Sorting Lower Bounds: Decision trees & QuickSort

Adapted from lectures of
Prof. Charles Leiserson , MIT &
CLRC textbook 2nd ed. Chapters 7,8

Lecture Outline

- Quick sort algorithm: divide and conquer
- Analysis of quick sort
- Decision tree: comparison-based sort
- Optimal runtime for comparison sort algorithms

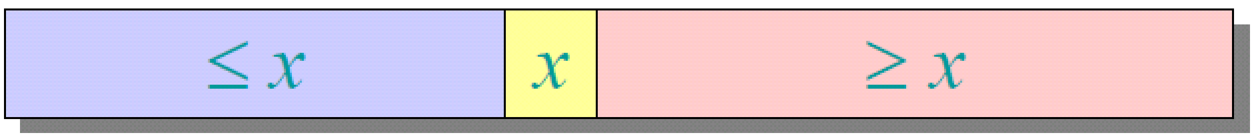
QuickSort

- Proposed by C.A.R. Hoare in 1962.
- Divide-and-conquer algorithm.
- Sorts “in place”(like insertion sort, but not like merge sort).
- Very practical (with tuning).

Divide and conquer

- Quicksort an n-element array:

1. Divide: Partition the array into two subarrays around a pivot x such that elements in lower subarray ≤x≤elements in upper subarray.



2. Conquer: Recursively sort the two subarrays.
3. Combine: Trivial.

- **Key**: Linear-time partitioning subroutine.

