Chapter 7 Quicksort

Quicksort is a divide and conquer sorting algorithm. An array A[p..r] is first partitioned into two non-empty sub-arrays A[p..q-1] and A[q+1..r] such that every key in A[p..q-1] is less than or equal A[q] and every key in A[q+1.. r] is greater than or equal to A[q]. Then the two sub-arrays are sorted by recursive calls to Quicksort.

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)
```

QUICKSORT(A, 1, A.length) will sort the entire array, A.

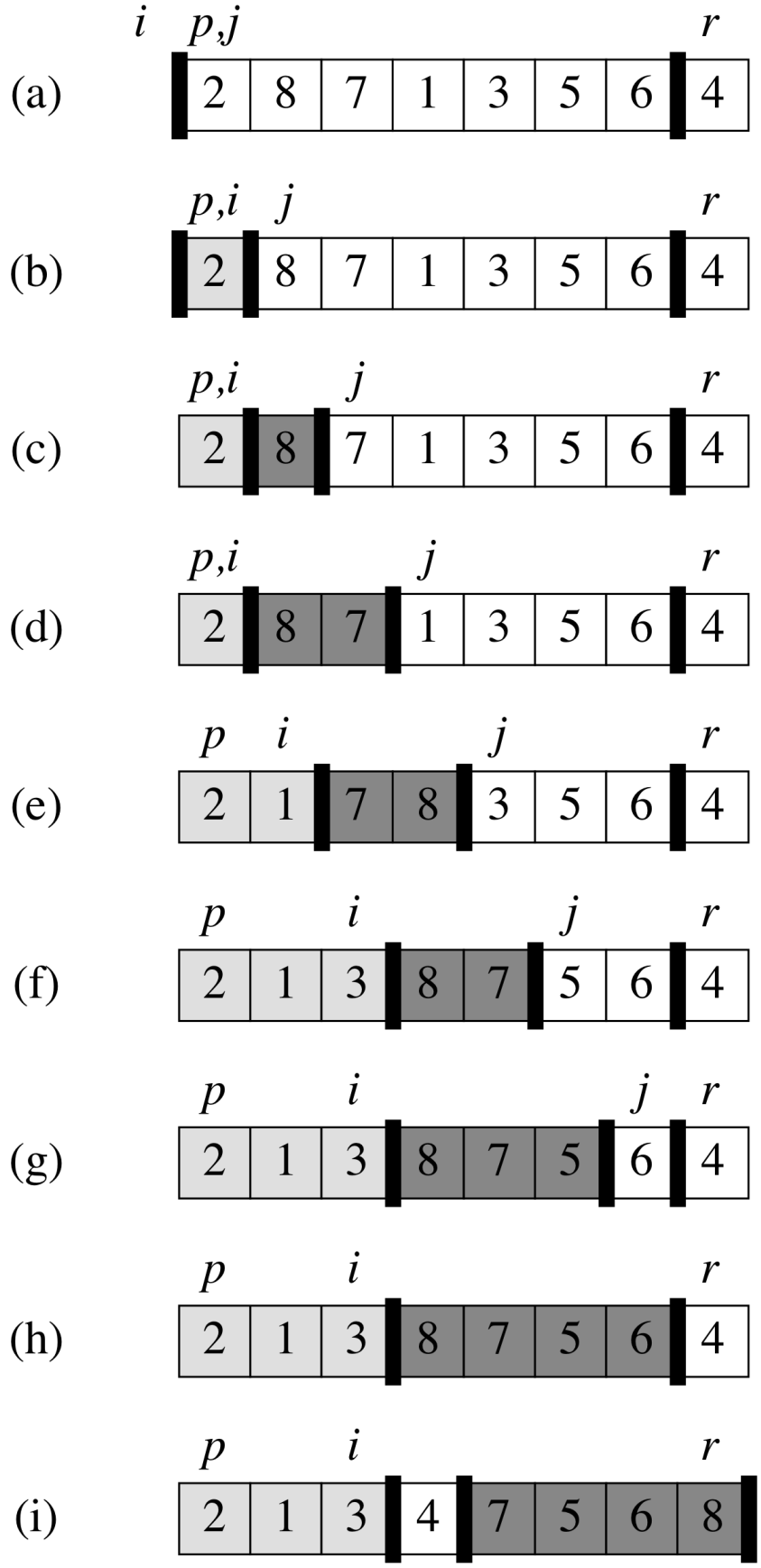
The initial call is QUICKSORT(A, 1, n)

Partitioning

Partition subarray A[p.. r] by the following procedure

PARTITION(A, p, r)

