Análisis y diseño de algoritmos
Proyecto final

Emilio E. G. Cantón Bermúdez
Roberto Gervacio Guendulay

09/05/2019

Resumen

El ordenamiento es uno de los bloques elementales para la creación de algoritmos más complejos. Para el desarrollo de estos algoritmos, el tiempo que te tome ordenar tu información puede llegar a ser vital. En este trabajo analizamos los algoritmos **Bubble Sort, Quick Sort, Radix Sort, Heapsort** y **Mergesort**.

Heapsort

De acuerdo con Srini Devadas, un *heap* es una implementacióna de la estructura de datos abstracta, *priority queue*. En general el objetivo, tal como es definido en *priority queue*, es poder **insertar una llave**, **remover el valor máximo** (tomando en cuenta un tipo de comparación indefinida). Un *heap* lo que logra es implementar la manera más eficiente de cumplir estos requisitos al aplicar un árbol binario semi-completo.

Comenzamos considerando un arreglo de números (llaves)

$$[4, 1, 6, 8, 9, 5, 2]$$

Dado éste arreglo observamos que si tomamos el índice del primer número (comenzando por 1) y aplicamos la fórmula $2 \cdot i + 1$ para el hijo izquierdo y $2 \cdot i + 2$ para el hijo derecho podemos ver que nuestro arreglo ya es un árbol binario completo (en este caso).

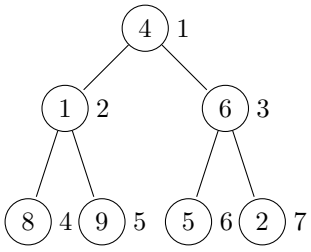


Fig 1.1: Representación de *heap*

La operación elemental para esta estructura de datos es *heapify*, la cual ordena de acuerdo con la comparación establecida un padre con sus hijos (el sub-árbol más pequeño). En este caso construiremos lo que se conoce como un *max-heap* al comparar con el número mayor.

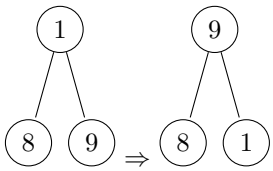


Fig. 1.2: Max-Heapify

Tomamos ésta como nuestra operación elemental y asumimos que para cualquier árbol al que se le aplique *max-heapify* sus hijos **ya son max-heaps**. Teniendo en cuenta que si a un padre le aplicamos nuestra operación elemental, es decir, lo cambiamos por unos de los hijos, entonces tenemos que asegurar que le hijo sigue siendo *max-heap*, por lo tanto, aplicamos recursivamente al hijo la operación y así sucesivamente.

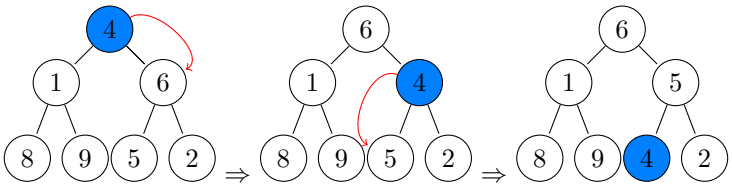


Fig 1.3: Max-Heapify nodo raíz

Como podemos ver para el peor de los casos para un nodo raíz en un *heap* (asumiendo que todos sus hijos son *max-heaps*) el número de operaciones elementales que realizará es $\lfloor \log_2 n \rfloor$ donde n es el número de nodos en el árbol. En este caso tenemos que $n = 7$ y $\lfloor \log_2 n \rfloor$=2 y como vemos en la Fig. 1.3 efectivamente se realiza dos veces la operación.

Aprovechando la suposición que todos los nodos hijos ya son *max-heaps* comenzamos nuestro algoritmo de ordenamiento en $\frac{n}{2} \in \mathbb{N}$[1] basado en la observación en que **todos los nodos** $[\frac{n}{2} + 1, \cdots, n]$ **son hojas**. Nos interesan estos nodos en particular ya que al recibir el arreglo nuestra representación no es un *max-heap* por ahota, por lo que, hay que aplicar las operaciones elementales correspondientes para dejarlo en éste estado. De esta manera si comenzamos por los nodos en los que sus hijos son hojas, tenemos asegurado que éstos ya son *max-heaps* por definición.