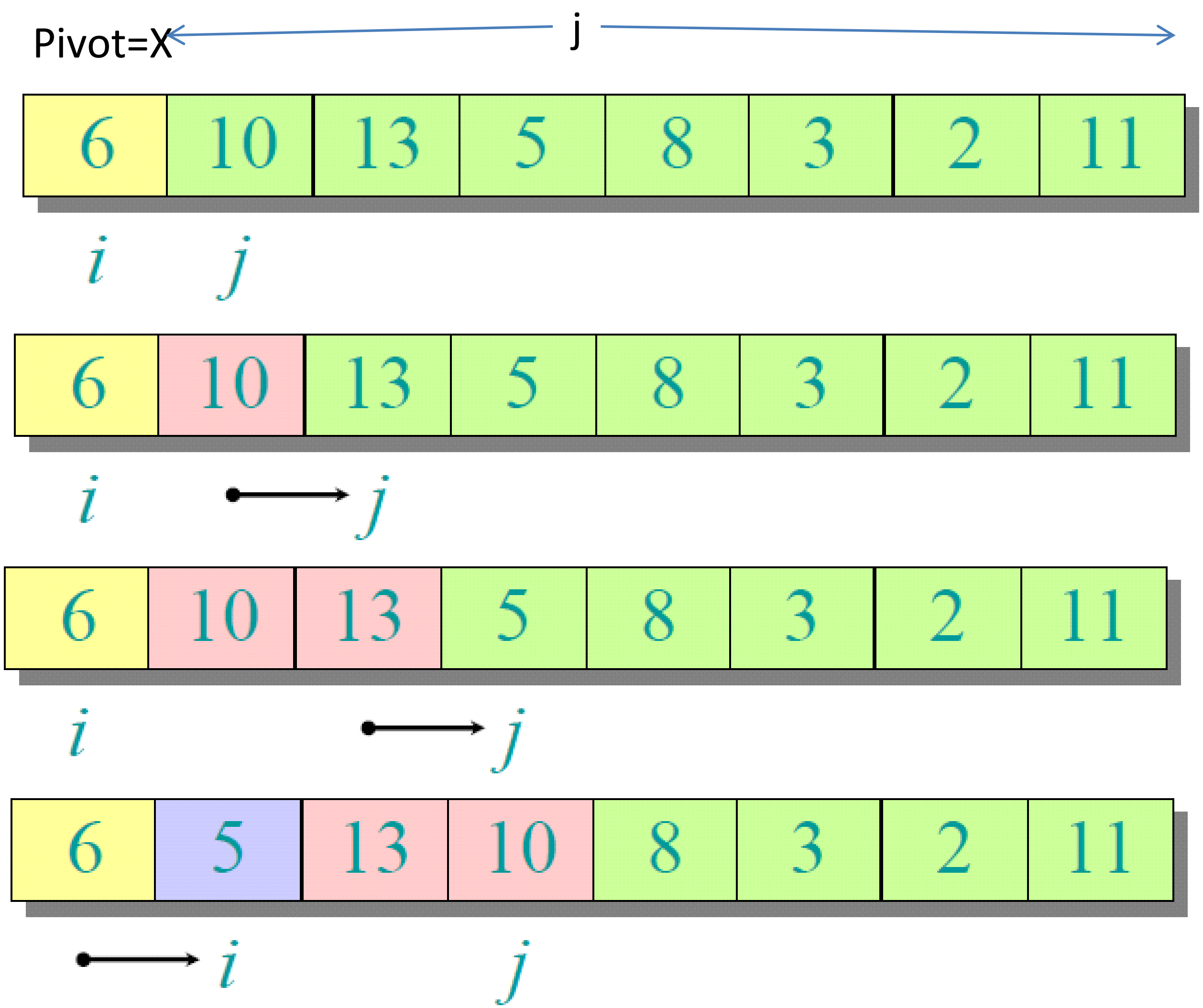
Partitioning subroutine

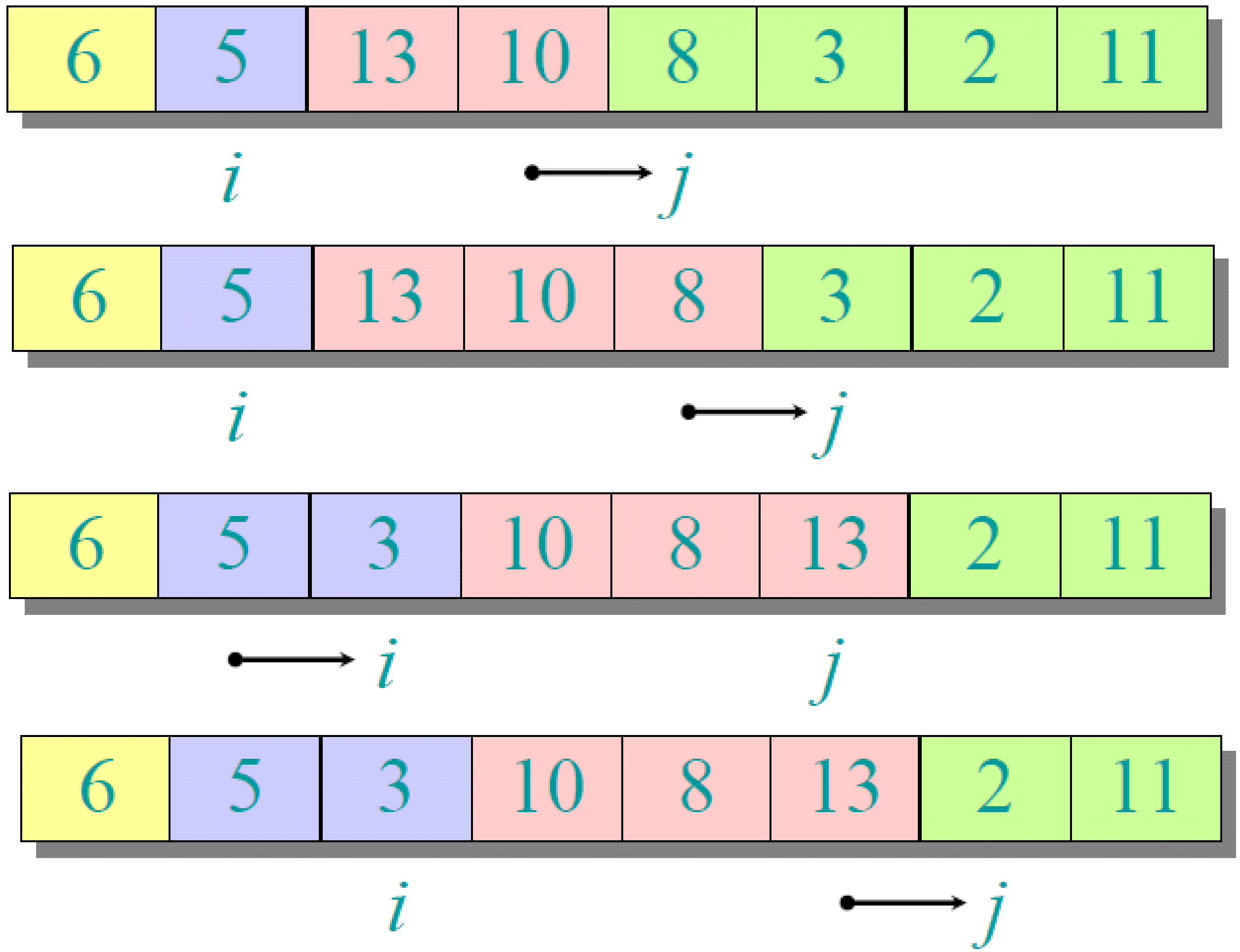
```
PARTITION(A, p, q)            ⊳A[p. . q]
x←A[p]                        ⊳pivot= A[p]
i←p
for j←p+ 1 to q
    do if A[j] ≤x
        then  i←i+ 1
              exchange A[i] ↔A[j]
exchange A[p] ↔A[i]
return i
```

