QuickSort (Ch. 7)

A divide-and-conquer algorithm

Divide: Rearrange the array A[1..n] into two (one possibly empty) subarrays A[1..q-1] and A[q+1..n] such that each element of A[1..q-1] ≤ A[q] and each element of A[q+1..n] > A[q] after computation of index q.

Conquer: Sort the two subarrays A[1..q-1] and A[q+1..n] recursively.

Combine: No work is needed to combine subarrays since they are sorted in-place.

QuickSort

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)
```

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

```
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
```

Divide: Rearrange the array A[p..r] into two (one possibly empty) subarrays A[p..q-1] and A[q+1..r] such that each element of A[p..q-1] ≤ A[q] and each element of A[q+1..r] > A[q] after computation of index q.

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=r, then A[k] = x.

```
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
```

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=r, then A[k] = x.

```
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
```


Initialization: i = p-1 = 0 and j = p = 1. k cannot be between 0 and 1 (cond 1), nor can k be between i+1 = 1 and j - 1 = 0 (cond 2). Partition satisfies condition 3 in this case, and conditions 1 and 2 are vacuously true.

Inductive Hypothesis: Assume the invariant holds through iteration j = k < n -1.

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=r, then A[k] = x.

```
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
```


Ind. Step: At the start of iteration k+1, either A[j] > x or A[j] ≤ x. If A[j] > x, j is incremented and cond 2 holds for A[j-1] with no other changes. If A[j] ≤ x, A[i+1] and A[j] are swapped, and then j is incremented. Cond 1 holds for A[i] after swap by the IH. By the IH, the item in A[j] was in A[i+1] 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. The item in A[i+1] > x in the last iteration, so it remains > x at termination, when it is swapped with the element in A[r] = x. QED

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=r, then A[k] = x.

```
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
```


So at the end of the algorithm, A is split into 3 parts, from left to right: items in A[1...i] that are ≤ A[r], A[i+1] = A[r], and the items in A[i+2...n] that are > A[r].

Therefore, the item in position A[i+1] = A[q] is in its proper sorted position after the first iteration, Partition(A, 1, n). The Quick-Sort algorithm then makes recursive calls on A[1...q-1] and A[q+1...n].

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.

Quicksort Running Time

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

$$T(n) = T(n-1) + T(0) + \theta(n) = T(n-1) + \theta(n)$$

This is an arithmetic series which evaluates to $\theta(n^2)$. 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 $\lfloor n/2 \rfloor : \lceil n/2 \rceil$ -1 split (i.e., close to balanced split each time), so recurrence is

$$T(n) = 2T(n/2) + \theta(n)$$

By the master theorem, this recurrence evaluates to

$$\theta(n\lg n)$$

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 $\theta(n\lg n)$ running time (with a larger constant factor to ignore than if all splits were good).

Randomized Quicksort

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)
2. swap A[r] ↔ A[i]
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(nlgn)*

Sorting Algorithms

- A sorting algorithm is *comparison-based* if the only operation we can perform on keys is to compare them.
- A sorting algorithm is *in place* if only a constant number of elements of the input array are ever stored outside the array.

Running Time of Comparison-Based Sorting Algorithms

	worst-case	average-case	best-case	in place?
Insertion Sort	n^2	n^2	n	yes
Merge Sort	$n\lg n$	$n\lg n$	$n\lg n$	no
Heap Sort	$n\lg n$	$n\lg n$	$n\lg n$	yes
Quick Sort	n^2	$n\lg n$	$n\lg n$	yes