¿Cómo ordenar una lista de números?

Germán Ariel Torres

Resumen. Este trabajo trata acerca de métodos y técnicas usadas para el ordenamiento eficiente de listas de números. Se analizan los métodos básicos, *bubble*, *insertion*, *selection* y *shell*, y los más elaborados como *heapsort* y *quicksort*, ejemplificados con casos sencillos desarrollados paso a paso. Finalmente se realiza una comparación sobre tiempos de ejecución para listas de un gran número de elementos.

1. Introducción.

Ordenar una lista es la operación de arreglar los elementos de acuerdo a algún criterio (no necesariamente matemático). En el caso de tratarse de números el criterio de orden podría ser "<", es decir, ordenar los elementos de la lista de menor a mayor. Aunque naturalmente un ser humano es capaz de implementar una metodología propia para ordenar un conjunto de elementos, esta tarea se vuelve extremadamente complicada cuando el número de elementos es grande, puesto que se necesitaría mucho tiempo y se podrían cometer errores. Ejemplos de esta situación podrían ser: ordenar alfabéticamente a los habitantes de una ciudad, ordenar una biblioteca, clasificar alfabéticamente las palabras de un lenguaje, ordenar una serie de páginas de internet, ordenar un conjunto de números enteros, etc.

La siguiente sección trata acerca de los métodos básicos para el ordenamiento de números. Si bien varios de estos métodos no son usados de manera práctica (como por ejemplo *bubble*, *insertion* o *selection*), la simplicidad en su planteo no significa necesariamente falta de eficiencia (por ejemplo, el algoritmo *shell* es uno de los más veloces). Inclusive, estos métodos sencillos se usan de manera combinada con un algoritmo eficiente con el fin de aprovechar las buenas cualidades de ambos métodos. La sección 3 presenta el método *heapsort*, cuyo planteo se basa en árboles binarios. En la sección 4 se analiza el algoritmo *quicksort* que aplica una estrategia del tipo dividir-conquistar. En la sección 5 se analizan tiempos de ejecución para listas muy grandes.

2. Métodos básicos.

En [1, 2] se presentan varios métodos de ordenamiento, entre los cuales podemos encontrar los que se describen a continuación.

2.1. El método *bubble*.

El método *bubble* es uno de los métodos más simples para clasificar datos. El algoritmo comienza comparando los dos primeros elementos, y si el primero es mayor que el segundo, entonces los intercambia. Después se compara el segundo elemento con el tercero, y así sucesivamente hasta terminar. Luego se empieza nuevamente desde el principio hasta que no haya más intercambios. Este algoritmo es altamente ineficiente y raramente es usado de manera práctica, razón por la cual se lo utiliza sólo para fines educativos.

En la figura 1 partimos con una lista desordenada. Se comienzan a hacer los intercambios de a pares hasta el final (primer columna). Luego se empieza nuevamente desde el principio (segunda columna). Finalmente, la lista se recorre otra vez desde el inicio sin haber ningún intercambio (tercer columna). Esto indica que se ha terminado el proceso y que se ha obtenido la lista ordenada. Notar que los pares recuadrados representan un cambio ya realizado en la lista.

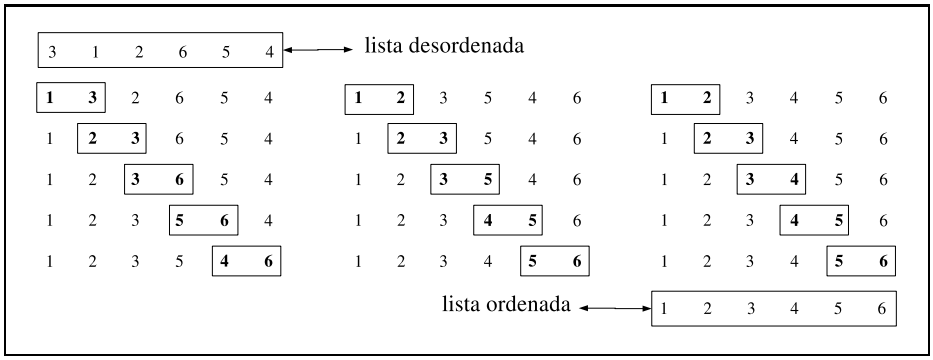

Figura 1: Ejemplo del método *bubble*.

