Sorting Algorithms — Taking Advantage of Order Present

When we analyze the *expected* running time of sorting a algorithm, we generally assume

i) All array elements are inequivalent (all keys are distinct).

In practice, this is often not the case. What happens then? The algorithm probably works at least as well, but the analysis becomes quite difficult.

ii) The array elements appear in "random" order. (Each of the $n!$ alternatives for the sorting permutation is equally likely.)

Again, this may well not be true. What if it fails? We consider this below.

In some circumstances, the input to the sorting algorithm is likely to be "*almost in order*".

Perhaps we started with a sorted array, and performed a few transactions that disturbed the order. Now we need to restore the array to sorted order.

The array is not in sorted order, but it is scarcely random. There is a good deal of order present. Assumption (ii) above doesn't apply.

For example, the input might look like this:

12 25 17 22 29 43 34 39 45 49 58 53 63 68 79 82 87 77 91

Ideally, a sorting algorithm would take advantage of order present, so that an array "almost in order" could be sorted faster than a random array.

Many sorting algorithms don't do this (or do it only to a small extent).

With quicksort (implemented in the simplest possible way), something far more drastic occurs: An input "almost in order" produces a *worst or near-worst case*.

What does it mean for there to be "order present" in an array. To make this precise, we need the concept of an *inversion*.

If A is an array of size n, an **inversion** of A is an ordered pair (i,j) such that

$$\boldsymbol{i < j} \ \textbf{and} \ \boldsymbol{A[i] > A[j]} \qquad (1 \le i, j \le n),$$

i.e., a pair that is out-of-order.

	1	2	3	4	5	6
A	**27**	**12**	**45**	**31**	**23**	**25**

Inversions of A: (1,2), (1,5), (1,6), (3,4), (3,5), (3,6), (4,5), (4,6)

Let $I(A)$ denote the number of inversions of A.

Obviously $\mathbf{0 \le \boldsymbol{I(A)} \le \boldsymbol{n(n}–1)/2}$. (Only $n(n–1)/2$ pairs exist.)

$I(A) = 0$	corresponds to A being in sorted order.
$I(A) = n(n–1)/2$	corresponds to A being in reverse order.
$I(a) = n(n–1)/4$	(expected) for A being in random order.

What does it mean for there to be *order present* in an array A?

A reasonable definition would be: $\boldsymbol{I(A) \ll n(n–1)/4}$.

In particular, we are justified in calling an array *almost in order* if $\boldsymbol{I(A) = O(n)}$.

What does it mean for a sorting algorithm to *take advantage of order present*?

No sorting algorithm can perform fewer than $n–1$ comparisons, even if the input is exactly in sorted order. (Merely verifying that the array is sorted requires $n–1$ comparisons.)

We are certainly justified in saying that a sorting algorithm takes advantage of order present if the number of comparisons $C(n)$ is $\boldsymbol{O(n + I(A))}$.

In particular, the running time is linear in n if the number of inversions is linear.

Under this criterion, *straight insertion sort* qualifies:

$$C(n) = n–1+I(A).$$

Most other sorting algorithms don't qualify under this strict criterion, but some of them benefit, to a lesser extent, from order present in the input.

Note (not specifically related to discussion of order present, above):

i) If $A[i] > A[i+1]$, exchanging $A[i]$ and $A[i+1]$ removes *exactly one* inversion from A. (It removes inversion $(i,i+1)$.)

ii) If $A[i] > A[j]$, where $j > i$, exchanging $A[i]$ and $A[j]$ may remove as many as $1 + 2(j–i–1)$ inversions from A. (This is an upper bound; it may remove fewer.)

Implication: Any sorting algorithm that rearranges its input by exchanging adjacent elements (e.g., bubble sort, straight insertion sort) cannot have a worst-case running time better than $\Theta(n^2)$.

This limitation doesn't apply to algorithms that exchange non-adjacent elements, e.g., *quicksort* and *heapsort*.