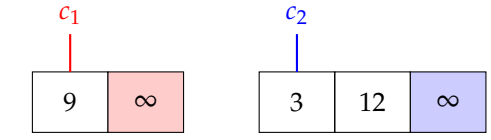
[1] El ∞ es algo conceptual que sólo sirve para indicar el momento en el cual se llega al final de la lista. En la implementación no se debe colocar, se debe saber cuándo se llega al final de una lista y copiar el resto de los elementos de la otra en orden.

Por último, ya se leyó toda la segunda lista, se copian los elementos de la primera luego del 3, es decir, se pone el 12 en el casillero en blanco, ordenando esta pequeña lista de dos elementos.



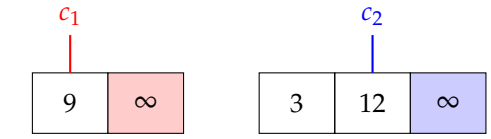
Merge Sort 1.2

- Como Merge Sort 1.1 y 1.2 son listas ya ordenadas, se procede a aplicar el mismo procedimiento para formar la lista ordenada que resulta de Merge Sort 1. Observa cómo en las siguientes imágenes se mueven los cursores a medida que se rellena la lista. **Eso se puede hacer porque se puede asumir con seguridad de que las listas a mezclar ya están ordenadas.**



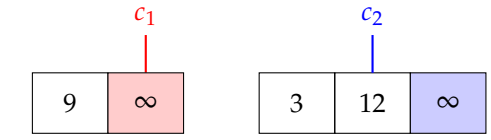
Merge Sort 1.1 Merge Sort 1.2

Merge Sort 1

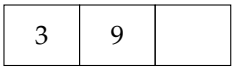


Merge Sort 1.1 Merge Sort 1.2

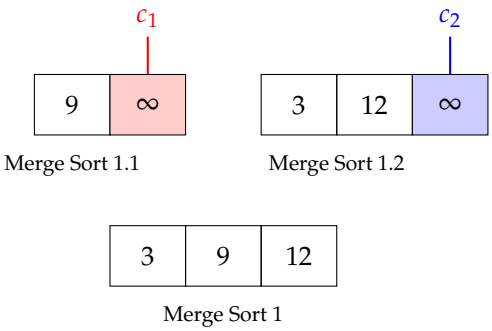
Merge Sort 1



Merge Sort 1.1 Merge Sort 1.2



Merge Sort 1



Y así sucesivamente hasta que toda la lista se encuentra ordenada.

2.2. Análisis de complejidad

- Cada recursión involucra dividir la lista en dos mitades. $2T(\frac{n}{2})$
- Al momento de combinar las listas se debe recorrer los elementos de ambas listas una sola vez. $\theta(n)$

Por lo tanto, la complejidad de este algoritmo corresponde a:

$$T(n) = 2T(\frac{n}{2}) + \theta(n)$$

Lo cual, mediante método maestro, puede concluirse que la complejidad del Merge Sort es de $T(n) = \theta(n \log(n))$

3. Heap Sort

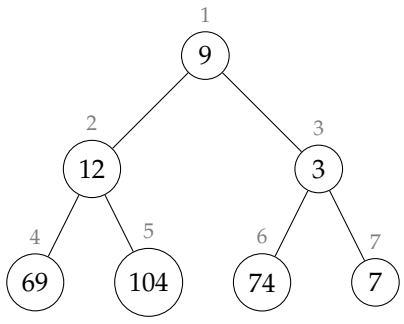
Este, a diferencia de Quick Sort y Merge Sort no se basa en recursividad para lograr una ejecución eficiente, sino que se basa en su estructura *heap*, para lograr un acceso más rápido a los elementos que se quieren ordenar.

Como ya se mencionó, se basa en una estructura llamada *max-heap*, similar a un árbol binario, pero no es más que una simple representación de un arreglo.

Se empleará el mismo arreglo para explicar el *Heap Sort*.

9	12	3	69	104	74	7

Se representa de esta manera para formar un *heap*.



Se puede observar una relación entre los índices de los padres y los hijos izquierdos. Esta es:

A partir de esto se debe construir un **Max-Heap** para comenzar a ordenar. El Heap-sort se basa en el ordenamiento usando este Max-Heap. LA REGLA del Max-Heap es la siguiente:

> **LA REGLA**
>
> El o los hijos de un nodo SIEMPRE deben ser menores a este.

