Chap7. Quick Sort

- What is quick sort?
- How does it work?
- Performance of quick sort
- Several strategies to pick the pivot
- Randomized quick sort

Quick Sort

- by C.A.R. Hoare(British computer scientist at age 26)
- Divide-and-conquer algorithm.
- Sorts in place (like insertion sort, but not like merge sort).
- Very practical (with code tuning).
- Worst-case : $\mathbf{\Theta(\boldsymbol{n}^2)}$.
- Average-case : $\mathbf{\Theta(\boldsymbol{n} \log \boldsymbol{n})}$.
- Empirical and analytical studies show that quicksort can be *e xpected* to be twice as fast as its competitors(merge sort).

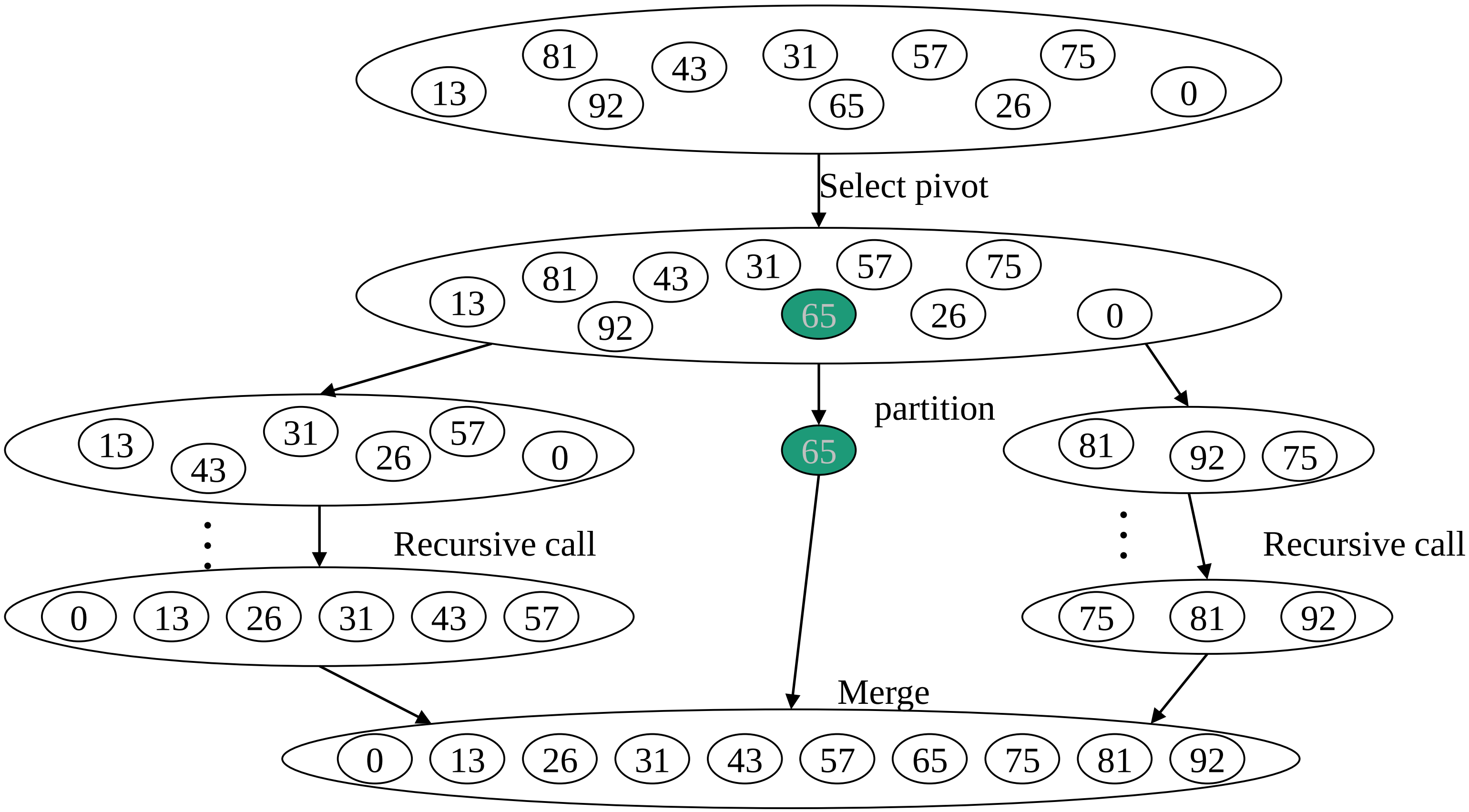
Quick Sort

- Divide-and-conquer approach to sorting
- Like Merge Sort, except
  - Don't divide the array in half
  - Partition the array based elements being less than or greater than some element of the array (*the pivot*)
  - divide phase does all the work; merge phase is trivial.

