Quick Sort—An In-Place Implementation

Yufei Tao

Department of Computer Science and Engineering
Chinese University of Hong Kong
We talked about quick sort, which finishes in $O(n^2)$ worst case time, and $O(n \log n)$ expected time. This does not seem attractive at all theoretically, given that merge sort can do $O(n \log n)$ in the worst case.

Nevertheless, quick sort is really quick in practice. An important reason is that it allows a simple yet fast “in-place” implementation which reduces the hidden constant in its $O(n \log n)$ complexity. By in-place, we mean that the sorting can be performed entirely in the input array, thus removing the overhead of creating another array and copying the elements back and forth.
Recall:

The Sorting Problem

Problem Input:

A set $S$ of $n$ integers is given in an array of length $n$.

Goal:

Design an algorithm to store $S$ in an array where the elements have been arranged in ascending order.
Recall:

**Quick Sort**

We will denote the input array as \( A \), and describe the algorithm by recursion.

**Base Case.** If \( n = 1 \), return directly.

**Reduce.** Otherwise, the algorithm runs the following steps:

1. **Randomly** pick an integer \( p \) in \( A \)—call it the **pivot**.
   - This can be done in \( O(1) \) time using \text{RANDOM}(1, n).

2. Re-arrange the integers in an array \( A' \) such that
   - All the integers **smaller** than \( p \) are positioned **before** \( p \) in \( A' \).
   - All the integers **larger** than \( p \) are positioned **after** \( p \) in \( A' \).

3. Sort the part of \( A' \) before \( p \) recursively.

4. Sort the part of \( A' \) after \( p \) recursively.
Example

After Step 1 (suppose that 26 was randomly picked as the pivot):

\[
\begin{array}{cccccccccccccccc}
5 & 9 & 12 & 17 & 26 & 41 & 72 & 83 & 69 & 20 & 12 & 68 & 5 & 52 & 35 & 9 \\
\end{array}
\]

After Step 2:

\[
\begin{array}{cccccccccccccccc}
9 & 5 & 12 & 17 & 20 & 26 & 72 & 83 & 69 & 41 & 88 & 68 & 28 & 52 & 35 & 38 \\
\end{array}
\]

After Steps 3 and 4:

\[
\begin{array}{cccccccccccccccc}
5 & 9 & 12 & 17 & 20 & 26 & 28 & 35 & 38 & 41 & 52 & 68 & 69 & 72 & 83 & 88 \\
\end{array}
\]

We will discuss how to perform Step 2.
Quick Sort—Step 2 (Distributing)

We have an array $A$, and a pivot $v$ stored at $A[p]$. We want to move every element smaller (or larger) than $v$ to the left (or right, resp.) of $v$. 
Quick Sort—Step 2 (Distributing)

Record $v$ separately and erase $A[p]$ (now there is a “gap” at $A[p]$). At any moment, maintain pointers $i, j$. In the outset, $i = 1, j = n$.

- Keep moving $i$ to the right until $i = p$ or $A[i] \geq v$
- Keep moving $j$ to the left until $j = p$ or $A[j] \leq v$
- If neither $i$ nor $j$ is at $p$, swap $A[i], A[j]$, and repeat.

When $i$ or $j$ is $p$, we enter a second phase as explained on the next slide.
Quick Sort—Step 2 (Distributing)

Now either $i$ or $j$ is pointing to a gap.

If $i$ has the gap:

- Move $j$ to the left until $j = i$ or $A[j] \leq v$.

If $j$ has the gap:

- Move $i$ to the right until $i = j$ or $A[i] < v$.

When $i = j$, fill in $A[i] = v$ and finish.
Yufei Tao  Quick Sort—An In-Place Implementation