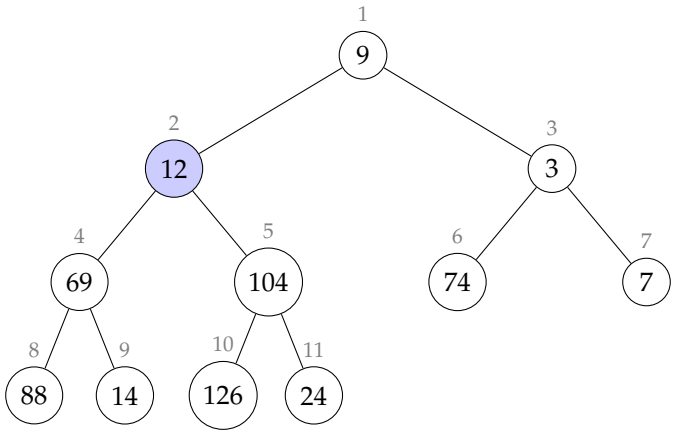
- Hijo izquierdo: $2n$
- Hijo derecho: $2n + 1$
- Padre: $\lfloor \frac{n}{2} \rfloor$

El método de operación para el Heap sort es el siguiente:

1. Convertir el *heap* al principio en un Max-Heap (rutina `max-heapify`), *hundiendo* los elementos desde la mitad del heap hasta el principio que no cumplan con LA REGLA. (rutina `sink-node`)
2. Luego, hasta que la lista se termine de ordenar:
   - Intercambiar el primer elemento del max-heap (el mayor) con el último elemento del arreglo.
   - *Hundir* el (ahora) primer elemento del max-heap para que cumpla LA REGLA (rutina `sink-node`).
   - Repetir el Heap Sort con la misma lista excepto el último elemento.

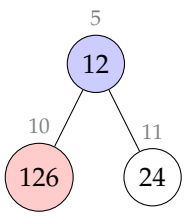
3.1. Rutina *sink-node*

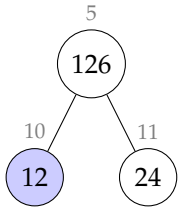
Para *hundir* un nodo, se verifica si cumple LA REGLA. En el siguiente ejemplo supongamos que queremos hundir el tercer elemento (un 12).



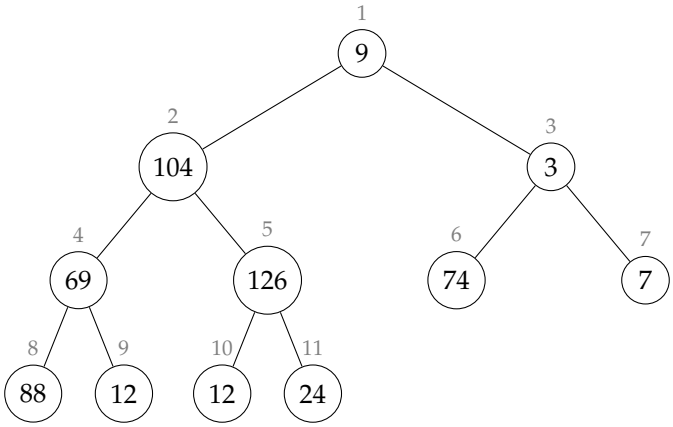
El subárbol entre el segundo, el cuarto y el quinto elemento claramente no cumplen LA REGLA. Por lo tanto se intercambia el 12 con su hijo mayor, el cual es el 104.

Y se repite el proceso hasta llegar al fondo del árbol o hasta que el elemento cumpla LA REGLA.



