A Basic Comparison of Heap-Sort and Quick-Sort Algorithms

Merciadri Luca
Luca.Merciadri@student.ulg.ac.be

Abstract. We summarize here the biggest differences between Heap-sort and Quick-sort algorithms, two useful algorithms for array sorting.

Keywords: algorithms, sort, heap-sort, quick-sort.

1 Introduction

It is well known that, to learn the *complexity* (denoted by $\mathcal{O}$) of an algorithm, it is important to find a resource-costly operation. Here, we describe the complexity of the Heap-sort and Quick-sort algorithms, evidently depending upon the time T. These are useful algorithms for array sorting.

2 Heap-Sort

The *heap-sort* algorithm is an *in situ* algorithm: it does not need supplementary memory to work.

2.1 How It Works

The idea is to look at the array to sort as it was a *binary tree*. The first element is the root, the second is descending from the first element, ... A binary tree is drawn at Figure 1.

The aim is to obtain a *heap*, thus a binary tree verifying the following properties:

1. The maximal difference between the two leaves is 1 (all the leaves are on the last, or on the penultimate line);
2. The leaves having the maximum depth are "heaped" on the left;
3. Each node has a bigger (resp. lower) value than its of its two children, for an ascending (resp. descending) sort.

2.2 Complexity

The complexity of the heap-sort algorithm, for sorting a N elements array, is $\mathcal{O}(\beta N \cdot \log_2 N)$. The proof is given at Subsection 3.2, p. 2 assuming β is known.

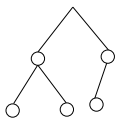

Fig. 1. Example of a binary tree.

3 Quick-sort

The *quick-sort* algorithm is not an *in situ* algorithm: it does need supplementary memory to work. This stack memory can vary between $\log_2 N$ and N. This algorithm is generally the most used one. We shall see after (see Subsection 3.3, p. I) why.

3.1 How It Works

The idea is to choose a *pivot* element, and to compare each element to this pivot element. If an element is inferior or equal to the pivot element, this element is put on the left of the pivot element. If the element is strictly superior to the pivot element, it is put on the right of the pivot element. This operation is called *partitioning*, and is recursively called until all the elements are sorted.

3.2 Complexity

The complexity of the quick-sort algorithm, for sorting a N elements array, is not always $\mathcal{O}(\alpha N \cdot \log_2 N)$ (assuming α is known). In fact, it can vary between $\alpha N \cdot \log_2 N$ and αN^2 (assuming α is known).

$\mathcal{O}(N \cdot \log_2 N)$ The complexity of the quick-sort algorithm is essentially $\mathcal{O}(N \cdot \log_2 N)$ when the array's elements are not almost all sorted (assuming the pivot element is the first one of the array). The complexity is the same (close to a $\frac{\beta}{\alpha}$ factor) for the heap-sort algorithm.

Proof. Let t_q, N and t_h denote respectively the time for the quick-sort algorithm if the elements have not already been sorted, the number of elements of the array, and the time for the heap-sort algorithm, whatever the case. We then have, knowing that $t_\mathrm{q} = t_\mathrm{h}$, and considering[1] each rearrangement gives two subzones of identical sizes:

$$\begin{aligned}
t_\mathrm{q} &= \alpha N + 2t_\mathrm{q}\left(\frac{N}{2}\right) \\
&= \alpha N + 2\left(\frac{\alpha N}{2} + 2t_\mathrm{q}\left(\frac{N}{4}\right)\right) \\
&= \alpha N + \alpha N + 4t_\mathrm{q}\left(\frac{N}{4}\right) \\
&= \alpha N + \alpha N + 4\left(\frac{\alpha N}{4} + 2t_\mathrm{q}\left(\frac{N}{8}\right)\right) \\
&\vdots \\
&= \alpha N + \alpha N + \cdots + \alpha N + \underbrace{2^x t_\mathrm{q}\left(\frac{N}{2^x}\right)}_{=0} \\
&= \underbrace{\alpha N + \alpha N + \cdots + \alpha N}_{x \text{ times with } x=\log_2(2^x)=\log_2(N)} \\
&= \alpha N \log_2(N),
\end{aligned}$$

where $N = 2^x$. □

[1] In fact, each rearrangement gives $\frac{N-1}{2}$-sized subzones, but we here want to find a magnitude, and we consider $\frac{N}{2} \approx \frac{N-1}{2}$.

$\boldsymbol{\mathcal{O}\left(N^2\right)}$ The complexity of the quick-sort algorithm is essentially $\mathcal{O}\left(N^2\right)$ when the array's elements are almost all sorted (assuming the pivot element is the first one of the array).

Proof. Let t_{q} denote the time for the quick-sort algorithm if the elements have already been sorted. We then have

$$\begin{aligned} t_{\mathrm{q}} &= \sum_{i=1}^{N-1} \underbrace{\alpha\left(N-i\right)}_{\text{at the arrangement } i} \\ &= \alpha(N-1) + \alpha(N-2) + \cdots + \alpha \\ &= \alpha\left(\frac{N^2-N}{2}\right) \\ &\simeq \alpha \cdot N^2. \end{aligned}$$

□

Remark 1. For another pivot choice, there is an initial disposition of the array's element which lands to the unfavorable case.

3.3 Why Is It Quick, Then

One can ask why the quick-sort is the most used algorithm if its complexity has a minimum of $N \cdot \log_2 N$, which is the same as the quick-sort. In fact, when both algorithms have same complexity ($\alpha N \cdot \log_2 N$ for the quick-sort, and $\beta N \cdot \log_2 N$ for the heap-sort), the quick-sort is faster because he has a proportionnality coefficient which equals the half of the heap-sort's proportionnality coefficient; mathematically, we have

$$\alpha = \frac{\beta}{2}.$$

4 Which One to Choose

There is no "ideal" solution about arrays' sorting. If you want to predict precisely when your algorithm will have finished, it is better to use the heap-sort algorithm, because the complexity of this one is always the same. It is also the case if you know *in advance* that your array is almost sorted.

References

1. de Marneffe, Pierre-Arnoul: Introduction à l'algorithmique (1998) ULg.
2. Cormen, Thomas H. and Leiserson, Charles E. and Rivest, Ronald L. and Stein, Clifford: Introduction to Algorithms. McGraw-Hill Book Company (2001)