Algorithm inPlacePartition(S, a, b):
Input: An array, S, of distinct elements; integers a and b such that $a \leq b$
Output: An integer, l, such that the subarray $S[a\,..\,b]$ is partitioned into $S[a..l-1]$ and $S[l..b]$ so that every element in $S[a..l-1]$ is less than each element in $S[l..b]$

```
Let r be a random integer in the range [a, b]
Swap S[r] and S[b]
p ← S[b]        // the pivot
l ← a           // l will scan rightward
r ← b − 1       // r will scan leftward
while l ≤ r do  // find an element larger than the pivot
    while l ≤ r and S[l] ≤ p do
        l ← l + 1
    while r ≥ l and S[r] ≥ p do   // find an element smaller than the pivot
        r ← r − 1
    if l < r then
        Swap S[l] and S[r]
Swap S[l] and S[b]        // put the pivot into its final place
return l
```

Algorithm inPlaceQuickSort(S, a, b):
Input: An array, S, of distinct elements; integers a and b
Output: The subarray $S[a\,..\,b]$ arranged in nondecreasing order

```
if a ≥ b then return        // subrange with 0 or 1 elements
l ← inPlacePartition(S, a, b)
inPlaceQuickSort(S, a, l − 1)
inPlaceQuickSort(S, l + 1, b)
```

Algorithm 8.9: In-place randomized quick-sort for an array, S.

Dealing with the Recursion Stack

Actually, the above description of quick-sort is not quite in-place, as it could, in the worst case, require a linear amount of additional space besides the input array. Of course, we are using no additional space for the subsequences, and we are using only a constant amount of additional space for local variables (such as l and r).

So, where does this additional space come from?

It comes from the recursion, since we need space for a stack proportional to the depth of the recursion tree in order to keep track of the recursive calls for quick-sort. This stack can become as deep as $\Theta(n)$, in fact, if we have a series of bad pivots, since we need to have a method frame for every active call when we make the call for the deepest node in the quick-sort tree.