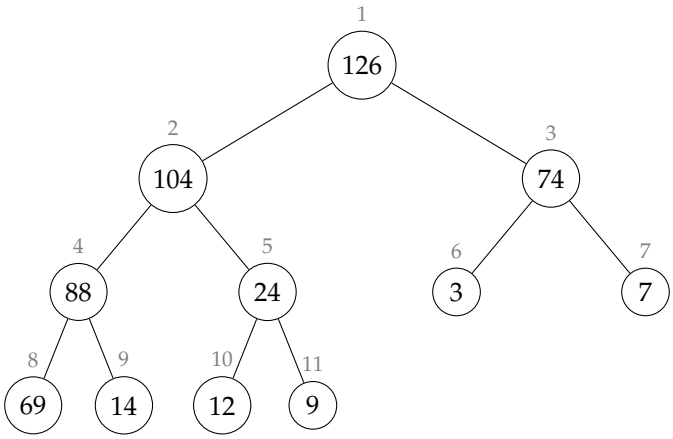


El *heap* debería quedar así:



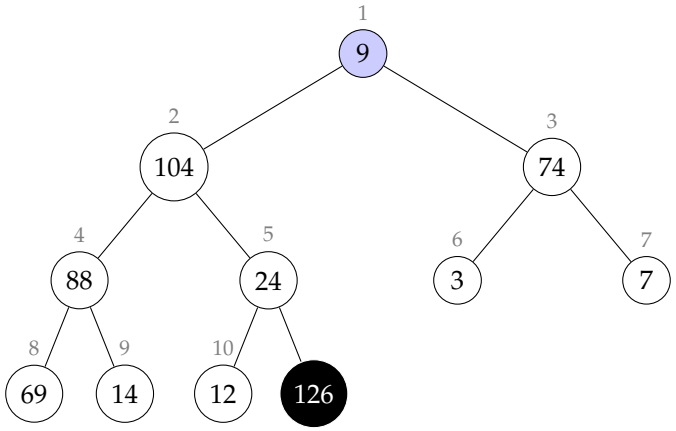
3.2. La rutina *max-heapify*

Con un solo `sink-node` no se puede ordenar toda la lista. Por eso se repite el `sink-node` para todos los nodos que tengan hijos, y se hunden, en este caso, desde el elemento 5 hasta el 1. Esto genera un max-heap, y en este caso se ve así:

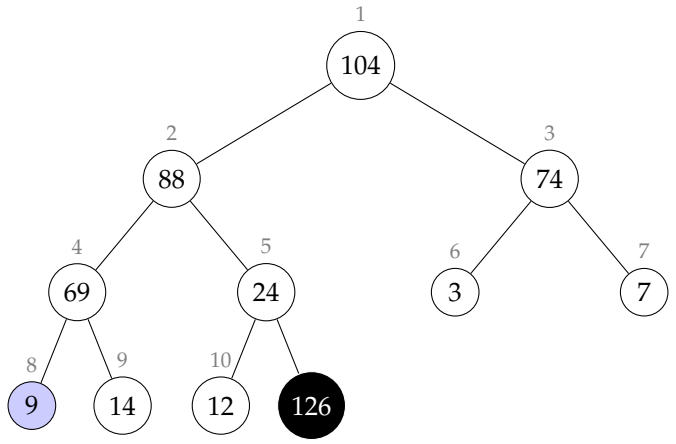


3.3. La rutina Heap-sort

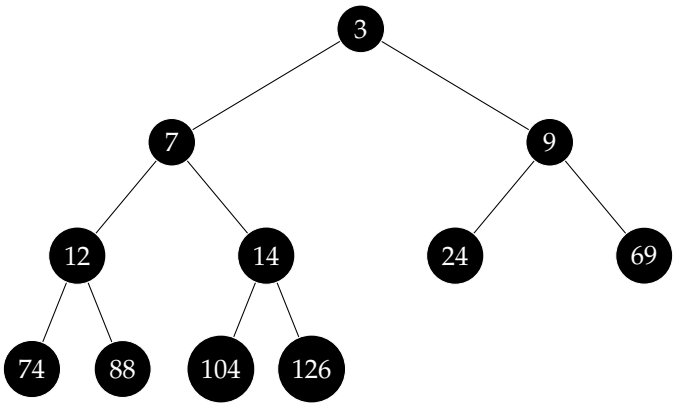
Luego de ordenar por primera vez el heap para tener un Max-heap que cumpla con LA REGLA, se intercambia el primer elemento por el último, y este último se "bloquea".



Tras bloquear el último elemento, se hunde el primero, obteniendo lo siguiente:



Y se repite el heap-sort hasta que se bloquean todos los elementos. Al final se debería obtener algo como lo siguiente, que al representarlo como un arreglo, se obtiene una lista ordenada.