[1] *La división que se realiza es entera, es decir, que* $n = 2m + r$ *y el resultado de aplicar la división entera* $\frac{n}{2}$ *nos da* m

Ahora comenzamos a analizar el número de operaciones que realizaremos para obtener la complejidad del algoritmo. Dada la estructura de un árbol binario completo[2] sabemos que el número de nodos a partir de la raíz de puede representar por la siguiente serie

$$1,\ 2,\ 4,\ 8, \cdots, \frac{n+1}{16}, \frac{n+1}{8}, \frac{n+1}{4}, \frac{n+1}{2}$$

De acuerdo con nuestras observaciones sobre el número máxmo de operaciones para un nodo raíz en un árbol, el número de nodos en cierto nivel y que cualquier nodo intermedio en un árbol binario es por sí solo un árbol binario, podemos ver que el número total de operaciones se da por la siguiente sumatoria

$n_0 = n + 1$

$$\frac{n_0}{4} \cdot 1 + \frac{n_0}{8} \cdot 2 + \frac{n_0}{16} \cdot 3 + \cdots + \frac{n_0}{n_0/2} \cdot (\frac{n_0}{2} - 1) + \frac{n_0}{n_0} \cdot \log_2 n$$

$$= (n_0)(\frac{1}{4} \cdot 1 + \frac{1}{8} \cdot 2 + \frac{1}{16} \cdot 3 + \cdots + \frac{1}{n_0/2} \cdot (\frac{n_0}{2} - 1) + \frac{1}{n_0} \cdot \log_2 n)$$

$$= (\frac{n_0}{4})(1 \cdot 1 + \frac{1}{2} \cdot 2 + \frac{1}{4} \cdot 3 + \cdots + \frac{1}{n_0/8} \cdot (\frac{n_0}{2} - 1) + \frac{1}{n_0/4} \cdot \log_2 n)$$

$$= (\frac{n_0}{4})(1 \cdot 1 + \frac{1}{2} \cdot 2 + \frac{1}{4} \cdot 3 + \cdots + \frac{1}{n_0/8} \cdot (\frac{n_0}{2} - 1) + \frac{1}{n_0/4} \cdot \log_2 n)$$

$k = \log_2(n)$

$$= \frac{n_0}{4} \cdot \sum_{i=0}^{k} \frac{i+1}{2^i}$$

Con la prueba de proporción de d'Alembert podemos ver que la sumatoria converge

$$L = \lim_{n\to\infty} \left| \frac{\frac{n+2}{2^{n+1}}}{\frac{n+1}{2^n}} \right| = \lim_{n\to\infty} \left| \frac{2^n \cdot (n+2)}{2^{n+1} \cdot (n+1)} \right| = 0$$

Dado que

$$(n)2^n + 2^{n+1} < (n+1)2^{n+1}$$
$$(n)2^n < (n+1)2^{n+1} - 2^{n+1}$$

[2] *Tomamos un árbol binario completo como ejemplo ya que es el peor caso*

$$(n)2^n < (n)2^{n+1}$$
$$2^n < 2^{n+1}$$

Y finalmente podemos ver que nuestra complejidad se comporta de la siguiente manera

$$\frac{n_0}{4}\sum_{i=0}^{\infty}\frac{i+1}{2^i} = \frac{n_0}{4}(1+2+\frac{3}{4}+\frac{4}{8}+\frac{5}{16}...) = \frac{n_0}{4}\cdot 4 = n_0 = n+1$$

$$= O(n)$$

Merge Sort

Tomaremos la comparación entre dos número como la operación elemental. Dicho ésto, primero definiremos el procedimiento *merge*, el cual es básico para la demostración de la complejidad del algoritmo *merge sort*.