Quick Sort

- Quicksort is another divide-and-conquer algorithm.
- Basically, what we do is divide the array into two subarrays, so that all the values on the left are smaller than the values on the right.
- We repeat this process until our subarrays have only 1 element in them.
- When we return from the series of recursive calls, our array is sorted.

QuickSort Example



Quicksort

- **Divide:** Partition A[p..r] into two subarrays A[p..q-1] and A[q+1 .. r] such each element of A[p..q-1] ≤ A[q] and A[q] ≤ each element of A[q+1..r]. Compute the index q as part of this partitioning procedure.
- **Conquer:** Sort the two subarrays by recursive calls to quicksort.
- **Combine:** Since the subarrays are sorted in place, no work is needed to combine them: A[p..r] is now sorted.

The Quicksort Algorithm

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

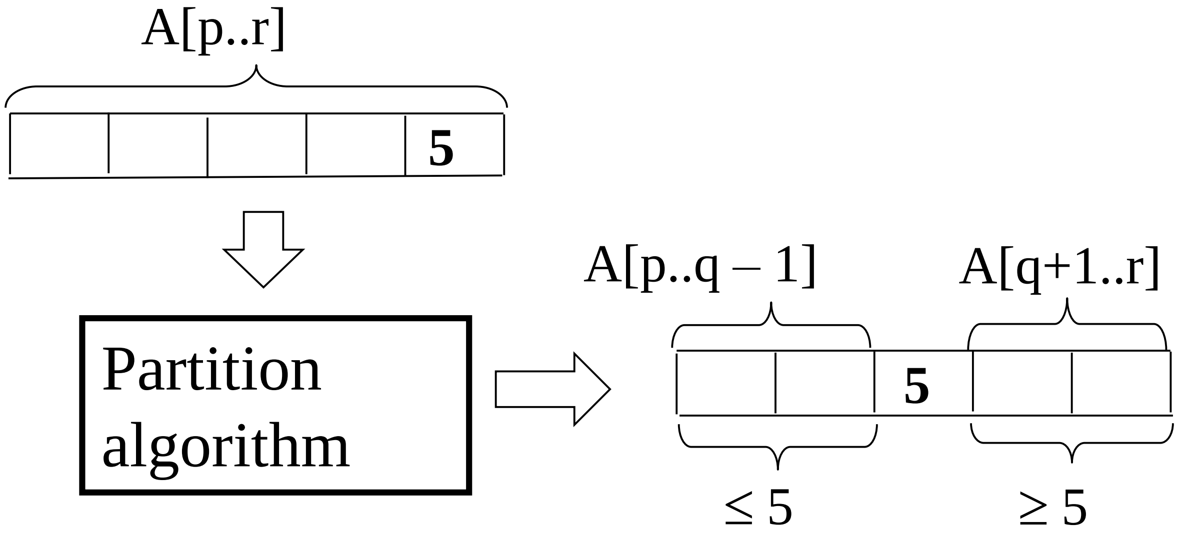
Initial call:

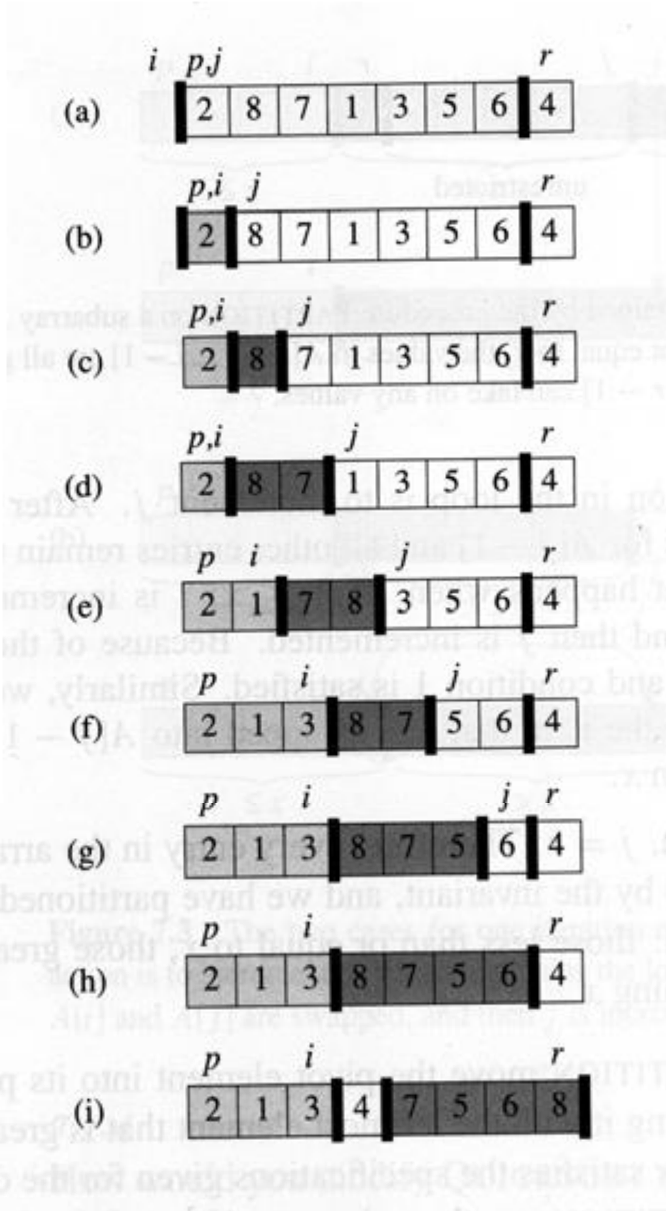
QUICKSORT(A,1, length[A])

Partition Algorithm

```
PARTITION(A,p,r)
x ← A[r]
i ← p - 1
for j ← p to r-1
        do if A[j] ≤ x
                then i ← i + 1
                        exchange A[i] ↔ A[j]
exchange A[i+1] ↔ A[r]
return i+1
```

Simple example






```
PARTITION(A,p,r)
    x ← A[r]
    i ← p - 1
    for j ← p to r-1
        do if A[j] ≤ x
            then i ← i + 1
                exchange A[i] ↔ A[j]
    exchange A[i+1] ↔ A[r]
    return i+1
```

Example

```
                 p                               r
initially:       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  8  3  9  4  1  7  10  6
                    i     j

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

note: pivot (x) = 6

PARTITION(A,p,r)

```
x ← A[r]
i ← p - 1
for j ← p to r-1
    do if A[j] ≤ x
        then i ← i + 1
            exchange A[i] ↔ A[j]
exchange A[i+1] ↔ A[r]
return i+1
```

Example (Continued)

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

i at 3, j at 9

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

i at 3, j at 4

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

i at 4, j at 1

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

i at 1, j at 7

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

i at 1, j at 10

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

i at 1, j at 6

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

i at 1, j at 8