2.2. El método *selection*.

El método *selection* es uno de los algoritmos más intuitivos, y consta de tres etapas: (1) encontrar el mínimo de la lista, (2) colocarlo en la primera posición

y (3) repetir los pasos anteriores para el resto de la lista (comenzando en la posición siguiente). Asumiendo que las comparaciones pueden ser hechas en tiempo constante, el costo de este algoritmo es $\mathcal{O}(n^2)$ para una lista de orden n.

En la figura 2 comenzamos con una lista de seis elementos desordenados. Primero se detecta que el elemento 1 es el mínimo de la lista y se lo traslada a la primera posición. De la lista restante, el elemento 2 es el más pequeño y se lo cambia al segundo lugar. A continuación la lista queda sin alterarse, puesto que el mínimo de la sublista restante ya está en la posición correcta. Luego intercambiamos el elemento 4 y finalmente el 5. Al concluir la operación se obtiene la lista ordenada de menor a mayor.

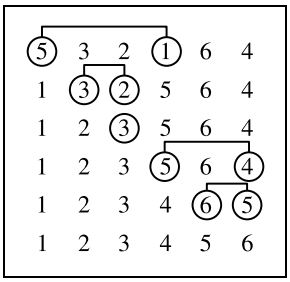

Figura 2: Ejemplo del método selection.

2.3. El método *insertion*.

El método *insertion* trabaja colocando cada elemento en el lugar correspondiente de una sublista. Si bien la complejidad de este algoritmo es como la del método *bubble*, es el doble más eficiente. Se procede formando una sublista ordenada de elementos a la cual se le va insertando el resto de la lista en el lugar adecuado, de tal manera que la sublista no pierda el orden. Esta sublista ordenada se va haciendo cada vez mayor, de modo que al terminar, la lista entera queda ordenada.

En la figura 3 se puede apreciar cómo es el procedimiento para una lista de seis elementos.

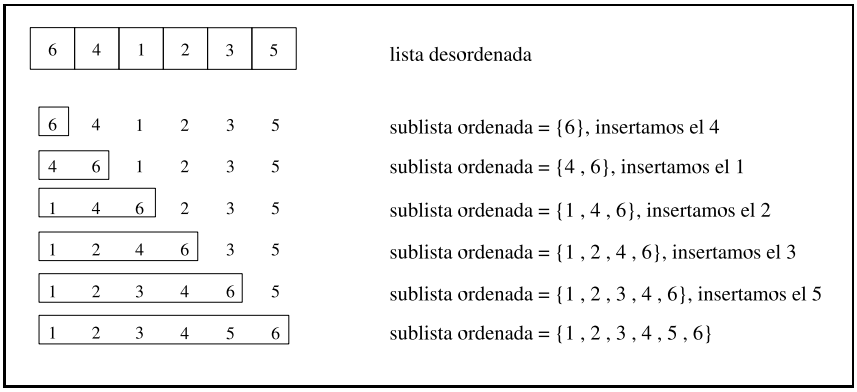

Figura 3: Ejemplo del método *insertion*.

2.4. El método *shell*.

El método *shell* fue inventado por Donald Shell en 1959 y es un algoritmo relativamente rápido y fácil de programar. Es importante destacar que este método realmente no clasifica datos, sino que incrementa la eficiencia de otros algoritmos de clasificación. Usualmente el método *bubble* (o el *insertion*) es usado en cada paso, pero otros algoritmos pueden ser usados.

Este método ordena los datos clasificando cada n elementos, donde n puede ser un número menor que la mitad de la longitud de la lista. Una vez que el ordenamiento inicial es realizado, n es reducido y se realiza otro ordenamiento hasta que n sea igual a 1. Notar que el ordenamiento final (cuando $n = 1$) es importante puesto que podrían quedar elementos desordenados.

Veamos un ejemplo de una lista desordenada de nueve elementos: 8, 4, 1, 5, 7, 6, 9, 3, 2. Una sucesión apropiada para n puede ser $\{3, 2, 1\}$. En la parte izquierda de la figura 4 se observa el ordenamiento para $n = 3$: en el primer recuadro se tienen en cuenta el primer, cuarto y séptimo elemento, y se ordenan solamente ellos (los que están sombreados); en el segundo recuadro nos desplazamos una posición a la derecha, y clasificamos los elementos sombreados, es decir, el segundo, el quinto y el octavo; en el tercer recuadro clasificamos solamente el tercer, sexto y noveno elemento. En la parte derecha de la misma figura se ve el ordenamiento para cuando $n = 2$ (es decir, se clasifica elemento por medio) y $n = 1$ (último recuadro).