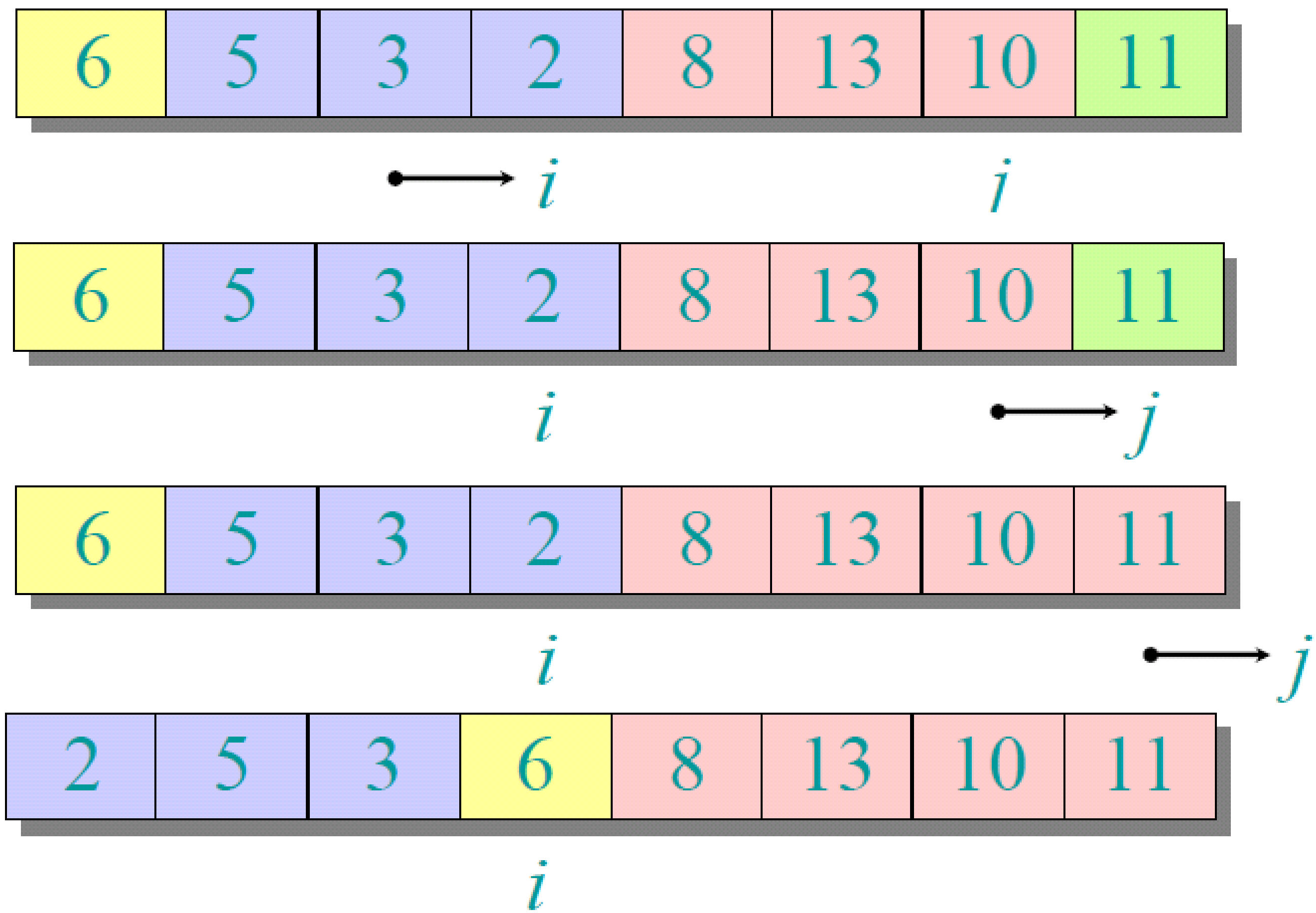
Partitioning subroutine

- Runtime for the partitioning algorithm is:
- $O(n)$ for n elements

Ex: quicksort <6,10,13,5,8,3,2,11>




6 5 13 10 8 3 2 11
i j
6 5 13 10 8 3 2 11
i j
6 5 3 10 8 13 2 11
i j
6 5 3 10 8 13 2 11
i j