```
PARTITION(A,p,r)
x ← A[r]
i ← p - 1
for j ← p to r-1
      do if A[j] ≤ x
            then i ← i + 1
                  exchange A[i] ↔ A[j]
exchange A[i+1] ↔ A[r]
return i+1
```

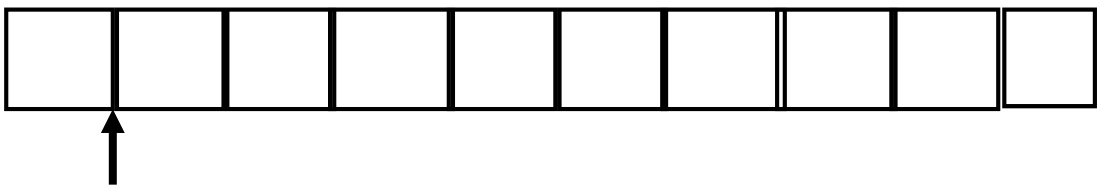
$\Theta(n)$

Performance of Quicksort

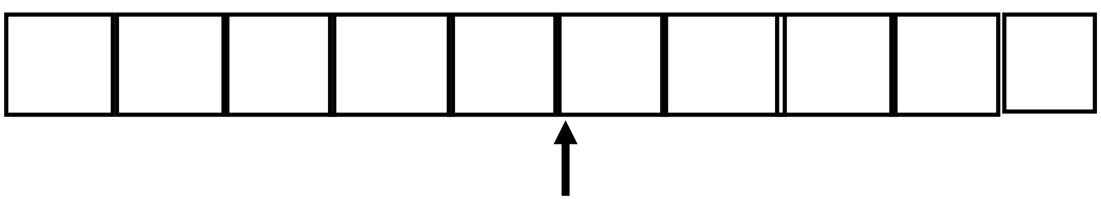
- Depends on whether the partition is balanced or unbalanced:
  - Balance of partition depends on selection of pivot
  - If balanced, runs as fast as Merge sort.
  - If unbalanced, runs as slowly as Insertion sort

Worst/Best case partitioning

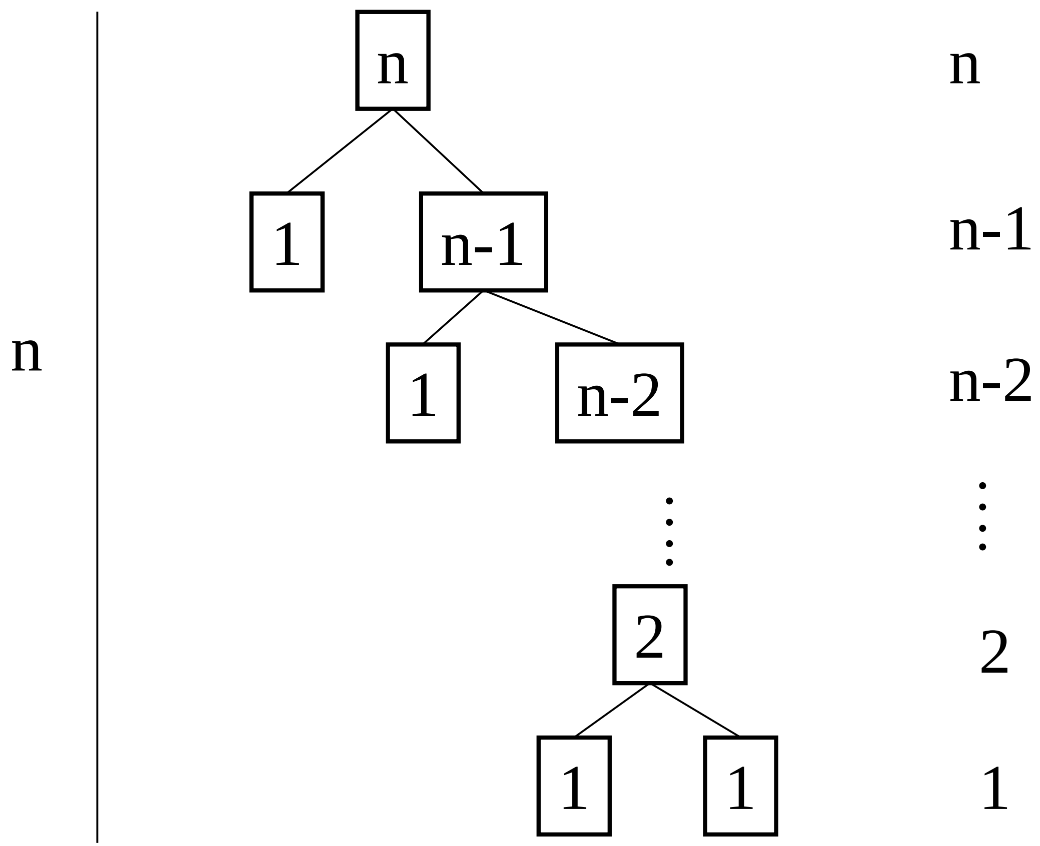
- **Worst case:**
  - One partition contains n – 1 elements
  - The other partition contains 1 element



- **Best case:**
  - Both partitions are of equal size



Worst case partitioning



Worst case performance

- Assume we have a maximally unbalanced partition at each step, splitting off just 1 element from the rest each time. This means we will have to call Partition n-1 times.
- The cost of Partition is: $\Theta(n)$
- So the recurrence for Quicksort is:

$$T(n) = T(n-1) + T(0) + \Theta(n)$$
$$= T(n-1) + \Theta(n)$$
$$= \Theta(n^2) \quad \text{by iterative substitution}$$

Best case performance

- Size of each subproblem $\leq n/2$.
