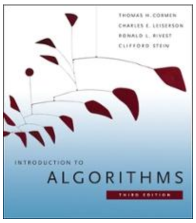
CS146 Data Structures and Algorithms



Chapter 7: Quicksort

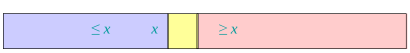
Quicksort

- Sorts in place
- Sorts O(n lg n) in the average case
- Sorts O(n^2) in the worst case
  - But in practice, it's quick
  - And the worst case doesn't happen often (but more on this later…)
  - Empirical and analytical studies show that quicksort can be ***expected*** to be twice as fast as its competitors.

7.1 Description of Quicksort

Quicksort an *n*-element array:

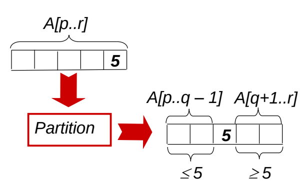
- *Divide:* Partition the array into two subarrays around a *pivot x* such that elements in lower subarray $\le x \le$ elements in upper subarray.



- *Conquer:* Recursively sort the two subarrays.
- *Combine:* The subarrays are sorted in place – no work is needed to combine them.
- How do the divide and combine steps of quicksort compare with those of merge sort?

Design

- Key: Linear-time partitioning subroutine.
- ***Divide*:** Partition (separate) the array $A[p..r]$ into two (possibly empty) subarrays $A[p..q-1]$ and $A[q+1..r]$.
  - Each element in $A[p..q-1] \le A[q]$.
  - $A[q] \le$ each element in $A[q+1..r]$.
  - Index q is computed as part of the partitioning procedure.



```
QUICKSORT(A, p, r)
if p < r
      Q = PARTITION(A, p, r)
      QUICKSORT(A, p, q-1)
      QUICKSORT(A, q+1, r)
```

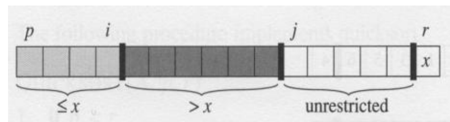
Initial call: QUICKSORT(*A*, 1, *n*)

L7.5

Loop Invariant

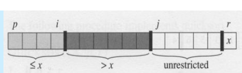
Loop Invariant

1. All entries in A[p..i] are ≤ pivot.
2. All entries in A[i+1..j-1] are > pivot.
3. A[r] = pivot.



L7.6

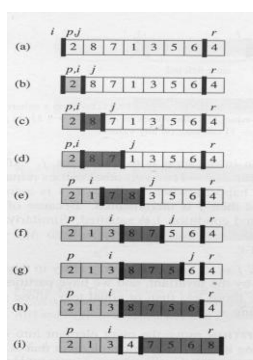
Partition(A, p, r)



```
x = A[r]
i  = p – 1
for j = p to r -1
  if A[j] ≤ x
      i = i + 1
      exchange A[i] ↔ A[j]
exchange A[i+1] ↔ A[r]
return  i+1
```

L7.7

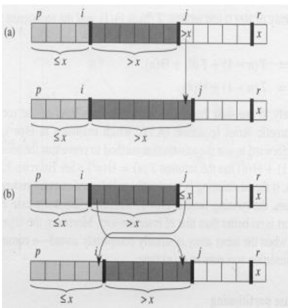
The operation of *Partition* on a sample array




```
x = A[r]
i = p – 1
for j = p to r -1
  if A[j] ≤ x
     i = i + 1
     exchange A[i] ↔ A[j]
exchange A[i+1] ↔ A[r]
return  i+1
```

L7.8

Two cases for one iteration of procedure *Partition*



```
x = A[r]
i = p – 1
for j = p to r -1
 if A[j] ≤ x
    i = i + 1
    exchange A[i] ↔ A[j]
exchange A[i+1] ↔ A[r]
return  i+1
```

Complexity:
Partition on A[p…r] is Θ(n)
where n = r –p +1

Why???
(Exercise 7.1-3)

Exercise 7.1-3

- Give a brief argument that the running time of PARTITION on a subarray of size *n* is Θ(*n*).

Answer:

The for loop makes exactly r – p iterations, each of which takes at most constant time. The part outside the for loop takes at most constant time. Since r – p is the size of the subarray, PARTITION takes at most time proportional to the size of the subarray it is called on.

Another partitioning example

p *r*
initially: *2 5 8 3 9 4 1 7 10* ***6***
i j

note: *pivot (x) = 6*

next iteration: *2 5 8 3 9 4 1 7 10* ***6***
i j

next iteration: *2 5 8 3 9 4 1 7 10* ***6***
i j

next iteration: *2 5 8 3 9 4 1 7 10* ***6***
i j

next iteration: *2 5 3 8 9 4 1 7 10* ***6***
i j

```
x = A[r]
i = p – 1
for j = p to r -1
 if A[j] ≤ x
   i = i + 1
   exchange A[i] ↔ A[j]
exchange A[i+1] ↔ A[r]
return  i+1
```


Another example (Continued)

next iteration: 2 5 3 8 9 4 1 7 10 **6**
i j

next iteration: 2 5 3 8 9 4 1 7 10 **6**
i j

next iteration: 2 5 3 4 9 8 1 7 10 **6**
i j

next iteration: 2 5 3 4 1 8 9 7 10 **6**
i j

next iteration: 2 5 3 4 1 8 9 7 10 **6**
i j

next iteration: 2 5 3 4 1 8 9 7 10 **6**
i j

after final swap: 2 5 3 4 1 **6** 9 7 10 8
i j

```
x = A[r]
i = p – 1
for j = p to r -1
 if A[j] ≤ x
   i = i + 1
   exchange A[i] ↔ A[j]
exchange A[i+1] ↔ A[r]
return  i+1
```