6 5 3 2 8 13 10 11
i
j
6 5 3 2 8 13 10 11
i
j
6 5 3 2 8 13 10 11
i
j
2 5 3 6 8 13 10 11
i

Pseudocode for quicksort

```
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, n)

Analysis of quicksort

- Assume all input elements are distinct.
- In practice, there are better partitioning algorithms for when duplicate input elements may exist.
- Let *$T(n)$=worst-case running time on an array of n elements.*

Worst-case of quicksort

- Input sorted or reverse sorted.
- Partition around min or max element.
- One side of partition always has no elements.

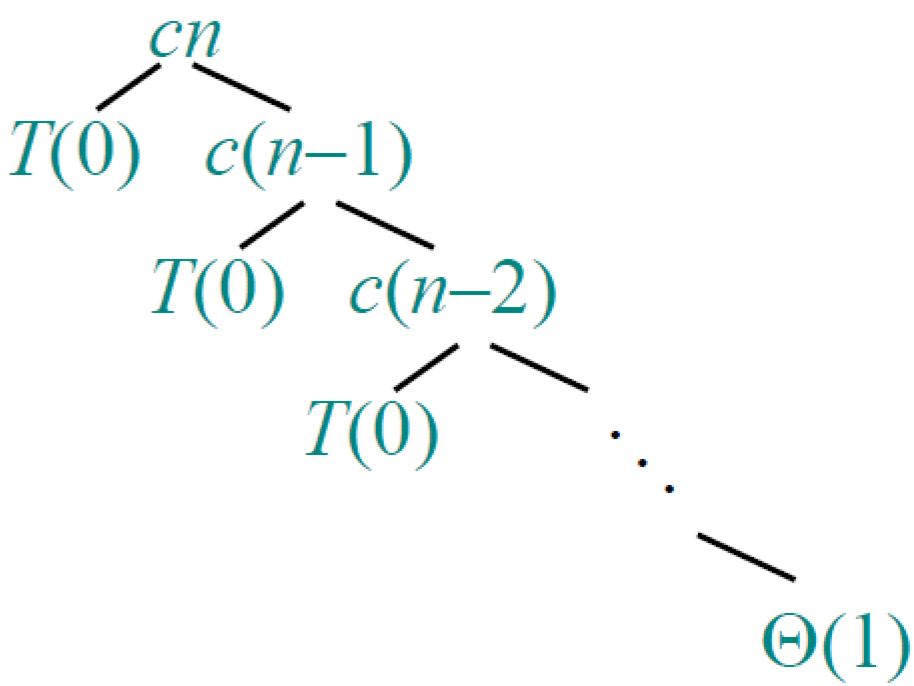
Worst-case of quicksort

- Input sorted or reverse sorted.
- Partition around min or max element.
- One side of partition always has no elements.

$$\begin{aligned} T(n) &= T(0) + T(n-1) + \Theta(n) \\ &= \Theta(1) + T(n-1) + \Theta(n) \\ &= T(n-1) + \Theta(n) \end{aligned}$$

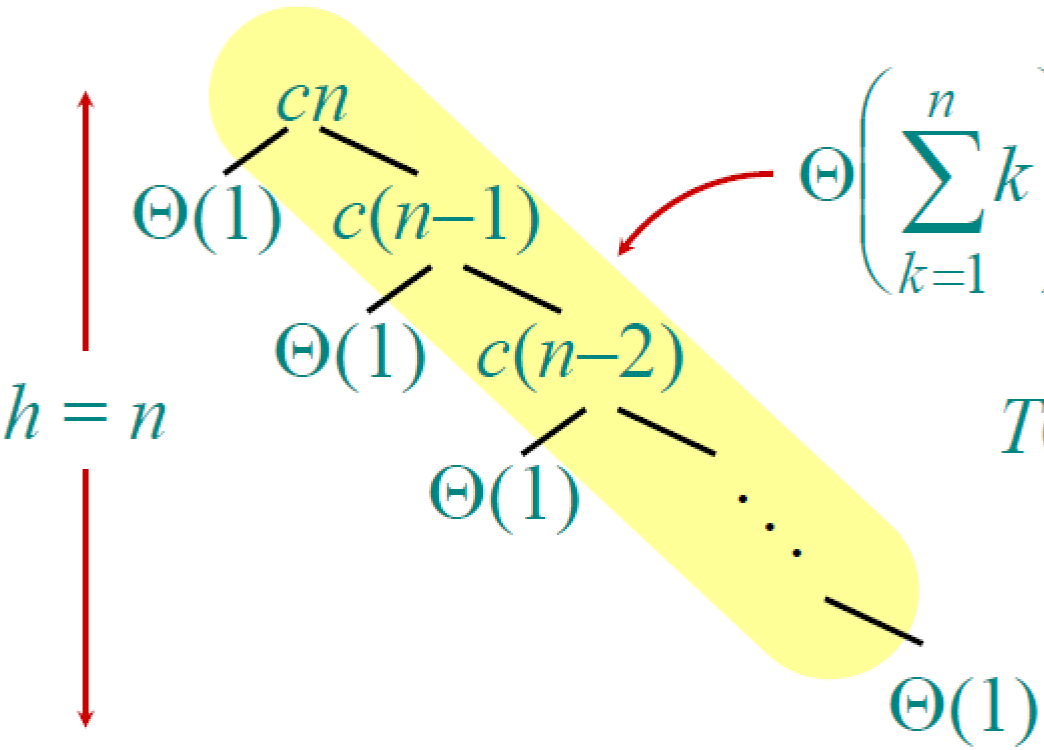
