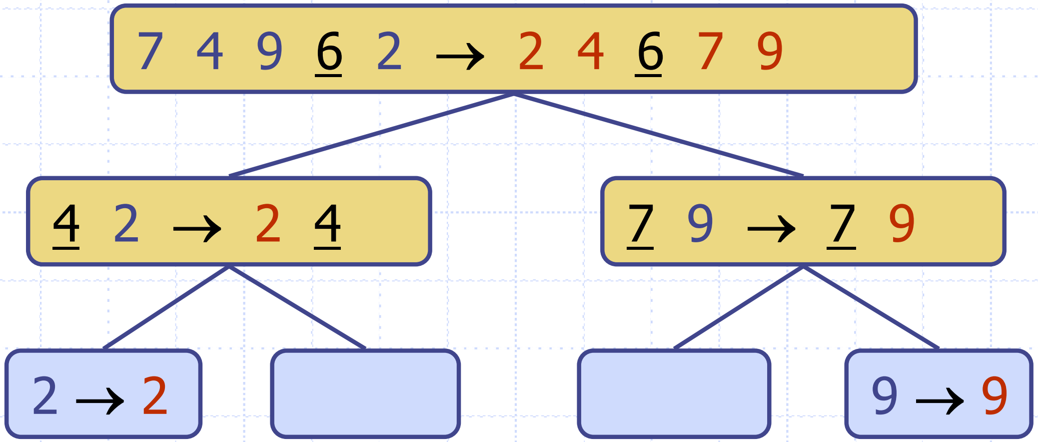
Quick-Sort

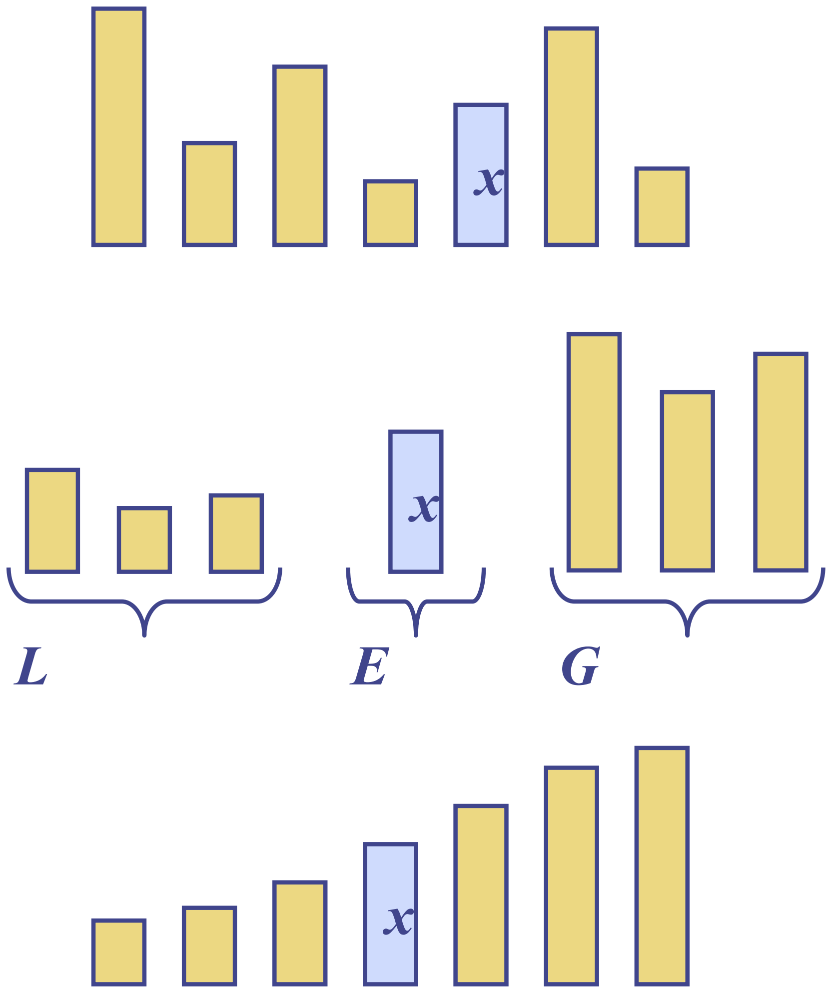


We will look at this later …

Algorithm	Time	Notes
selection-sort	$O(n^2)$	▪ in-place ▪ slow (good for small inputs)
insertion-sort	$O(n^2)$	▪ in-place ▪ slow (good for small inputs)
quick-sort	$O(n \log n)$ expected	▪ in-place, randomized ▪ fastest (good for large inputs)
heap-sort	$O(n \log n)$	▪ in-place ▪ fast (good for large inputs)
merge-sort	$O(n \log n)$	▪ sequential data access ▪ fast (good for huge inputs)

