

Análisis y diseño de algoritmos: Proyecto final

Emilio E. G. Cantón Bermúdez y Roberto Gervacio Guendulay

09/05/2019

Resumen

El ordenamiento es uno de los bloques más básicos para la creación de algoritmos más complejos en los cuales precisamente 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)[<https://www.youtube.com/watch?v=B7hVxCmfPtM&t=2641s>], un *heap* es una implementación 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).

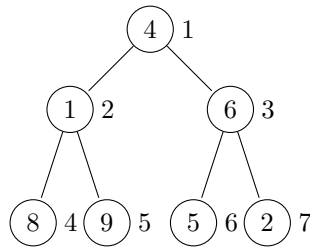


Fig 1.1: Representación de *heap*

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.

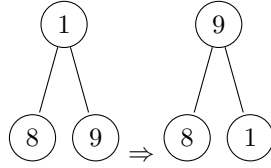


Fig. 1.2: Max-Heapify

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 uno de los hijos, entonces tenemos que asegurar que el hijo sigue siendo *max-heap*, por lo tanto, aplicamos recursivamente al hijo la operación y así sucesivamente.

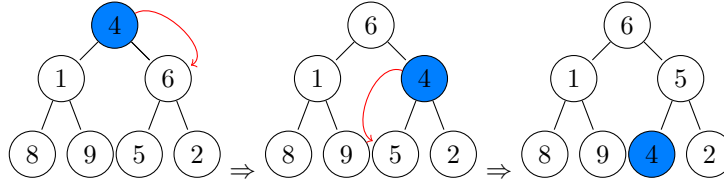


Fig 1.3: Max-Heapify nodo raíz

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** $[\frac{n}{2} + 1, \dots, n]$ **son hojas**. Nos interesan estos nodos en particular ya que al recibir el arreglo nuestra representación no es un *max-heap* por ahora, 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.

¹La 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 completo² sabemos que el número de nodos a partir de la raíz se puede representar por la siguiente serie

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

De acuerdo con nuestras observaciones sobre el número máximo 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

$$n_0 = n + 1$$

$$\begin{aligned} & \frac{n_0}{4} \cdot 1 + \frac{n_0}{8} \cdot 2 + \frac{n_0}{16} \cdot 3 + \dots + \frac{n_0}{n_0/2} \cdot \left(\frac{n_0}{2} - 1\right) + \frac{n_0}{n_0} \cdot \log_2 n \\ &= (n_0) \left(\frac{1}{4} \cdot 1 + \frac{1}{8} \cdot 2 + \frac{1}{16} \cdot 3 + \dots + \frac{1}{n_0/2} \cdot \left(\frac{n_0}{2} - 1\right) + \frac{1}{n_0} \cdot \log_2 n \right) \\ &= \left(\frac{n_0}{4}\right) \left(1 \cdot 1 + \frac{1}{2} \cdot 2 + \frac{1}{4} \cdot 3 + \dots + \frac{1}{n_0/8} \cdot \left(\frac{n_0}{2} - 1\right) + \frac{1}{n_0/4} \cdot \log_2 n \right) \\ &= \left(\frac{n_0}{4}\right) \left(1 \cdot 1 + \frac{1}{2} \cdot 2 + \frac{1}{4} \cdot 3 + \dots + \frac{1}{n_0/8} \cdot \left(\frac{n_0}{2} - 1\right) + \frac{1}{n_0/4} \cdot \log_2 n \right) \end{aligned}$$

$$k = \log_2(n)$$

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

Con la prueba de proporción de [d'Alembert](#) podemos ver que la sumatoria converge

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

Dado que

$$\begin{aligned} (n)2^n + 2^{n+1} &< (n+1)2^{n+1} \\ (n)2^n &< (n+1)2^{n+1} - 2^{n+1} \end{aligned}$$

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

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

$$= O(n)$$

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 tuplas³, 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 .

$$\therefore \text{merge} = O(n)$$

Habiendo definido el procedimiento *merge* podemos comenzar a calcular la complejidad. Dividimos en dos⁴ recursivamente una n -tupla t de manera que nos quedamos con una tupla a y una tupla b tal que

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

$$b = (t_{\frac{n}{2}}, t_{\frac{n}{2}+1}, \dots, t_{n-1})$$

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 $(4, 2, 5, 3, 1)$

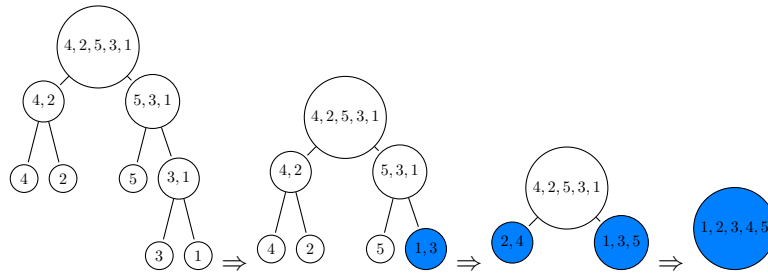


Fig 2.1: Ejemplo de Merge Sort

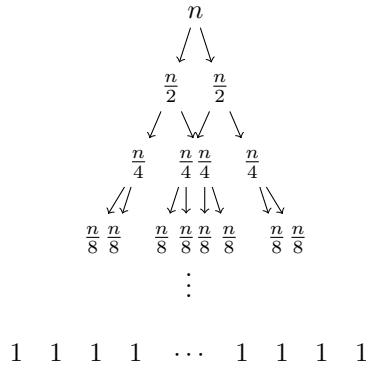
³ Definida matemáticamente como una lista ordenada de elementos (en este caso números)

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

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

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



Como podemos observar se forma las siguiente sumatoria

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