Worst-case recursion tree

$$T(n) = T(0) + T(n-1) + cn$$



$$T(n) = T(0) + T(n-1) + cn$$

Arithmetic series



$$\Theta\left(\sum_{k=1}^{n} k\right) = \Theta\left(n^2\right)$$

$$T(n) = \Theta(n) + \Theta(n^2)$$
$$= \Theta(n^2)$$

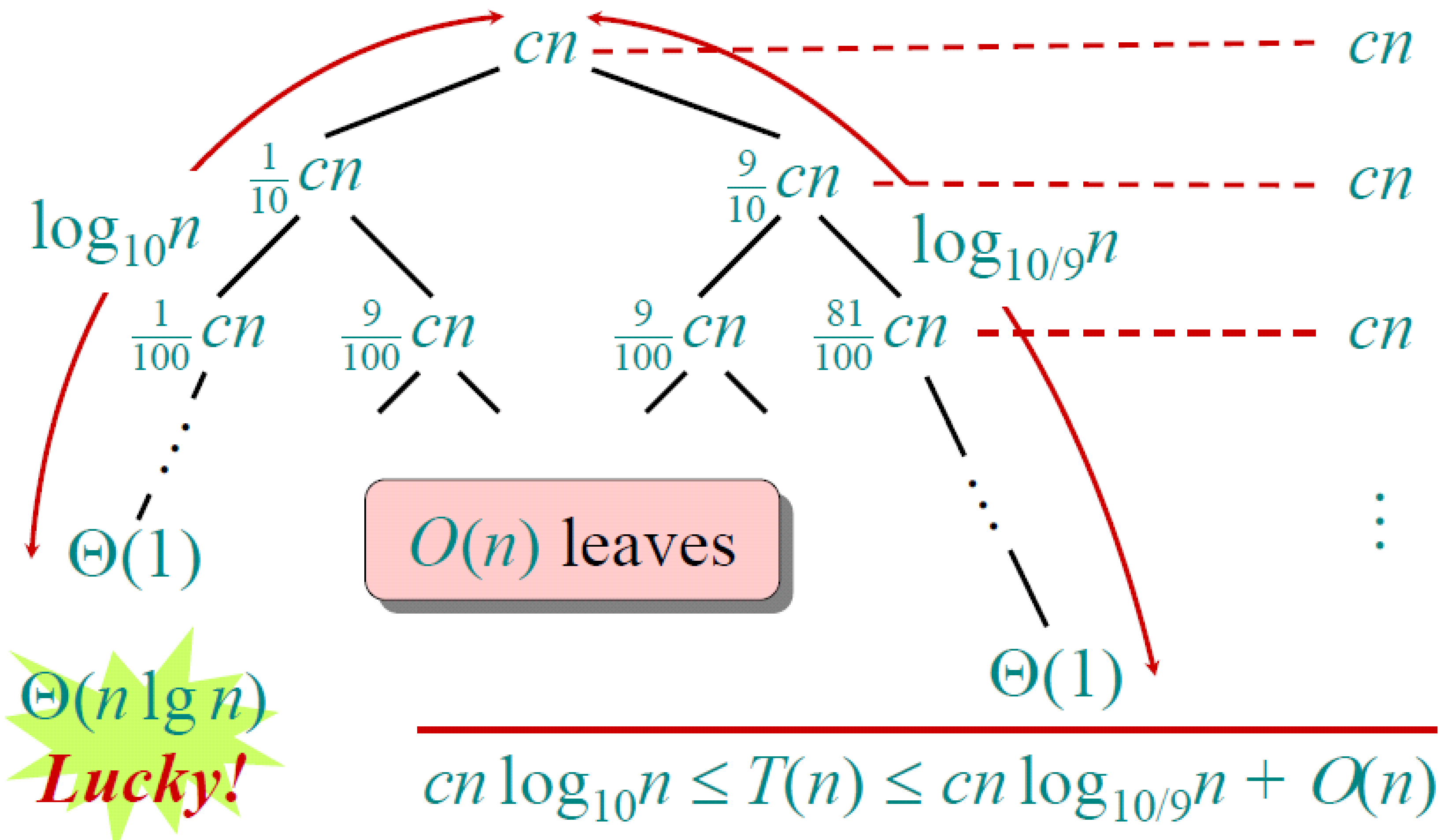
Best-case analysis
(For intuition only!)

- If we're lucky, PARTITION splits the array evenly:
- *T(n)= 2T(n/2) + Θ(n)= Θ(nlg n) (same as merge sort!)*
- What if the split is always 1/10 : 9/10?

$$T(n) = T\left(\tfrac{1}{10}n\right) + T\left(\tfrac{9}{10}n\right) + \Theta(n)$$

- What is the solution to this recurrence?

Recursion tree



$$cn \log_{10} n \le T(n) \le cn \log_{10/9} n + O(n)$$

Randomized quicksort

- Since we don’t know where the split will take place, randomized algorithm analysis is applied.
- **IDEA: Partition around a *random element.***
- Running time is independent of the input order.
