Object-Oriented Sorting

Dung ("Zung") Nguyen, Stephen B. Wong
Dept. of Computer Science
Rice University
Houston, TX 77251
dxnguyen@cs.rice.edu, swong@cs.rice.edu

Figure 1 below shows the recursive call tree for the sort() method of a hypothetical sort algorithm.

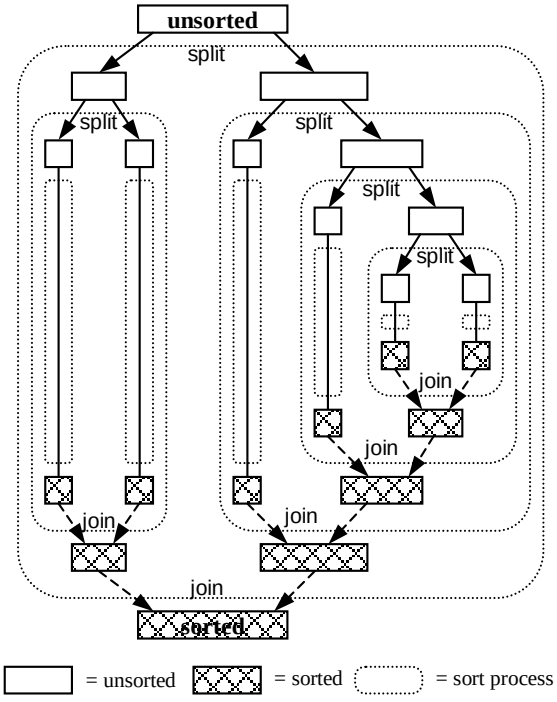

Figure 1: Hypothetical Sort Recursion Tree.

Table 1 below summarizes the split/join operations of a few common sort algorithms.

Sort	**Split operation**	**Join operation**
Insertion	Return *hi*	Insert *A*[*hi*] into proper location.
Merge	Return midpoint index.	Merge subarrays.
Quick	Find and return pivot point index	Do nothing.
Selection	Swap extremum with *A*[*hi*] and return *hi*	Do nothing.
Bubble	Bubble up extremum to *A*[*hi*] and return *hi*	Do nothing.
Heap	Swap extremum (*A*[*lo*]) and *A*[*hi*], reheapify *A*[*lo*, *hi*-1], and return *hi*.	Do nothing.

Table 1: Concrete split/join Operations

From **Table 1**, we can see that selection sort, bubble sort, and heap sort[1] are essentially identical processes, though they have different algorithmic complexities: they all pull out the extremum from the array and split it off. A trivial no-op join then follows this. Quick sort is similar to the selection/bubble/heap genera except that it pulls off a set of one or more extrema values.

On the flip side of the coin, we see that insertion sort and merge sort are similar in that their split operations are trivial while their join operations are more complex. Insertion splits off one element at a time while merge sort splits the array in half each time. One can think of the join() method in insertion sort as merging a sorted array with a one-element array (which is obviously sorted).

[1] Heap sort heapifies the array only once at construction time.

Complexity Analysis

On one hand, the sort template method engenders a recursion tree (see **Figure 1**), which provides some heuristics on the sort complexity. On the other hand, it leads to a canonical recurrence relation that serves as a common starting point for the analysis of each of the concrete sort algorithms.

It is easy to see from **Figure 1** that the total running time of a sort is equal to the sum of the running time of each level of the recursion sort tree. If the running time at each level is uniformly bounded by some function *f(n)*, then the total running time is bounded by *f(n)* times the height of the sort tree.

A formal treatment of complexity involves deriving a recurrence relation for *T*(*lo, hi*), the running time to sort an array *A*[*lo..hi*] indexed from *lo* to *hi* with *lo* <= *hi*. The code for sort() clearly indicates that

R1: $T(l, h) = \begin{cases} c \text{ if } l = h \\ c + S(l,h) + T(l,s\text{-}1) + T(s,h) + J(l,s,h) \text{ if } l < h \end{cases}$

where *c* is the constant running time to compare *lo* with *hi*, *S(lo, hi)* is the running time to split *A* into two subarrays, *A*[*lo..s*-1] and *A*[*s, hi*], and *J*[*lo, s, hi*] is the running time for joining the two sorted subarrays *A*[*lo..s*-1] and *A*[*s..hi*] to form the sorted array *A*[*lo..hi*].

It is necessary to examine the code for the specific split() and join() methods of a particular sort algorithm to compute *S*[*lo, hi*] and *J*[*lo, s, hi*] in order to solve R1. Let *n* denote the size of the array. The steps in the computation of *T*(*n*) are identical for all of the sort algorithms: start with the canonical relation **R1**, plug in the values for *s*, *S*[*lo, hi*], and *J*[*lo, s, hi*], and simplify. Note that the functional form of *s* may depend on whether one sorts from *lo* to *hi* or *hi* to *lo*. The simplification will lead to one of the following two recurrence relations:

R2: $T(n) = \begin{cases} O(1) \text{ if } n = 1, \\ T(n\text{-}1) + O(f(n)) \text{ if } n > 1 \end{cases}$

R3: $T(n) = \begin{cases} O(1) \text{ if } n = 1, \\ aT(n/b) + O(f(n)) \text{ if } n > 1 \end{cases}$

R2 and **R3** can then be solved using the same standard discrete mathematics technique yielding the results shown in **Table 2** below.

Sort	s	S[lo, hi]	J[lo, s, hi]	T(n)
Insertion	hi	O(1)	O(hi-lo)	$O(n^2)$
Merge	(lo+hi+1)/2	O(1)	O(hi-lo)	O(n log n)
Quick	varies	O(hi-lo)	O(1)	$O(n^2)$ worst case
Selection	lo + 1	O(hi-lo)	O(1)	$O(n^2)$
Bubble	hi	O(hi-lo)	O(1)	$O(n^2)$
Heap	hi	O(log(hi-lo))	O(1)	O(n log n)

Table 2: Running Time for Sorting