Quicksort (Ch. 7)

**Input:** An n-element array A (unsorted).
**Output:** An n-element array A in non-decreasing order.

Quicksort(A, p, r)

1. If p < r
2. q = Partition(A, p, r)
3. Quicksort(A, p, q-1)
4. Quicksort(A, q+1, r)

Partition(A, p, r)

1. x = A[r] // choose pivot
2. i = p - 1
3. for j = p to r - 1
4. if A[j] <= x
5. i = i + 1
6. swap A[i] and A[j]
7. swap A[i+1] and A[r]
8. return i + 1

Initial call:
Quicksort(A, 1, A.length)

---

Quicksort (Ch. 7)

**Input:** An n-element array A (unsorted).
**Output:** An n-element array A in non-decreasing order.

Quicksort(A, p, r)

1. If p < r
2. q = Partition(A, p, r)
3. Quicksort(A, p, q-1)
4. Quicksort(A, q+1, r)

Partition(A, p, r)

1. x = A[r] // choose pivot
2. i = p - 1
3. for j = p to r - 1
4. if A[j] <= x
5. i = i + 1
6. swap A[i] and A[j]
7. swap A[i+1] and A[r]
8. return i + 1

Initial call:
Quicksort(A, 1, A.length)

---

Correctness of Quicksort

**Claim:** Partition satisfies the specifications of the Divide step.

**Loop Invariant:** At the beginning of each iteration of the for loop (lines 3-6), for any array index k,

1. If p <= k <= i, then A[k] <= x.
2. If i+1 <= k <= j-1, then A[k] > x.
3. If k <= i+1 and A[k] > x.

**Initialization:** i = p - 1 and j = p + 1. k cannot be between 0 and 1 (cond. 1).

**Inductive Hypothesis:** Assume the invariant holds through iteration j - i < n - 1.

**Inductive Step:** In iteration k+1, other A[j] > x or A[j] <= x. In the first case, j is incremented and cond. 2 holds for A[j] > x. In the second case, j is incremented, A[j] > x or A[j] <= x. In the first case, j is swapped, and then j is incremented. Cond. 1 holds for A[j] > x after swap. By the IH, item A[i] was in A[j] during the last iteration, and was > x then, so cond. 2 holds at the end of iteration k+1.

**Termination:** At termination, j = r and A has been partitioned into 3 sets: items <= x, items > x, and A[j] = x.

---

Quicksort Running Time

\[ T(n) = T(q - p) + T(r - q) + O(n) \]

The value of T(n) depends on the location of q in the array A[p..r]. Since we don’t know this in advance, we must look at worst-case, best-case, and average-case partitioning.

**Worst-case partitioning:** Each partition results in a 0 : n-1 split. T(0) = Θ(1) and the partitioning costs Θ(n), so recurrence is

\[ T(n) = T(n-1) + T(0) + Θ(n) = T(n-1) + Θ(n) \]

This is an arithmetic series which evaluates to \(Θ(n^2)\). So worst-case for Quicksort is no better than Insertion sort!

What does the input look like in Quicksort’s worst-case?
Quicksort Best-case

\[ T(n) = T(q - p) + T(r - q) + O(n) \]

Best-case partitioning: Each partition results in a \( \lceil n/2 \rceil : \lceil n/2 \rceil - 1 \) split (i.e., close to balanced split each time), so recurrence is 
\[ T(n) = 2T(n/2) + \theta(n) \]

By case 2 of the master theorem, this recurrence evaluates to \( \theta(n \log n) \)

---

Quicksort Average-case

Intuition: Some splits will be close to balanced and others close to unbalanced \( \Rightarrow \) good and bad splits will be randomly distributed in recursion tree.

The running time will be (asymptotically) bad only if there are many bad splits in a row.

- A bad split followed by a good split results in a good partitioning after one extra step.
- Implies a \( \theta(n \log n) \) average case running time (with a larger constant factor to ignore).

---

How can we modify Quicksort to get good average case behavior on all inputs?

2 techniques:

1. randomly permute input prior to running Quicksort. Will produce tree of possible executions, most of them finish fast.
2. choose partition randomly at each iteration instead of choosing element in highest array position.

Randomized-Partition(A, p, r)

1. i = Random(p, r)
3. return Partition(A, p, r)

In section 7.4, a probabilistic analysis is presented, showing that the expected running time of Randomized-Quicksort is \( O(n \log n) \)