

Let us start with a *best-case* assumption: that the partitioning step leaves the pivot in the middle of the array. Then n' and $n - n' + 1$ are approximately $n/2$, and we get:

$$\text{comps}(n) \approx n + 2 \text{comps}(n/2) \quad \text{if } n > 1 \quad (3.15a)$$

$$\text{comps}(n) = 0 \quad \text{if } n \leq 1 \quad (3.15b)$$

The solution (see Appendix A.4) is approximately:

$$\text{comps}(n) \approx n \log_2 n \quad (3.16)$$

So, on the best-case assumption, the quick-sort algorithm has time complexity $O(n \log n)$. The best-case assumption turns out to be valid if the array is thoroughly unsorted.

Now let us consider a *worst-case* assumption: that the partitioning step leaves the pivot at the left of the array. Then $n' = 0$ and $\text{comps}(n') = \text{comps}(0) = 0$, so we get:

$$\text{comps}(n) \approx n + \text{comps}(n - 1) \quad \text{if } n > 1 \quad (3.17a)$$

$$\text{comps}(n) = 0 \quad \text{if } n \leq 1 \quad (3.17b)$$

The solution (see Appendix A.4) is approximately:

$$\text{comps}(n) \approx n^2/2 \quad (3.18)$$

So, on the worst-case assumption, the quick-sort algorithm has time complexity $O(n^2)$. The worst-case assumption turns out to be valid if the array is already sorted (or nearly sorted). This is very strange behavior indeed; intuitively we would expect a sorting algorithm to take *less* time when the given array turns out to be already sorted!

Of course, we would never sort an array *known* to be already sorted. But if we are given an unknown array, we do not know whether it is sorted, nearly sorted, or thoroughly unsorted. There is only a tiny probability that an unknown array will happen to be sorted, purely by chance — just as there is only a tiny probability that a shuffled pack of playing cards will happen to be sorted. But in practical data processing things are often not left to chance, and there is quite a high probability that an unknown array will turn out to be sorted (or nearly sorted).

For example, the telephone supervisor responsible for maintaining a business's internal telephone directory might be given a batch of new entries by an assistant. The best way for the supervisor to update the directory is to sort the new entries and then merge them into the directory. But what if the assistant, intending to be helpful, has already sorted the new entries?

If the array is already sorted, $a[\textit{left}]$ will be the least of $a[\textit{left} \dots \textit{right}]$. But Figure 3.40 chooses the value of $a[\textit{left}]$ as the pivot, and that pivot will remain at the left end of the array, which is our worst-case assumption. To avoid this situation, the partitioning algorithm should try to choose a pivot that is less likely to be the least value in the array.

One simple idea is to modify Algorithm 3.41 to choose the pivot from the middle of the array:

1. Let *pivot* be the value of $a[m]$ (where m is about midway between *left* and *right*), swap *pivot* into $a[\textit{left}]$, and set $p = \textit{left}$.