An In-Place Implementation of Quick Sort

By Yufei Tao's Teaching Team

We have learned that quick sort guarantees running time $O(n \log n)$ in expectation. Today, we will discuss how to implement it in an **"in-place"** manner.

> In general, an implementation is said to be **in-place** if it uses **exactly** n memory cells, namely, just enough to store the input array.

Recall:

The Sorting Problem. The input is an array A of n distinct integers. The goal is to output an array where the n integers are stored in ascending order.

Recall: The Quick Sort Algorithm

1. Pick an integer p from A **uniformly at random**, which is called the **pivot**.
2. Store the integers in another array A' such that
   - all the integers smaller than p are before p in A';
   - all the integers larger than p are after p in A'.
3. Sort the part of A' before p recursively (a subproblem).
4. Sort the part of A' after p recursively (a subproblem).

Example

Original array A (suppose that 26 was randomly picked as the pivot):

				p																									
38	28	88	17	26	41	72	83	20	47	12	68	5	52	35	9														

Step 2 creates another array A':

					p																								
17	20	12	5	9	26	38	28	88	41	72	83	47	68	52	35														

Creation of A' requires reading and writing n integers.

An In-Place Version of Quick Sort

1. Pick a pivot p from A uniformly at random.
2. Re-arrange the integers in A such that
   - all the integers smaller than p are before p in A;
   - all the integers larger than p are after p in A.
3. Sort the part of A before p recursively (a subproblem).
4. Sort the part of A after p recursively (a subproblem).

Next, we will explain how to implement Step 2 in $O(n)$ time without using any extra memory cells (other than those in A). The implementation uses only $O(1)$ CPU registers.

First, store the pivot $p = 26$ in a CPU register. This creates an empty slot, which we refer to as the **gap**.

				gap											
38	28	88	17		41	72	83	20	47	12	68	5	52	35	9

Set pointers $i = 1$ and $j = n = 16$. The values of i and j are in CPU registers.

i															j
38	28	88	17		41	72	83	20	47	12	68	5	52	35	9

Here, $A[i] > p = 26$ and $A[j] < p$. We swap $A[i]$ with $A[j]$, which gives:

i															j
9	28	88	17		41	72	83	20	47	12	68	5	52	35	38

Increase i until $A[i] > p = 26$ and decrease j until $A[j] < p$:

	i											j			
9	28	88	17		41	72	83	20	47	12	68	5	52	35	38

Swapping $A[i]$ with $A[j]$ gives:

	i											j			
9	5	88	17		41	72	83	20	47	12	68	28	52	35	38

Again, increase i until $A[i] > p$ and decrease j until $A[j] < p$:

		i								j					
9	5	88	17		41	72	83	20	47	12	68	28	52	35	38

Swapping $A[i]$ with $A[j]$ gives:

		i								j					
9	5	12	17		41	72	83	20	47	88	68	28	52	35	38

Again, we try to increase i to find the next $A[i] > p = 26$. However, this time i hits the gap before such an $A[i]$ is found:

				i						j					
9	5	12	17		41	72	83	20	47	88	68	28	52	35	38

Keeping i at the gap, we now decrease j to find the next $A[j] < p$:

				i				j							
9	5	12	17		41	72	83	20	47	88	68	28	52	35	38

Swapping $A[i]$ with $A[j]$ gives:

				i				j							
9	5	12	17	20	41	72	83		47	88	68	28	52	35	38

Note: j points to the gap now.

Keeping j at the gap, we now increase i to find the next $A[i] > p = 26$:

					i			j							
9	5	12	17	20	41	72	83		47	88	68	28	52	35	38

Swapping $A[i]$ with $A[j]$ gives:

					i			j							
9	5	12	17	20		72	83	41	47	88	68	28	52	35	38

Note: i points to the gap now.

Keeping i at the gap, we try to decrease j to find the next $A[j] < p = 26$. However, this time j hits the gap before such an $A[j]$ is found:

					ij										
9	5	12	17	20		72	83	41	47	88	68	28	52	35	38

As both i and j point to the gap, we now finish by entering p into the gap:

					ij										
9	5	12	17	20	26	72	83	41	47	88	68	28	52	35	38

The in-place implementation has at least two advantages over our old implementation:

- It uses less memory.
- It may perform less memory writes (think: why?).

Owing to these advantages, quick sort usually outperforms merge sort in practice, even though their time complexities are both $O(n \log n)$.