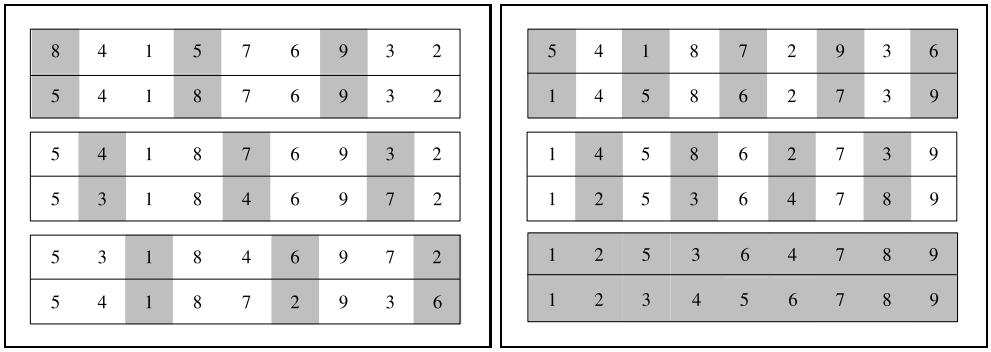

Figura 4: Ejemplo del método shell.

3. El método *heapsort*.

En las ciencias de la computación, un árbol binario es una estructura de datos en la cual cada nodo tiene a lo sumo dos hijos, los cuales son hermanos entre sí. Típicamente los nodos hijos son llamados hijo izquierdo e hijo derecho. Una línea dirigida conecta el padre con el hijo. Un nodo que no tiene hijos es llamado una hoja. La profundidad de un nodo n es la longitud del camino desde la raíz hasta el nodo. El conjunto de todos los nodos a una profundidad dada es a veces llamado el nivel del árbol. Si existe un camino descendente desde un nodo p a un nodo q, entonces p es un ancestro de q y q es un descendiente de p. El tamaño de un nodo es el número de descendientes que tiene. El tamaño de un árbol es la cantidad de elementos que lo componen. En (1) de la figura 5 podemos ver un ejemplo de un árbol binario de tamaño 9, con un nodo raíz cuyo valor es 2.

Un árbol binario casi completo es un árbol binario en el cual se cumplen tres condiciones: (a) todas las hojas están en el último nivel o en los últimos dos niveles, (b) todas las hojas están en la posición más a la izquierda posible, (c) todos los niveles están completados con nodos (posiblemente con excepción del último nivel). En (2) y (3) de la figura 5 se pueden ver ejemplos de árboles binarios casi completos. En (4)-(7) de la misma figura, se pueden ver ejemplos de árboles binarios no casi completos: en (4) el 9 debería estar a la izquierda, en (5) el segundo nivel debe ser completado, en (6) 2 y 6 deben ser movidos a la izquierda, y en (7) hay hojas en tres niveles.

Un *heap* minimal es un árbol binario casi completo en el cual el valor en

cada nodo padre es menor o igual al valor de sus nodos hijos. Claramente se ve que el nodo raíz tiene el valor mínimo. Notar también que cualquier camino desde una hoja a la raíz pasa por los datos de manera descendente. Cuando se habla acerca de "menor o igual a", lo que se quiere decir es que hay una relación de orden o criterio de comparación para clasificar los elementos del *heap*, y este criterio no es necesariamente matemático, puesto que los objetos del *heap* podrían no ser números. Notar también que la propiedad de los *heaps* minimales no dicen nada acerca de la relación de orden entre nodos hermanos. En (8) de la figura 5 se puede ver un ejemplo de un *heap* minimal.