  - One of the subproblems is of size $\lfloor n/2 \rfloor$
  - The other is of size $\lceil n/2 \rceil - 1$.
- Recurrence for running time
  - $T(n) \leq 2T(n/2)$ + PartitionTime(n)
    $= 2T(n/2) + \Theta(n)$
- $\boldsymbol{T(n) = \Theta(n \log n)}$ by master theorem case2

Average Case

- Average case analysis is complex and difficult.
- However, we can observe that average-case performance is much closer to best-case than worst case.
- Suppose split is always 9-to-1
- Recurrence:

  $T(n) \leq T(9n/10) + T(n/10) + \Theta(n)$

  $= T(9n/10) + T(n/10) + cn$

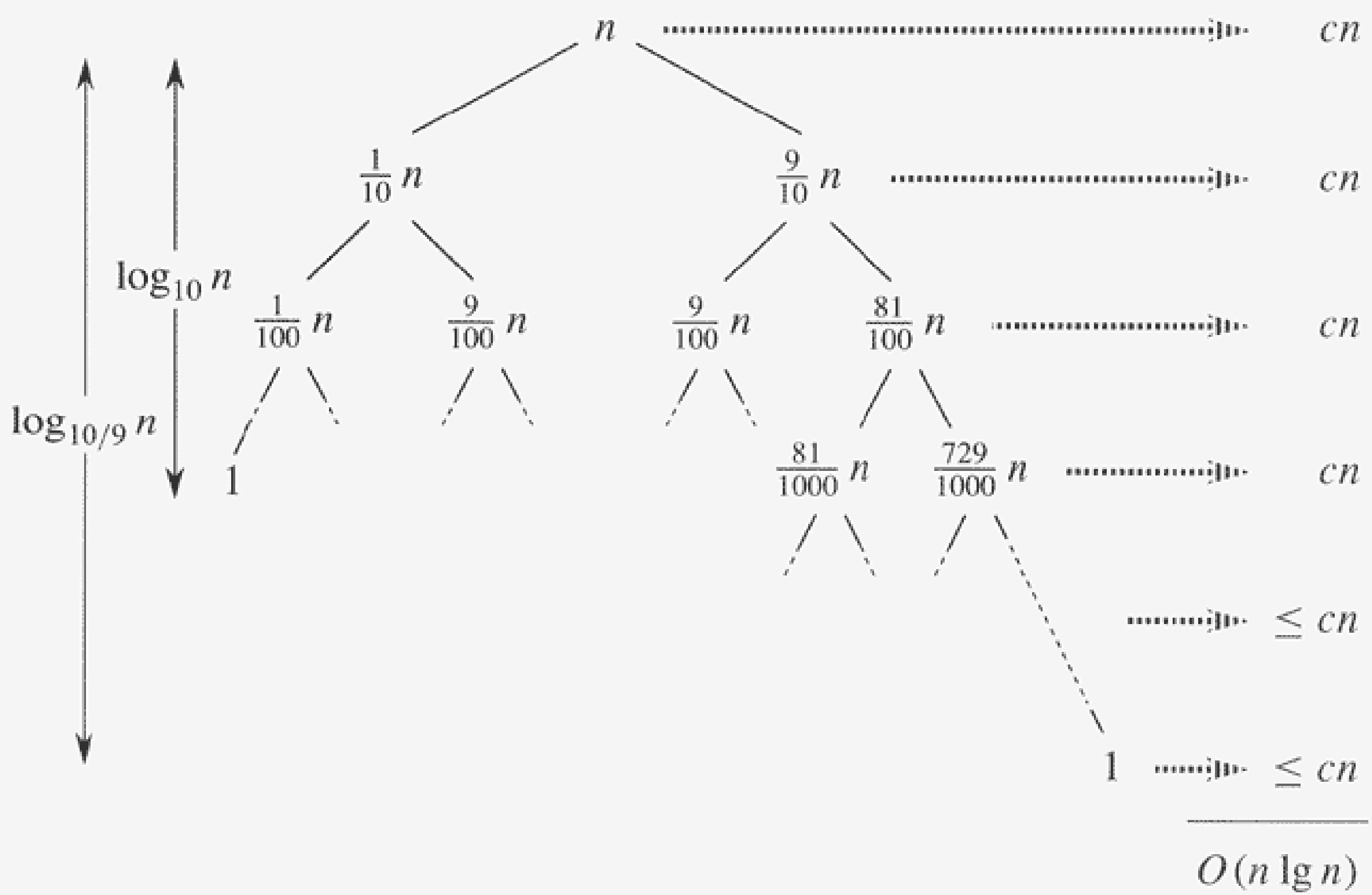


Figure 7.4 A recursion tree for QUICKSORT in which PARTITION always produces a 9-to-1 split, yielding a running time of $O(n \lg n)$. Nodes show subproblem sizes, with per-level costs on the right. The per-level costs include the constant c implicit in the $\Theta(n)$ term.

Average case

- What if we have a 99-1 split?
  $T(n) \leq T(99n/100) + T(n/100) + \Theta(n)$
- We still have a running time of $O(n \log n)$
- 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 $\Theta(\log n)$.
- So whenever the split of constant proportionality, Quicksort performs on the order of $O(n \log n)$.