# Lecture Notes for Chapter 7: Quicksort

## Chapter 7 overview

*[The treatment in the second edition differs from that of the first edition. We use a different partitioning method—known as "Lomuto partitioning"—in the second edition, rather than the "Hoare partitioning" used in the first edition. Using Lomuto partitioning helps simplify the analysis, which uses indicator random variables in the second edition.]*

### Quicksort

- Worst-case running time: $\Theta(n^2)$.
- Expected running time: $\Theta(n \lg n)$.
- Constants hidden in $\Theta(n \lg n)$ are small.
- Sorts in place.

## Description of quicksort

Quicksort is based on the three-step process of divide-and-conquer.

- To sort the subarray $A[p \,..\, r]$:

  **Divide:** Partition $A[p \,..\, r]$, into two (possibly empty) subarrays $A[p \,..\, q-1]$ and $A[q+1 \,..\, r]$, such that each element in the first subarray $A[p \,..\, q-1]$ is $\leq A[q]$ and $A[q]$ is $\leq$ each element in the second subarray $A[q+1 \,..\, r]$.

  **Conquer:** Sort the two subarrays by recursive calls to QUICKSORT.

  **Combine:** No work is needed to combine the subarrays, because they are sorted in place.

- Perform the divide step by a procedure PARTITION, which returns the index $q$ that marks the position separating the subarrays.

```
QUICKSORT($A, p, r$)
if $p < r$
  then $q \leftarrow$ PARTITION($A, p, r$)
       QUICKSORT($A, p, q - 1$)
       QUICKSORT($A, q + 1, r$)
```

Initial call is QUICKSORT$(A, 1, n)$.

## Partitioning

Partition subarray $A[p \mathinner{..} r]$ by the following procedure:

```
PARTITION($A, p, r$)
$x \leftarrow A[r]$
$i \leftarrow p - 1$
for $j \leftarrow p$ to $r - 1$
    do if $A[j] \le x$
         then $i \leftarrow i + 1$
              exchange $A[i] \leftrightarrow A[j]$
exchange $A[i + 1] \leftrightarrow A[r]$
return $i + 1$
```

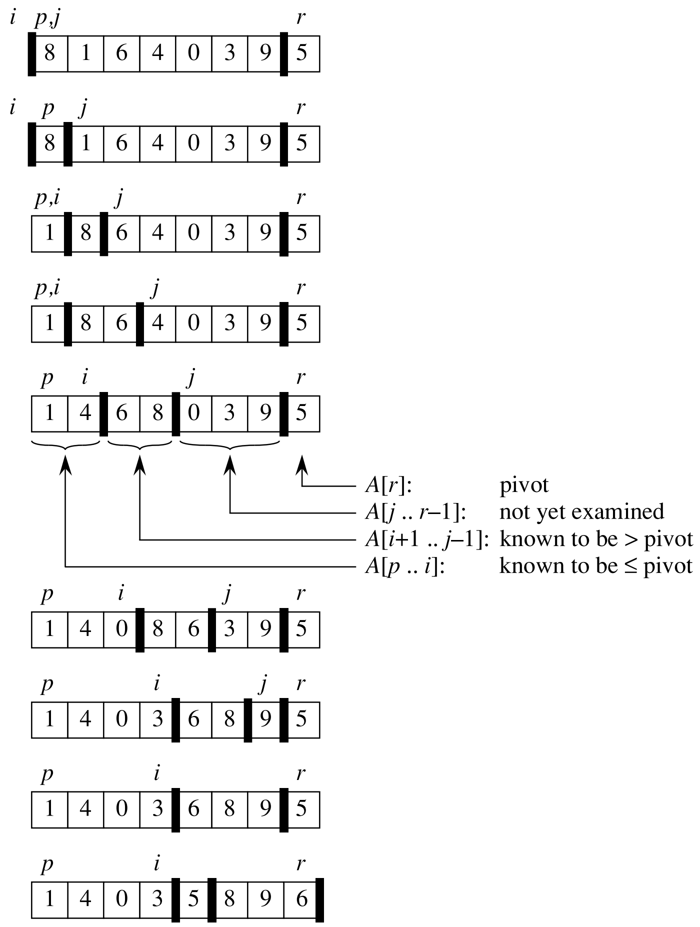
- PARTITION always selects the last element $A[r]$ in the subarray $A[p \mathinner{..} r]$ as the ***pivot***—the element around which to partition.
- As the procedure executes, the array is partitioned into four regions, some of which may be empty:

  **Loop invariant:**

  1. All entries in $A[p \mathinner{..} i]$ are $\le$ pivot.
  2. All entries in $A[i + 1 \mathinner{..} j - 1]$ are $>$ pivot.
  3. $A[r] =$ pivot.

  It's not needed as part of the loop invariant, but the fourth region is $A[j \mathinner{..} r-1]$, whose entries have not yet been examined, and so we don't know how they compare to the pivot.