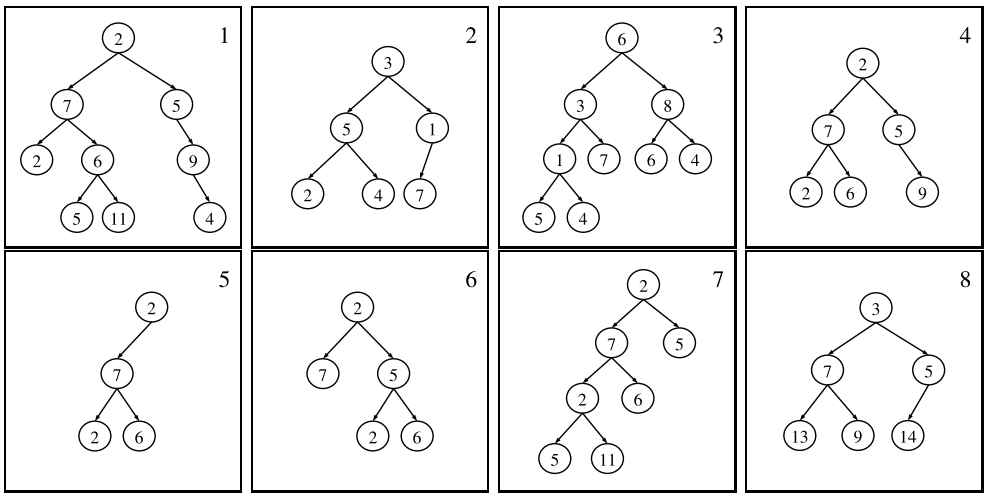

Figura 5: Ejemplos de árboles binarios.

Una lista de objetos puede ser pensada como un árbol binario casi completo numerando los objetos nivel por nivel, de arriba hacia abajo, y de izquierda a derecha, como muestra la figura 6.

Con este tipo de ordenamiento, si i es el índice de un nodo, se tiene que el hijo izquierdo tiene índice $2i$ y el hijo derecho tiene índice $2i+1$. Análogamente, el padre del nodo i tiene índice $i/2$, donde "/" significa la división entera.

Para agregar un elemento a un *heap* se procede de la siguiente manera: (a) se coloca el elemento a agregar al final del árbol, (b) si el valor del padre es mayor se intercambia su posición, (c) este mismo chequeo se realiza hasta llegar al nodo raíz. Al finalizar el procedimiento habremos obtenido un *heap* minimal con un objeto adicional. Observar el esquema de la figura 7 para entender el proceso: en (1) tenemos el *heap* original, en (2) agregamos el elemento 4 y comparamos,

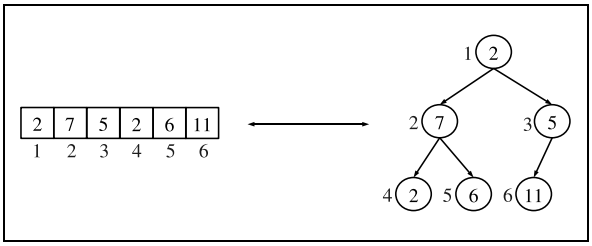

Figura 6: Lista de objetos como árboles binarios casi completos.

en (3) intercambiamos el hijo 4 con el padre 13, en (4) intercambiamos el hijo 4 con el padre 7 y la última comparación con el nodo raíz no produce cambios.

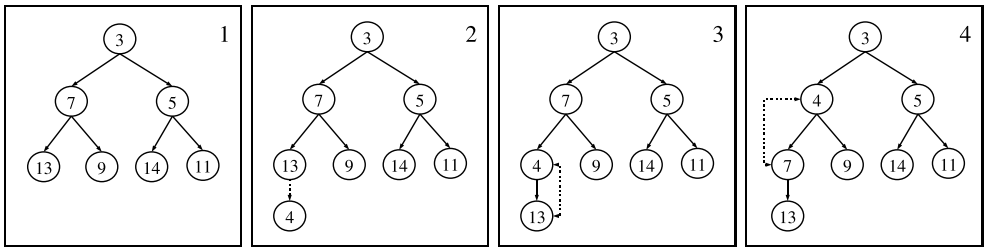

Figura 7: Agregando un elemento al *heap*.

En el caso de remover un elemento de un *heap*, siempre quitaremos el nodo raíz. Por lo tanto siempre estaremos eliminando el elemento más pequeño. El problema es entonces reordenar los elementos para obtener nuevamente un *heap*. Los pasos son los siguientes: (a) eliminar el nodo raíz y reemplazarlo por el último elemento, (b) si el hijo más pequeño del nodo raíz es menor que el nodo raíz, lo intercambiamos, caso contrario no cambiamos nada, (c) repetir este proceso, siguiente la rama, hasta llegar al último nivel. Al finalizar el procedimiento habremos obtenido un *heap* minimal con un objeto menos. Observar el esquema de la figura 8 para entender el proceso: en (1) tenemos el *heap* original, en (2) quitamos el primer elemento 3 y colocamos el último elemento 15 en su lugar, en (3) intercambiamos 15 con su hijo más chico 7, en (4) intercambiamos 15 con su hijo más chico 10 y a partir de este punto no se pueden hacer más intercambios.