3.4. Complejidad del Heap-sort

La complejidad del `sink-node` es de $\theta(\log n)$. Con ello se puede obtener la complejidad de `max-heapify` porque se repite $\frac{n}{2}$ veces la rutina `sink-node`, obteniendo que su compljidad es de $\theta(n \log n)$.

Luego de armar el max-heap se efectúa un `sink-node` n veces, teniendo otro algoritmo de complejidad $\theta(n \log n)$.

heap-sort se compone de ambas partes, por lo tanto:

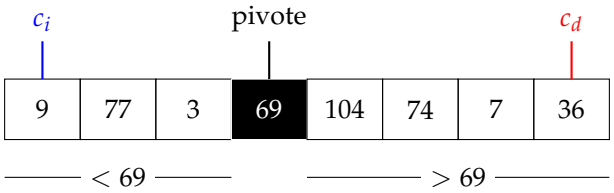
$$T(n) = \theta(n \log(n)) + \theta(n \log(n)) = \theta(n \log(n))$$

4. Quick Sort

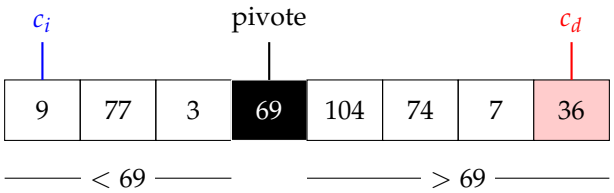
Este algoritmo es similar al merge-sort en el sentido de que se vale del método *Divide et Impera* para poder ordenar una lista, pero basándose en un precepto distinto:

- Se elige como pivote algún elemento (el Quick-sort se hace más eficiente cuando este pivote es la mediana de los datos a *quicksortear*).
- Se agrupan los elementos menores a este pivote a la izquierda y los mayores a la derecha del pivote, y se efectúa Quick-sort a estas dos listas más pequeñas, dejando el pivote temporalmente de lado y poniéndolo en medio de estas dos listas al momento de combinar y ordenar.

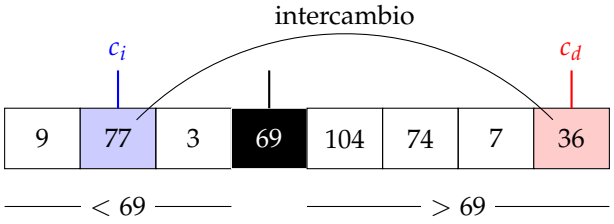
Tomemos como ejemplo esta lista. Se toma como pivote el 69.



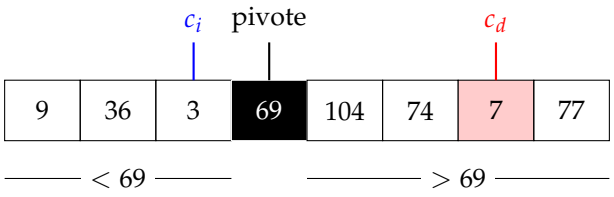
Se avanza primero desde el lado con más elementos, en este caso el derecho. En este caso se encuentra en el primer elemento el 36, el cual debería estar a la *izquierda* del 69. Se deja fijo el cursor c_d y se busca por la izquierda algún elemento mayor a 69.



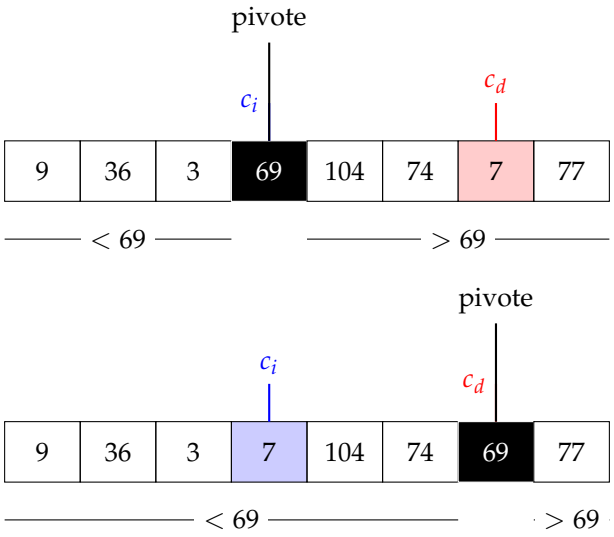
Como el 9 es menor, se deja intacto y se avanza el cursor c_i un espacio. Se encuentra el 77, el cual debería estar a la derecha, se intercambian y avanzan ambos cursores c_i y c_d.