```
1   x = A[r]
2   i = p -1
3   for j = p to r -1
4         if A[j] ≤ x
5            i = i +1
6            exchange A[i] with A[j]
7   exchange A[i+1] with A[r]
8   return i + 1
```


(a)
i p,j r
2 8 7 1 3 5 6 4
(b)
p,i j r
2 8 7 1 3 5 6 4
(c)
p,i j r
2 8 7 1 3 5 6 4
(d)
p,i j r
2 8 7 1 3 5 6 4
(e)
p i j r
2 1 7 8 3 5 6 4
(f)
p i j r
2 1 3 8 7 5 6 4
(g)
p i j r
2 1 3 8 7 5 6 4
(h)
p i r
2 1 3 8 7 5 6 4
(i)
p i r
2 1 3 4 7 5 6 8

PARTITION always selects the last element A[r] in the subarray A[p.. 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.. i] are less than or equal to the pivot.
2. All entries in A[i+1.. j-1] are greater than the pivot.
3. A[r] = pivot.

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

The running time of PARTITION on the array A[p..r] is $\Theta(n)$ where n = the number of keys in the array. Other Partition procedures described in the literature also run in $\Theta(n)$ time.

The running time of QUICKSORT depends on how well balanced the partitioning is performed. A very good partition splits an array up into two equal-sized pieces. A bad partition splits an array up into two pieces of very different sizes. The worst partition puts only one key in one piece and all the other keys in the other piece.

If the subarrays are balanced, then QUICKSORT can run as fast as MERGESORT.

If the subarrays are unbalanced, then QUICKSORT can run as slow as INSERTION sort.

Worst-Case:

- Occurs when the subarray are completely unbalanced.
- Have 0 elements in one subarray and n-1 elements in the other subarray.
- Get the recurrence
  $T(n) = T(n-1) + T(0) + \Theta(n)$
  $= T(n-1) + \Theta(n)$

= Θ(n^2)

- 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 subarray are completely balanced every time
- Each subarray has ≤ n/2 elements.
- Get the recurrence
  T(n) = 2T(n/2) + Θ(n)
  = Θ(n lg n)

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
  T(n) ≤ T(9n/10) + T(n/10) + Θ(n)
  = O(n lg n).
- Intuition: look at the recursion tree
- It's like the one for T(n) = T(n/3) + T(2n/3) + O(n)
- 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 Θ(lg n).

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 say **random sampling**, or picking one 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)
1   i = RANDOM(p, r)
2   exchange A[r] with A[i]
3   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.

```
RANDOMIZED-QUICKSORT (A, p, r)
1    if p < r
2         q = RANDOMIZED-PARTITION (A, p, r)
3        RANDOMIZED-QUICKSORT (A, p, q-1)
4        RANDOMIZED-QUICKSORT (A, q+1, r)
```

Randomization of QUICKSORT stops any specific type of array from causing worst-case behavior. For example, an already-sorted array causes worst-case behavior in non-randomized quicksort, but not in Randomized-Quicksort.

Worst-case analysis

The worst-case running time of QUICKSORT and RANDOMIZED-QUICKSORT is the same.

We will prove that a worst-case split at every level produces a worst-case running time of $O(n^2)$.

- Recurrence for the worst-case running time of QUICKSORT: $T(n) = \max_{0 \le q \le n-1} (T(q) + T(n-q-1)) + \Theta(n)$
- Because Partition produces two subproblems, totaling size n-1, q ranges from 0 to n-1.
- Guess: $T(n) \le cn^2$, for some c.

- Substituting our guess into the above recurrence:
  $T(n) \le \max_{0 \le q \le n-1} (cq^2 + c(n-q-1)^2) + \Theta(n)$

  $= c \cdot \max_{0 \le q \le n-1} (q^2 + (n-q-1)^2) + \Theta(n)$
- The maximum value of $(q^2 + (n - q - 1)^2)$ occurs when q is either 0 or n -1. This means that

$\max_{0 \le q \le n-1} (q^2 + (n-q-1)^2) \le (n-1)^2$

$= n^2 - 2n + 1.$

- Therefore,

  $T(n) \le cn^2 - c(2n - 1) + \Theta(n)$

  $\le cn^2$ if $c(2n - 1) \ge \Theta(n)$

- Pick c so that c(2n -1) dominates Θ(n).
- Therefore, the worst-case running time of QUICKSORT is $O(n^2)$.
- Can also show that the recurrence's solution is $\Omega(n^2)$. Thus the worst-case running time is $\Theta(n^2)$

Average-Case analysis

- The dominant cost of the algorithm is partitioning
- Partition removes the pivot element from future consideration each time.
- Thus, Partition is called at most n times.
- The amount of work that each call to Partition does is a constant plus the number of comparisons that are performed in its **for** loop.
- Let X = the total number of comparisons performed in all calls to Partition.
- Therefore, the total work done over the entire execution is O(n + X)
- The expected running time of QUICKSORT, using Randomized-Partition is O(n lg n).