Para convertir un árbol binario casi completo en un *heap* se procede de la siguiente forma: (a) Se calcula el índice del último nodo padre: TamañoHeap/2; (b) a partir del último padre y hasta el nodo raíz se realiza lo siguiente: (b.1)

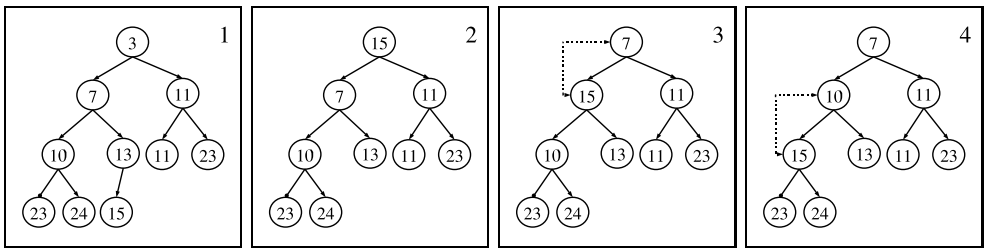

Figura 8: Quitando un elemento al *heap*.

se compara el nodo padre con el nodo más pequeño, y en caso de que éste último sea menor, se intercambia, caso contrario se deja como está, (b.2) en caso de intercambio comparar de nuevo pero ahora en un nivel más abajo, (b.3) continuar este proceso hasta que no se pueda intercambiar más. Al terminar habremos ordenado completamente el árbol y tendremos un *heap* minimal.

Veamos la figura 9 para entender el procedimiento: en (1) tenemos el árbol binario casi completo original, en (2) comenzamos con el último padre 11 y lo intercambiamos con su hijo mas pequeño 5 (notar que los índices de los padres están indicados al lado de los elementos del árbol), en (3) continuamos con el padre de índice 3 y valor 3, y lo cambiamos por su hijo más chico 1, en (4) continuamos con el padre de índice 2 y valor 18 y lo intercambiamos con su hijo más pequeño 5, en (5) seguimos este procedimiento con el nivel siguiente y obtenemos una nueva permutación con el hijo más chico 11, en (6) intercambiamos el nodo raíz con el hijo 15, en (7) el nodo 15 es ahora padre de otros hijos, y lo volvemos a permutar con su hijo más chico 3.

El algoritmo de ordenamiento *heapsort* se realiza de la siguiente manera: (a) se convierte la lista en un árbol binario casi completo (figura 10), (b) se convierte el árbol binario casi completo en un *heap* minimal (ver (1) de la figura 11), (c) se remueve el nodo raíz del árbol quedando nuevamente un *heap* minimal (ver (2) de la figura 11) y (d) repetir este procedimiento hasta vaciar el árbol (ver (3)-(8) de la figura 11). Al finalizar habremos cumplido nuestro objetivo de ordenar la lista (ver figura 12).

La velocidad de este algoritmo depende de la cantidad de elementos a ordenar. Si n es la longitud de la lista, decimos que la velocidad de este algoritmo es de $\mathcal{O}(n \log(n))$, aún en el caso promedio o en el peor caso, y además no necesita almacenamiento extra.