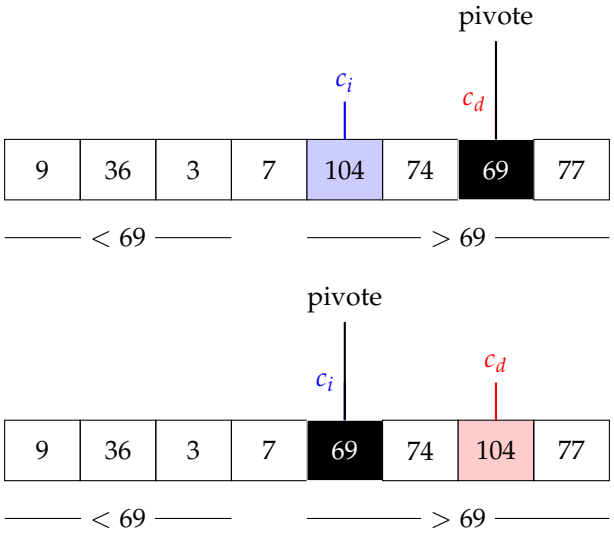
Luego de intercambiar, se encuentra desde la derecha un número menor al 69, así que se avanza por la izquierda para encontrar un número mayor.



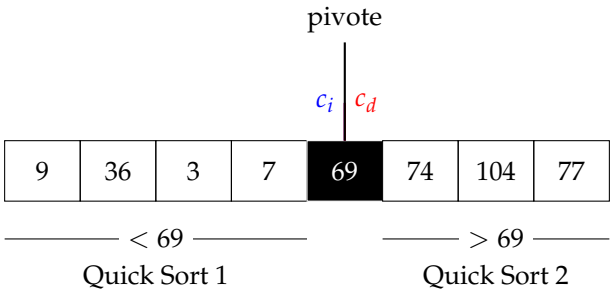
Pero eventualmente desde la izquierda se llega al pivote, no quedando más remedio que intercambiar al pivote por el número a la derecha. El valor del pivote no debe cambiar bajo ningún concepto porque, de lo contrario se perdería todo el trabajo hecho hasta ahora.



Luego, c_d no puede avanzar porque $c_d = pivote$, entonces c_i es el único que puede avanzar. Encontrará el 104, que es mayor al 69, por lo que se repite el mismo procedimiento.



Luego, como c_i no puede moverse más a la derecha, c_i se mueve a la izquierda. Como el 74 también es mayor al 69 se procede a la izquierda, hasta que $pivote = c_i = c_d$. Y se efectúa recursivamente el Quick Sort a la izquierda y otro a la derecha del pivote.



Eventualmente, a medida que continúan las recursiones y que terminen cuando la lista tiene 0 ó 1 elementos, se tiene la lista ya ordenada (si el ordenamiento se hace *in-place*, es decir, directamente sobre el arreglo y sin copiarlo o asignar memoria extra).

4.1. Complejidad del Quick Sort

A efectos de análisis, se tiene dos casos a considerar:

- **El mejor caso o caso promedio**: Cada recursión divide dos listas por la mitad (el pivote se considera despreciable si son listas muy grandes), teniendo así $T(n) = 2T(\frac{n}{2}) + ...$
- **El peor caso**: Si se toma el mayor o el menor elemento de toda la lista. En este caso, por cada recursión sólo se restará un elemento a la lista, teniendo así $T(n) = T(n-1) + ...$

Para cada recursión se debe recorrer una vez todos los elementos de la lista, dando como resultado $\theta(n)$. Teniendo así:

Para mejor caso:

$$T(n) = 2T(\frac{n}{2}) + \theta(n) \quad \Rightarrow \quad T(n) = \theta(n\log(n))$$

Y para el peor:

$$T(n) = T(n-1) + \theta(n) \quad \Rightarrow \quad T(n) = \theta(n^2)$$