***Example:*** On an 8-element subarray.



*[The index $j$ disappears because it is no longer needed once the **for** loop is exited.]*

***Correctness:*** Use the loop invariant to prove correctness of PARTITION:

**Initialization:** Before the loop starts, all the conditions of the loop invariant are satisfied, because $r$ is the pivot and the subarrays $A[p \,..\, i]$ and $A[i+1 \,..\, j-1]$ are empty.

**Maintenance:** While the loop is running, if $A[j] \leq$ pivot, then $A[j]$ and $A[i+1]$ are swapped and then $i$ and $j$ are incremented. If $A[j] >$ pivot, then increment only $j$.

**Termination:** When the loop terminates, $j = r$, so all elements in $A$ are partitioned into one of the three cases: $A[p \,..\, i] \leq$ pivot, $A[i+1 \,..\, r-1] >$ pivot, and $A[r] =$ pivot.

The last two lines of PARTITION move the pivot element from the end of the array to between the two subarrays. This is done by swapping the pivot and the first element of the second subarray, i.e., by swapping $A[i+1]$ and $A[r]$.

***Time for partitioning:*** $\Theta(n)$ to partition an $n$-element subarray.

## Performance of quicksort

The running time of quicksort depends on the partitioning of the subarrays:

- If the subarrays are balanced, then quicksort can run as fast as mergesort.
- If they are unbalanced, then quicksort can run as slowly as insertion sort.

### Worst case

- Occurs when the subarrays are completely unbalanced.
- Have 0 elements in one subarray and $n - 1$ elements in the other subarray.
- Get the recurrence
$$\begin{aligned} T(n) &= T(n-1) + T(0) + \Theta(n) \\ &= T(n-1) + \Theta(n) \\ &= \Theta(n^2) . \end{aligned}$$
- Same running time as insertion sort.
- In fact, the worst-case running time occurs when quicksort takes a sorted array as input, but insertion sort runs in $O(n)$ time in this case.

### Best case

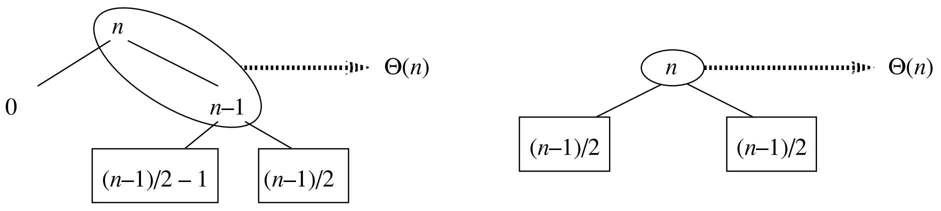
- Occurs when the subarrays are completely balanced every time.
- Each subarray has $\le n/2$ elements.
- Get the recurrence
$$\begin{aligned} T(n) &= 2T(n/2) + \Theta(n) \\ &= \Theta(n \lg n) . \end{aligned}$$

### Balanced partitioning

- Quicksort's average running time is much closer to the best case than to the worst case.
- Imagine that PARTITION always produces a 9-to-1 split.
- Get the recurrence
$$\begin{aligned} T(n) &\le T(9n/10) + T(n/10) + \Theta(n) \\ &= O(n \lg n) . \end{aligned}$$
- Intuition: look at the recursion tree.
  - It's like the one for $T(n) = T(n/3) + T(2n/3) + O(n)$ in Section 4.2.
  - Except that here the constants are different; we get $\log_{10} n$ full levels and $\log_{10/9} n$ levels that are nonempty.
  - As long as it's a constant, the base of the log doesn't matter in asymptotic notation.
  - Any split of constant proportionality will yield a recursion tree of depth $\Theta(\lg n)$.

**Intuition for the average case**

- Splits in the recursion tree will not always be constant.
- There will usually be a mix of good and bad splits throughout the recursion tree.
- To see that this doesn't affect the asymptotic running time of quicksort, assume that levels alternate between best-case and worst-case splits.



- The extra level in the left-hand figure only adds to the constant hidden in the $\Theta$-notation.
- There are still the same number of subarrays to sort, and only twice as much work was done to get to that point.
- Both figures result in $O(n \lg n)$ time, though the constant for the figure on the left is higher than that of the figure on the right.

## Randomized version of quicksort

- We have assumed that all input permutations are equally likely.
- This is not always true.
- To correct this, we add randomization to quicksort.
- We could randomly permute the input array.
- Instead, we use ***random sampling***, or picking one element at random.
- Don't always use $A[r]$ as the pivot. Instead, randomly pick an element from the subarray that is being sorted.

We add this randomization by not always using $A[r]$ as the pivot, but instead randomly picking an element from the subarray that is being sorted.

```
RANDOMIZED-PARTITION($A, p, r$)
$i \leftarrow$ RANDOM($p, r$)
exchange $A[r] \leftrightarrow A[i]$
return PARTITION($A, p, r$)
```

Randomly selecting the pivot element will, on average, cause the split of the input array to be reasonably well balanced.