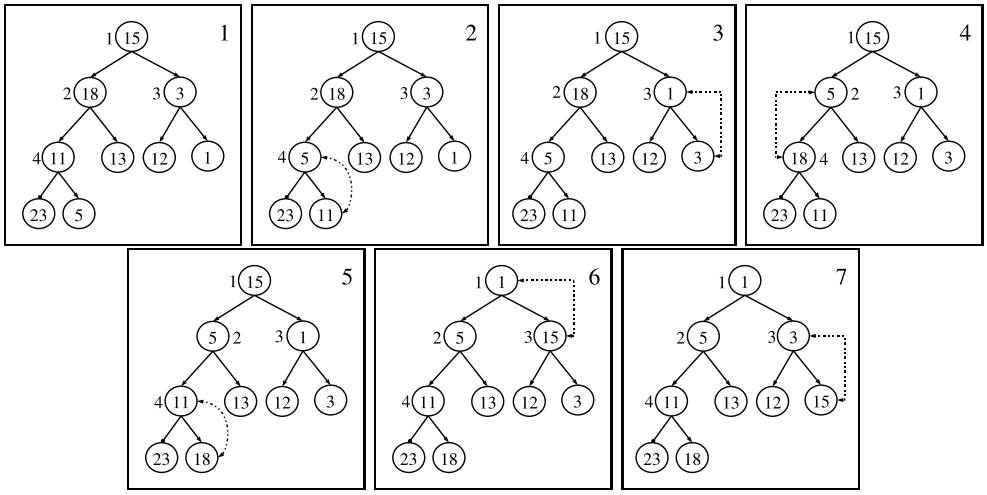

Figura 9: Convirtiendo un árbol binario casi completo en un *heap* minimal.

4. El método *quicksort*.

El método *quicksort* emplea una estrategia de dividir-conquistar para dividir una lista en dos sublistas, y de esa manera trabajar de manera recursiva. Una descripción detallada puede encontrarse en [4]. Los pasos son los siguientes:

- Tomar un elemento de la lista al cual llamaremos pivote.
- Reordenar la lista para que todos los elementos que son menores que el pivote estén a la izquierda del pivote, y todos los mayores a la derecha del

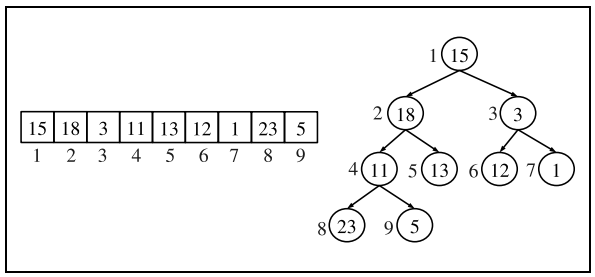

Figura 10: Ejemplo del algoritmo *heapsort*: transformar una lista en un árbol binario casi completo

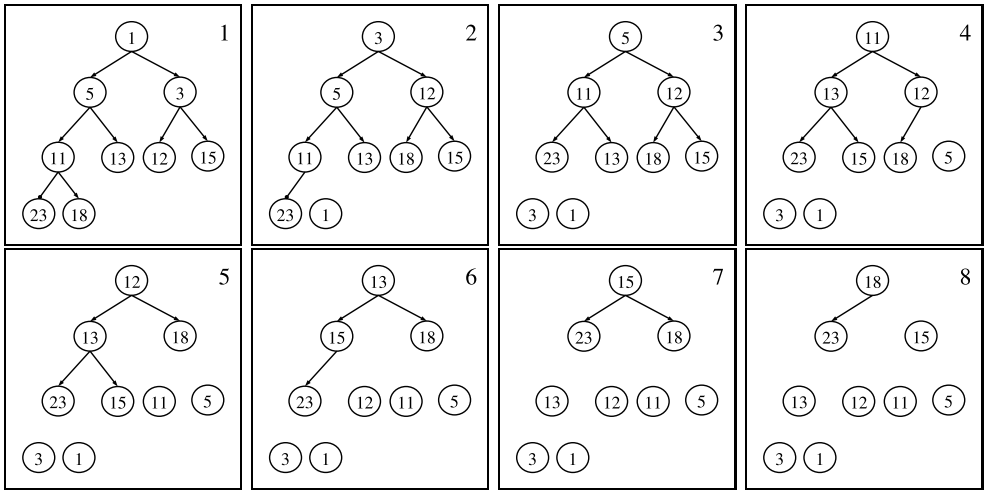

Figura 11: Ejemplo del algoritmo *heapsort*.

pivote.

- Recursivamente clasificar la sublista de los elementos mayores que el pivote, y la sublista de los elementos menores que el pivote.

El caso base de la recursión son listas de tamaño cero o uno, que están siempre ordenadas. El algoritmo termina siempre, puesto que en cada iteración coloca al menos un elemento en su lugar final.