Supongamos que tenemos dos tuplas[3], a y b, las cuales formarán una tupla, c. Para cada paso en este procedimiento haremos una comparación entre a_i y b_j en la cual (definiendo el comparador como el menor o igual) obtenemos $min(a_i, b_j)$ y lo agregamos a c. Consecuentemente, si a_i fue el menor de éstos dos, obtenemos $min(a_{i+1}, b_j)$ y así consecutivamente hasta obtener c el cual tendrá todos los valores de a y de b ordenados. Cómo podemos ver esta operación nos toma $|a| + |b|$ operaciones elementales ya que realizamos una operación por cada valor de a y b.

$$\therefore merge = O(n)$$

Habiendo definido el procedimiento *merge* podemos comenzar a calcular la complejidad. Dividimos en dos[4] recursivamente una n-tupla t de manera que nos quedamos con una tupla a y una tupla b tal que

$$a = (t_0, t_1, \cdots, t_{\frac{1}{2}-1})$$
$$b = (t_{\frac{1}{2}}, t_{\frac{1}{2}+1}, \cdots, t_{n-1})$$

De esta manera podemos continuar dividiendo (recursivamente) las tuplas resultantes y hasta llegar a nuestro caso base el cual es una tupla de cardinalidad 1 la cual por definición esta ordenada. Al llegar a este punto comenzamos a aplicar el procedimiento *merge* con las tuplas que hemos obtenido como podemos ver en la Fig. 2.1. para la tupla $(4, 2, 5, 3, 1)$

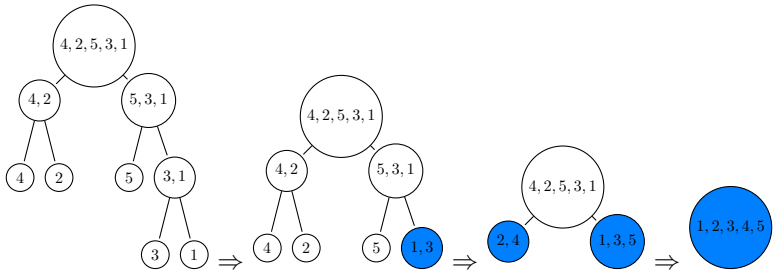


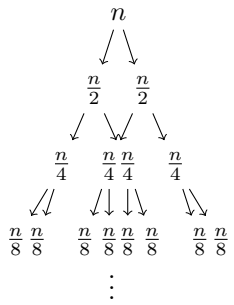
Fig 2.1: Ejemplo de Merge Sort

[3] *Definida matemáticamente como una lista ordenada de elementos (en este caso números)*
[4] *Tomamos $n/2$ como una división entera tal que si $n = 2m + q \Rightarrow \frac{n}{2} = m$*

Teniendo en cuenta todo lo anteriormente definido podemos definir las llamadas recursivas de la siguiente manera

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

Donde c es una constante que define el tiempo que toma nuestra operación elemental, la comparación. Representando nuestra función recursiva en un árbol (Fig 2.2) de llamadas podemos comenzar a ver la complejidad



$$1 \quad 1 \quad 1 \quad 1 \quad \cdots \quad 1 \quad 1 \quad 1 \quad 1$$

Como podemos observar se forma las siguiente sumatoria

$$\begin{aligned}
n + \frac{n}{2} + \frac{n}{2} + \frac{n}{4} + \frac{n}{4} + \frac{n}{4} + \frac{n}{4} + \cdots + \overbrace{1 + 1 + 1 + 1 + 1 + \cdots + 1}^{n} \\
&= n + 2 \cdot \frac{n}{2} + 4 \cdot \frac{n}{4} + \cdots + n \cdot 1 \\
&= n + \frac{2n}{2} + \frac{4n}{4} + \cdots + n \cdot 1 \\
&= \overbrace{n + n + n + \cdots + n}^{\log_2(n)+1} \\
&= n \cdot \log_2(n) + 1 \\
&= n \log_2(n) + n \\
&= O(n \log_2(n))
\end{aligned}$$

Radix Sort

Este es un algoritmo que se basa en el funcionamiento de **counting sort** por lo que obtiene las ventajas de este, como no requerir de la comparación entre números (lo cuál lo hace el mejor para ordenar número de longitud fija) y ser estable (no cambia la posición de los elementos de entrada). Para poder analizar la complejidad es necesario entender la complejidad de counting sort.