- No assumptions need to be made about the input distribution.
- No specific input elicits the worst-case behavior.
- The worst case is determined only by the output of a random-number generator.

How fast can we sort?

- All the sorting algorithms we have seen so far are ***comparison sorts*** ***(only use comparisons to determine the relative order of elements.)***
- ***E.g., insertion sort, merge sort, quicksort, heapsort.***

- The best worst-case running time that we've seen for comparison sorting is *O(nlgn).*

- ***Is O(nlgn) the best we can do?***

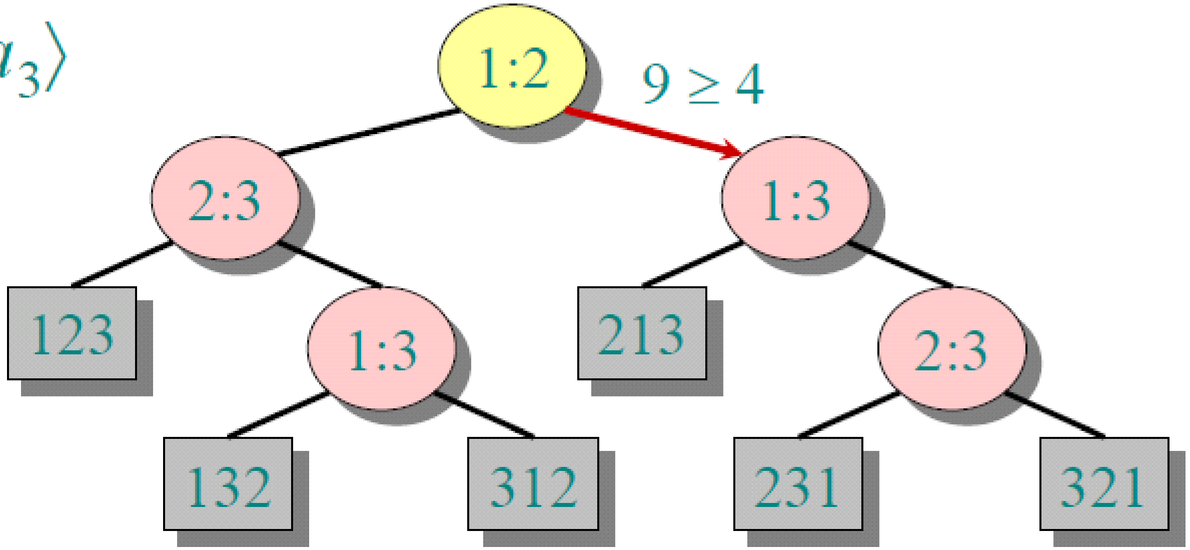
- ***Decision trees can help us answer this question.***

Decision-tree example

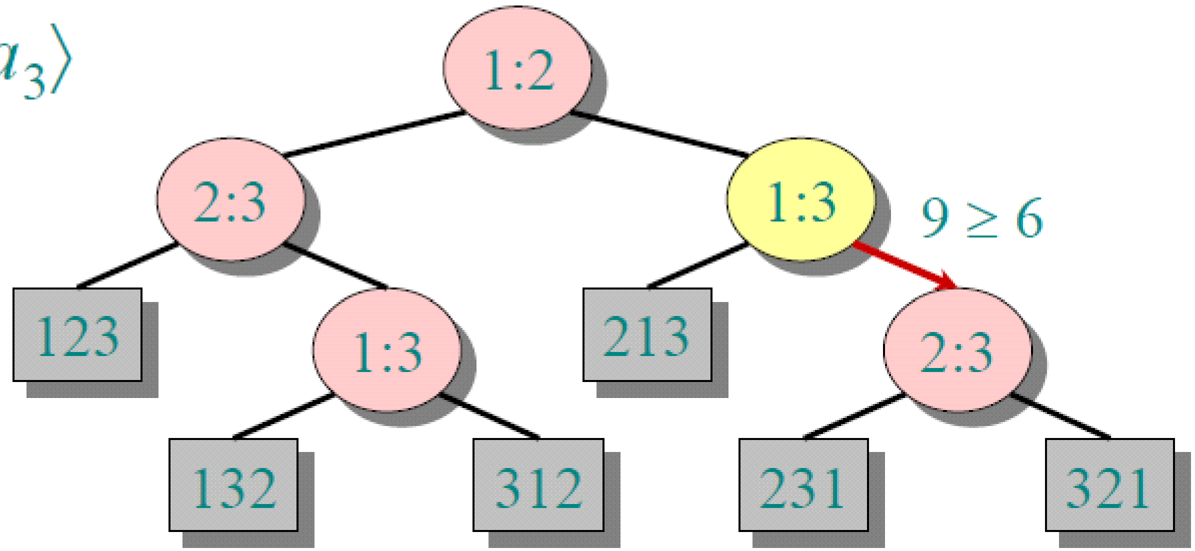
Sort $\langle a_1, a_2, \ldots, a_n \rangle$

- Draw a binary tree
- Each internal node is labeled *i:j for i, j ∈ {1, 2,…, n}.*
- The left subtree shows subsequent comparisons if *ai≤aj.*