Veamos un ejemplo de su funcionamiento en la figura 13, donde la elección del pivote será siempre el primer elemento de la lista. En el ejemplo los pivotes

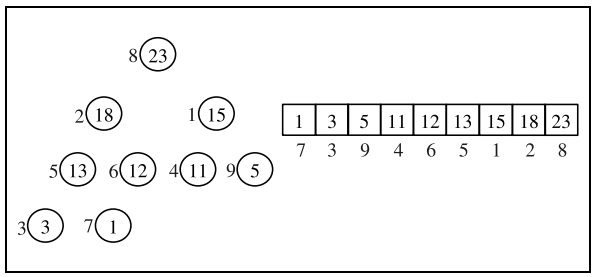

Figura 12: Ejemplo del algoritmo *heapsort*.

están indicados con un cuadrado en negrita, y los intercambios de elementos por cuadros sombreados. En la primera fila se encuentra una lista desordenada. Tomamos como pivote el primer elemento 8. Luego comenzamos a recorrer la lista desde el segundo elemento hacia el último y a la vez desde el último hacia el segundo. Se produce un intercambio cuando un elemento recorriendo de izquierda a derecha es mayor al pivote, y a la vez cuando un elemento recorriendo de derecha a izquierda es menor al pivote: el elemento 12 por el 7, el 14 por el 2, el 13 por el 1, y el 15 por el 5. Luego colocamos el pivote en su posición final (sexta fila). Ahora tenemos la lista subdivida en dos sublistas (una a la izquierda del pivote y otra a la derecha), y volvemos a repetir el procedimiento hasta encontrar sublistas de un solo elemento.

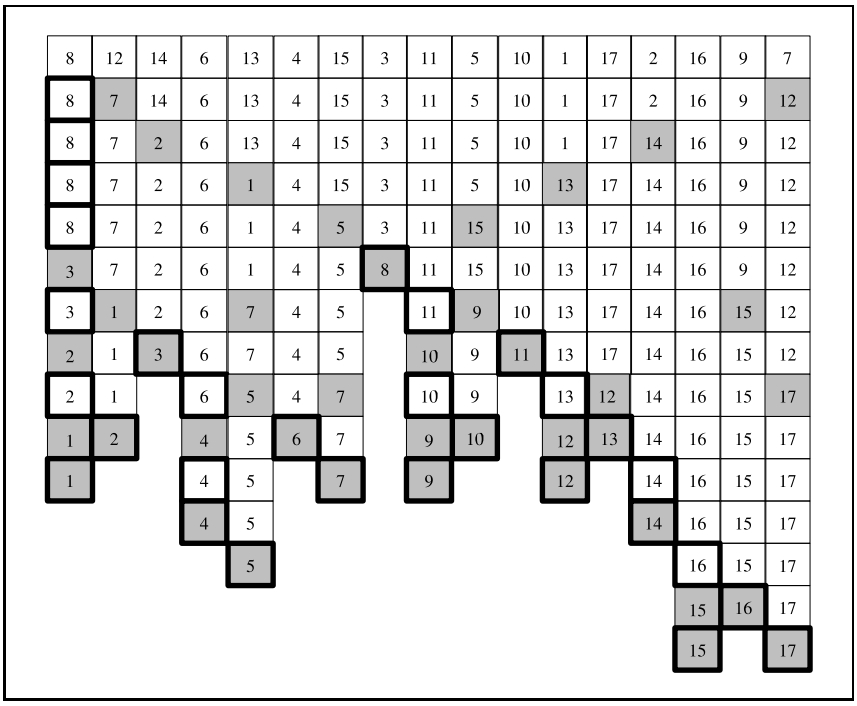

Figura 13: Ejemplo del algoritmo *quicksort*.

La eficiencia del algoritmo se ve mayormente afectada por cómo es elegido el pivote. Usualmente la elección del pivote se hace tomando el primer elemento de la lista, sin embargo esto puede ser muy perjudicial para el peor caso

(que irónicamente es el caso de una lista previamente ordenada) llevando a una eficiencia de $\mathcal{O}(n^2)$ donde n es el tamaño de la lista. Con una adecuada elección del pivote, como por ejemplo elegido aleatoriamente, se puede lograr una complejidad algorítmica de $\mathcal{O}(n \log(n))$. El peor caso de ordenamiento para el algoritmo *quicksort* no es un problema meramente teórico: en el caso de servicios web, es posible que un atacante deliberadamente explote la posibilidad de un peor caso de ordenamiento que causará una ejecución lenta. Otra opción sería elegir como pivote al elemento del medio, sin embargo una elección aleatoria es la mejor opción.