Dado un arreglo $A|A_i \in \mathbb{N}$ de tamaño n debemos contar las veces en las que aparece cada valor único (llave) en el. Para eso nos ayudaremos de un arreglo auxiliar C. Ya que solo nos importa guardar los valores únicos podemos aproximar el tamaño de C como:

$$\begin{gathered}\text{Sea } k = \max\{A\} \\ \text{Entonces } |\mathrm{C}| = \mathrm{k}\end{gathered}$$

Por ejemplo, supongamos que $A = [1, 4, 1, 7, 1, 7, 10, 3, 1]$, entonces $k = 10$, por lo que el tamaño de nuestro arreglo $|C| = 10$

De esta forma estamos creando un espacio para cada posible llave en A. Parece intuitivo ver que entre menor sea la diferencia entre las llaves de A este algoritmo tiene un comportamiento favorable y evita el desperdicio de memoria.

Ya que no hacemos la comparación entre los números podemos definir la operación básica como la iteración sobre el arreglo A para construir el arreglo auxiliar C

$$\sum_{i=1}^{n} C_{A_i} = C_{A_i} + 1$$

Lo anterior se hizo n veces por lo que hasta ahora la complejidad parece ser $O(n)$. Sin embargo aún no hemos terminado, apenas hemos construido nuestro arreglo C para poder ordenar A.

Para nuestro ejemplo $C = [3, 0, 1, 1, 0, 0, 2, 0, 0, 1]$

El siguiente paso es iterar sobre C y así obtener A ordenado, es decir:

$$\sum_{i=1}^{k} C_i$$

Donde el valor en C_i nos dira el número de veces que i se repitió. Como el tamaño de C fue definido por k, entonces la complejidad de esta segundo paso fue $O(k)$.

Para nuestro ejemplo podemos decir que si $i = 1$, entonces $C_i = 3$, lo cual se interpreta como que existe 3 veces el número 1, seguido de 1 vez el número 3, es decir:

$$[1, 1, 1, 3, 4, 7, 7, 10]$$

Nos damos cuenta que obtener el resultado nos llevo:

$$O(n+k)$$

iteraciones. Donde si k fuera muy pequeño tendriamos un tiempo lineal.

Ahora podemos seguir con el análisis de **radix sort**. Existen dos variantes de este algoritmo:

- Usar las cifra más significativa
- Usar las cifra menos significativa

Comunmente se utiliza la cifra menos significativa, lo cual se traduce a pasar por cada número de derecha a izquierda. Entonces si definimos a $D|D_i \in \mathbb{N}$ tenemos que ordenar digito a digito cada D_i, esto lo podemos hacer con **countig sort**.

Por ejemplo, sea $D = [53, 12, 11, 634, 89]$

Sea m el número de iteraciones sobre D

$$m = \text{mayor número de digitos en } D$$

para nuestro ejemplo $m = 3$ y en la primer iteracion de m, es decir $m = 1$ nos fijaremos en los útlimos digitos:

$$3 \leftarrow 5\mathbf{3}$$
$$2 \leftarrow 1\mathbf{2}$$
$$1 \leftarrow 1\mathbf{1}$$
$$4 \leftarrow 63\mathbf{4}$$
$$9 \leftarrow 8\mathbf{9}$$

Ahora con estos digitos y con ayuda de **counting sort** obtendriamos $[0, 1, 2, 3, 4, 0, 0, 0, 0, 1]$ pero cada uno de esos digitos en realidad es una referencia al numero completo, por lo que nuestra lista actual $D = [11, 12, 53, 634, 89]$. Debemos repetir esto hasta acabar con las iteraciones de m.

$$m = 2$$
$$1 \leftarrow \mathbf{1}1$$
$$1 \leftarrow \mathbf{1}2$$
$$5 \leftarrow \mathbf{5}3$$
$$3 \leftarrow 6\mathbf{3}4$$
$$8 \leftarrow \mathbf{8}9$$

$$D = [11, 12, 634, 53, 89]$$

$$m = 3$$
$$0 \leftarrow 11$$
$$0 \leftarrow 12$$
$$0 \leftarrow 53$$
$$6 \leftarrow \mathbf{6}34$$
$$0 \leftarrow 89$$

$$D = [11, 12, 53, 89, 634]$$

Cabe resaltar que podemos tomar como 0, los digitos que nos hagan falta.