- The right subtree shows subsequent comparisons if *ai≥aj.*

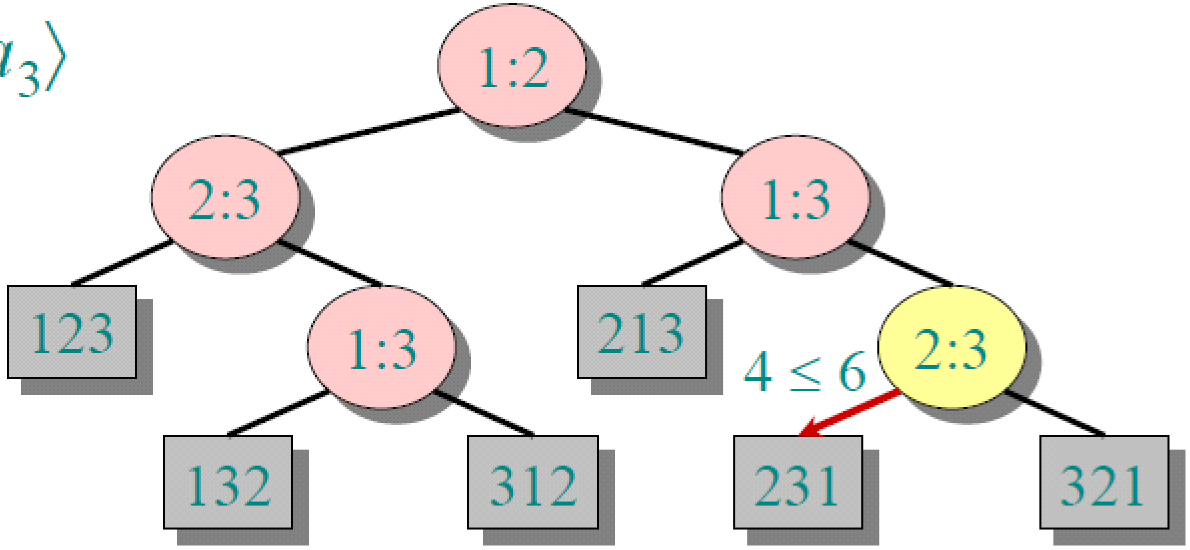
Sort $\langle a_1, a_2, a_3 \rangle$
$= \langle\ 9, 4, 6\ \rangle$:



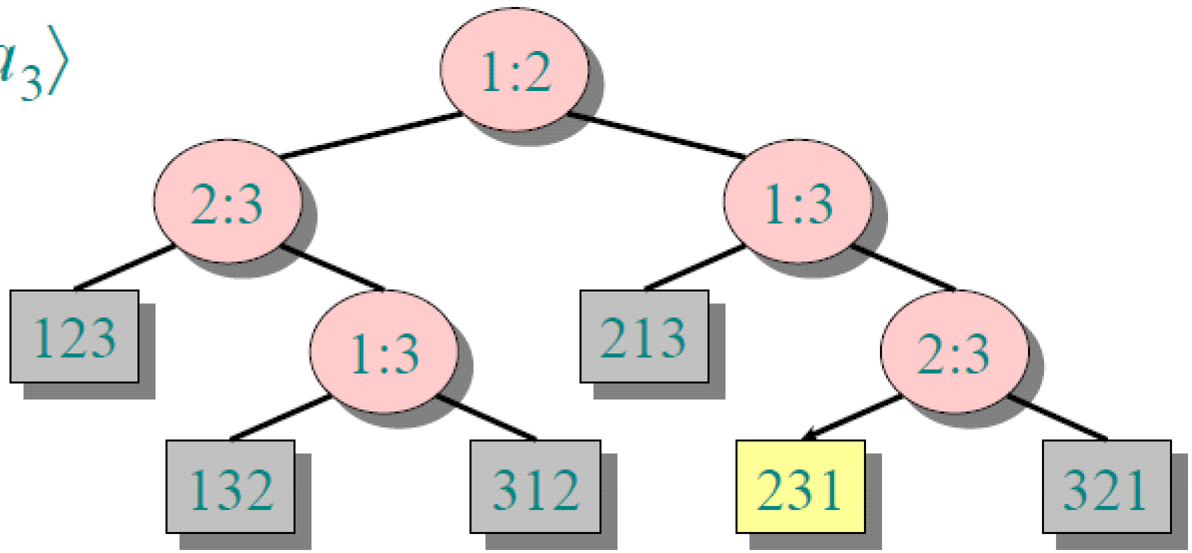
Sort $\langle a_1, a_2, a_3 \rangle = \langle 9, 4, 6 \rangle$:



Sort $\langle a_1, a_2, a_3 \rangle$
$= \langle 9, 4, 6 \rangle$:



Sort $\langle a_1, a_2, a_3 \rangle$
$= \langle 9, 4, 6 \rangle$:



$4 \leq 6 \leq 9$

- Each leaf contains a permutation

  $\langle \pi(1), \pi(2), \ldots, \pi(n) \rangle$ *to indicate that the ordering* $a_{\pi(1)} \leq a_{\pi(2)} \leq \ldots \leq a_{\pi(n)}$ *has been established.*

Decision-tree model

- *A decision tree can model the execution of any comparison-based sort:*
- One tree for each input size *n.*
- View the algorithm as splitting whenever it compares two elements.
- The tree contains the comparisons along all possible execution traces.
- The running time of the algorithm =

  the length of the path taken.
- Worst-case running time = height of tree.

Lower bound for decision-tree sorting

Lemma:

Any binary tree of height h has #leaves $l \leq 2^h$.

Lower bound for decision-tree sorting

Theorem:

Any decision tree that can sort *n elements must have height* $\Omega(n\lg n)$.

Proof:

The tree must contain $\geq n!$ *leaves, since there are* $n!$ *possible permutations.*

A height-h *binary tree has* $l \leq 2^h$ *leaves. (by Lemma)*

Thus, $n! \leq l \leq 2^h$.

$\therefore h \geq \lg(n!)$ *(lg is monotonically increasing)*

$h \geq \lg((n/e)n)$ *(Stirling's formula)*$= n \lg n - n\lg e = \Omega(n \lg n)$.

Lower bound for comparison sorting …

Corollary:

Merge sort (and heapsort) are asymptotically optimal comparison sorting algorithms.

$T(n) = O(n \lg n)$

(conforming with the previous theorem.)