Como virtualmente todo el tiempo de cálculo del algoritmo *quicksort* es empleado en particionar, una buena implementación es importante. En particular, si todos los elementos que están siendo particionados son iguales, las particiones degeneran en peores casos y se gasta tiempo haciendo intercambios que no afectarán en nada al resultado. Por esta razón es útil elegir un pivote-lista que contenga elementos idénticos al pivote.

Otra posible mejora es utilizar otro algoritmo de ordenamiento cuando se llega a sublistas de tamaño pequeño (como por ejemplo *insertion*). De esta manera se explotan las mejores características de cada algoritmo de ordenamiento.

Quicksort puede ser paralelizado debido a su naturaleza de dividir-conquistar. Cada sublista puede ser ordenada independientemente de las otras. Si tenemos p procesadores, podemos dividir una lista de n elementos en p sublistas en un tiempo promedio de $\mathcal{O}(n)$, y entonces clasificar cada una de éstas en un tiempo promedio de $\mathcal{O}((n/p) \log(n/p))$. Una ventaja de la paralelización es que no se necesita sincronización.

5. Comparaciones.

La siguiente tabla comparativa muestra tiempos de ejecución (en segundos de CPU) para ordenar listas con distintos métodos. Los programas están escritos en Fortran 90, y se usó una computadora con procesador de 2.8 GHz. de velocidad con 1 Gb. de memoria RAM. Varias de las subrutinas usadas en esta implementación aparecen en [5]. Para $n = 10000$, es decir, una lista de 10000 elementos, los algoritmos *shell*, *heapsort* y *quicksort* son tan veloces que no se alcanza a percibir el tiempo que tarda. Como era de esperarse, el método más lento es *bubble*. Para $n = 50000$ y $n = 100000$ las diferencias entre los métodos comienzan a ser más claras, siendo los más lentos *bubble*, *selection* e *insertion*, y los más rápidos *shell*, *quicksort* y *heapsort*. Para poder ver mejor las diferencias

entre los métodos más rápidos, aumentamos el número de elementos de la lista (notar que en el cuadro comparativo no aparecen los tiempos de ejecución para los métodos lentos pues estos tiempos son enormes). Claramente observamos que el método *quicksort* es el más veloz, seguido por el método *shell* y *heapsort*. Las listas de números a ordenar fueron construídos utilizando un generador aleatorio de números reales.

	10^4	5×10^4	10^5	5×10^5	10^6	5×10^6	10^7	5×10^7
Bubble	0.58	14.64	64.49					
Selection	0.14	3.55	15.18					
Insertion	0.08	1.93	8.28					
Shell	~ 0	0.02	0.04	0.26	0.64	5.02	11.94	88.03
Heapsort	~ 0	0.01	0.03	0.33	0.94	8.50	20.38	148.02
Quicksort	~ 0	0.01	0.02	0.11	0.23	1.31	2.78	15.46

6. Conclusiones.

Se han analizado una serie de métodos básicos y elaborados que tienen por objeto ordenar listas de números. Las pruebas de tiempos de ejecución son contundentes: para listas grandes hay algunos métodos (como el *bubble*, *selection* e *insertion*) cuya aplicación es imposible; por el contrario los métodos *heapsort*, *shell* y *quicksort* han demostrado tener una buena performance, siendo los más veloces los dos últimos. En la práctica se utilizan métodos híbridos, aprovechando las ventajas de dos métodos: por ejemplo, el método *quicksort* es el más rápido cuando la lista es muy grande, pero se usa un método básico cuando las sublistas llegan a un tamaño adecuado. De esta manera se han presentado una serie de estrategias, de las más simples a las más complejas, que dan una respuesta satisfactoria al problema de ordenamiento de objetos.

Referencias

[1] Knuth, D. The Art of Computer Programming, Volume 3: Sorting and Searching, Third Edition. Addison-Wesley, 1997.

[2] Heileman, G. L. Data Structures, Algorithms, and Object-Oriented Programming. 1996. McGraw-Hill.

[3] Pratt, V. Shellsort and sorting networks (Outstanding dissertations in the computer sciences). Garland, 1979.

[4] Hoare, C. A. R. Quicksort, Computer Journal, 1962, Vol. 5, Nro. 1, pp. 10-15.

[5] Numerical Recipes in Fortran.

Germán A. Torres
Facultad de Matemática, Astronomía y Física (FaMAF) – CIEM (CONICET)
Universidad Nacional de Córdoba
Ciudad Universitaria (5000), Córdoba, Argentina.
(Email: torres@famaf.unc.edu.ar)