Lo anterior fue hecho m veces es decir $O(m)$ pero en cada de una de esas m veces llamamos a **counting sort** es decir $O(m(n+k))$. Cabe resaltar que ese número k en realidad estará definido por la base de los número actuales en D, por ejemplo como los números en D eran de base 10, k para counting sort siempre fue 10. Por lo que la complejidad de **radix sort** queda definida como:

$$O(m(n+b))$$

Donde m es el máximo número de digitos, n el tamaño del arreglo y b la base de los números en el arreglo.

Quick Sort

Supongamos que tenemos la siguiente tupla

$$A = (a_0, a_1, a_2, \cdots a_n)$$

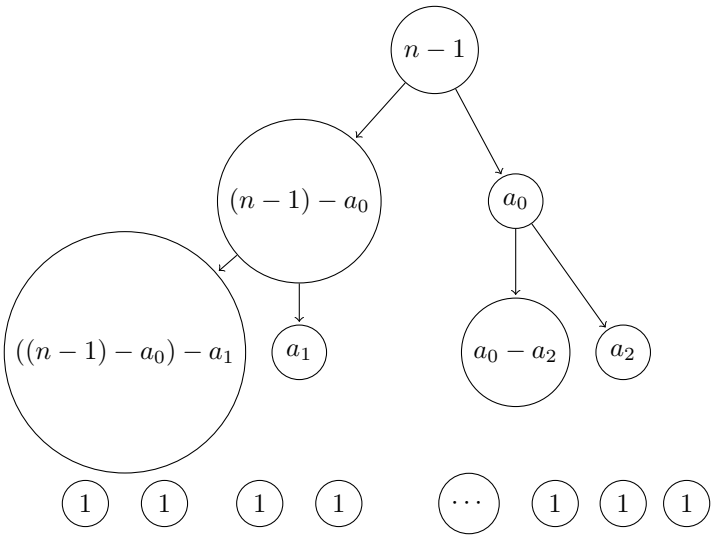
Tomamos una a_i que corresponda a la mediana de la tupla y comparamos a_i con cada elemento de la tupla tal que $i \neq j$. Finalmente, obtenemos dos k-tuplas tal que $k < n$ de la siguiente manera

$$B = (a_j \mid a_j \leq a_i)$$
$$C = (a_j \mid a_j > a_i)$$

Teniendo las tuplas resultantes aplicamos el mismo paso recursivamente a cada una. Por tanto, podemos expresar esta llamada recursiva a cada uno. Podemos expresar esta función recursiva de la siguiente manera.

$$T(n) = T((n-1) - a) + T(a) + (n-1)$$

Así mismo podemos representar esta función en un árbol recursivo



Podemos ver que se forma la siguiente sumatoria por subniveles donde cada sumando es un nodo

$$= (n-1)+((n-1)-a_0+a_0)+((((n-1)-a_0)-a_1)+a_1+(a_0-a_2)+a_2)+\cdots+1+1+1+1+1+\cdots+1+1+1$$

$$= (n-1)+((n-1)-a_0+a_0)+((n-1)-a_0-a_1+a_1+a_0-a_2+a_2)+\cdots+1+1+1+1+1+\cdots+1+1+1$$

$$= (n-1)+\cancel{((n-1)-a_0+a_0)}+\cancel{((n-1)-a_0-a_1+a_1+a_0-a_2+a_2)}+\cdots+\overbrace{1+1+1+1+\cdots+1+1+1}^{n-1}$$

$$= (n-1)+(n-1)+(n-1)+\cdots+1\cdot(n-1)$$

$$= \overbrace{(n-1)+(n-1)+(n-1)+\cdots+(n-1)}^{\log_2 n+1}$$

$$= (n-1)(\log_2 n+1)$$

$$= (n-1)\log_2 n+(n-1) = O(n\log_2(n))$$