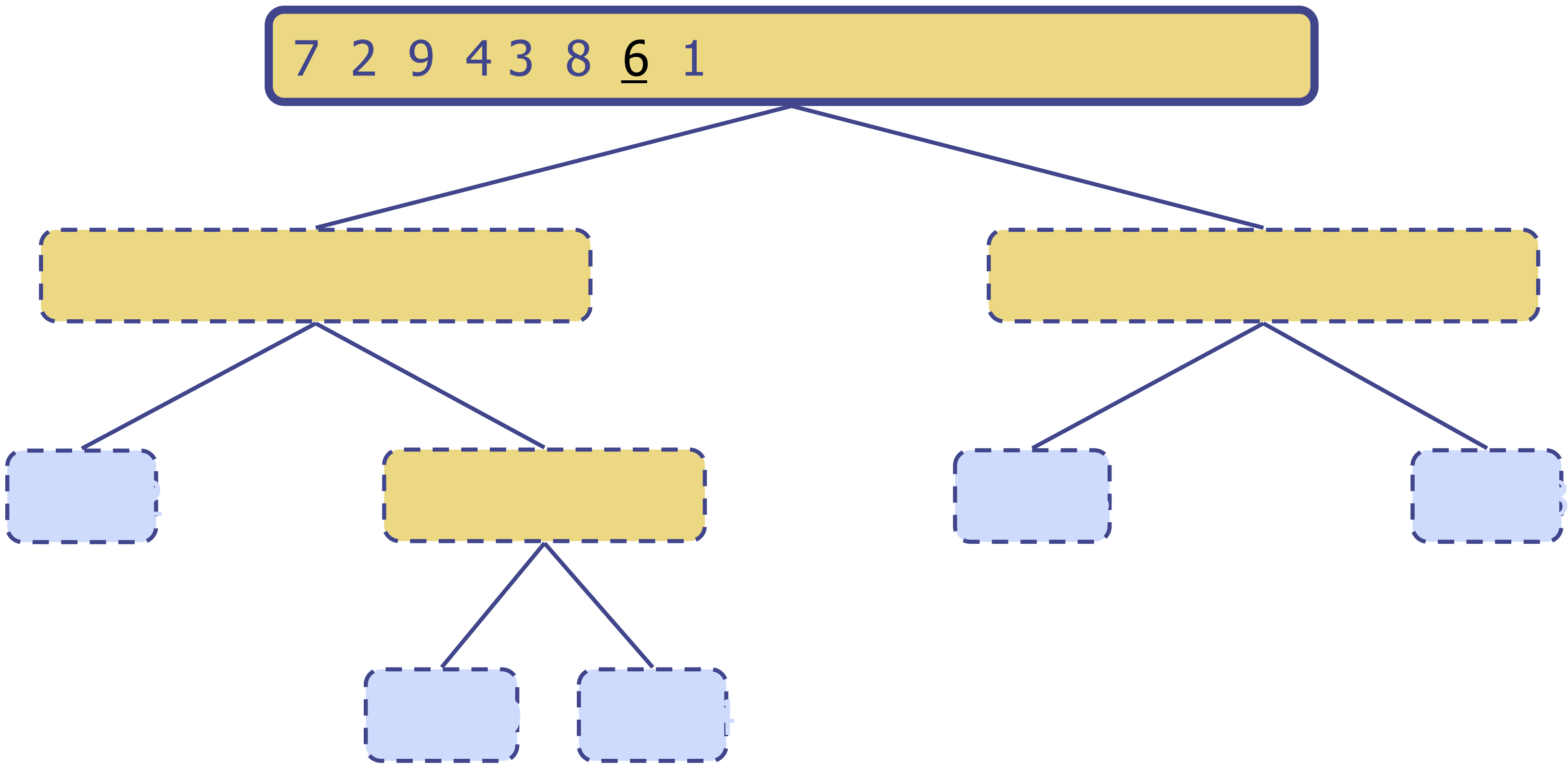
Quick-Sort

- Quick-sort is a randomized sorting algorithm based on the divide-and-conquer paradigm:
  - Divide: pick a **random** element x (called pivot) and partition S into
    - L elements less than x
    - E elements equal x
    - G elements greater than x
  - Recur: sort L and G
  - Conquer: join L, E and G



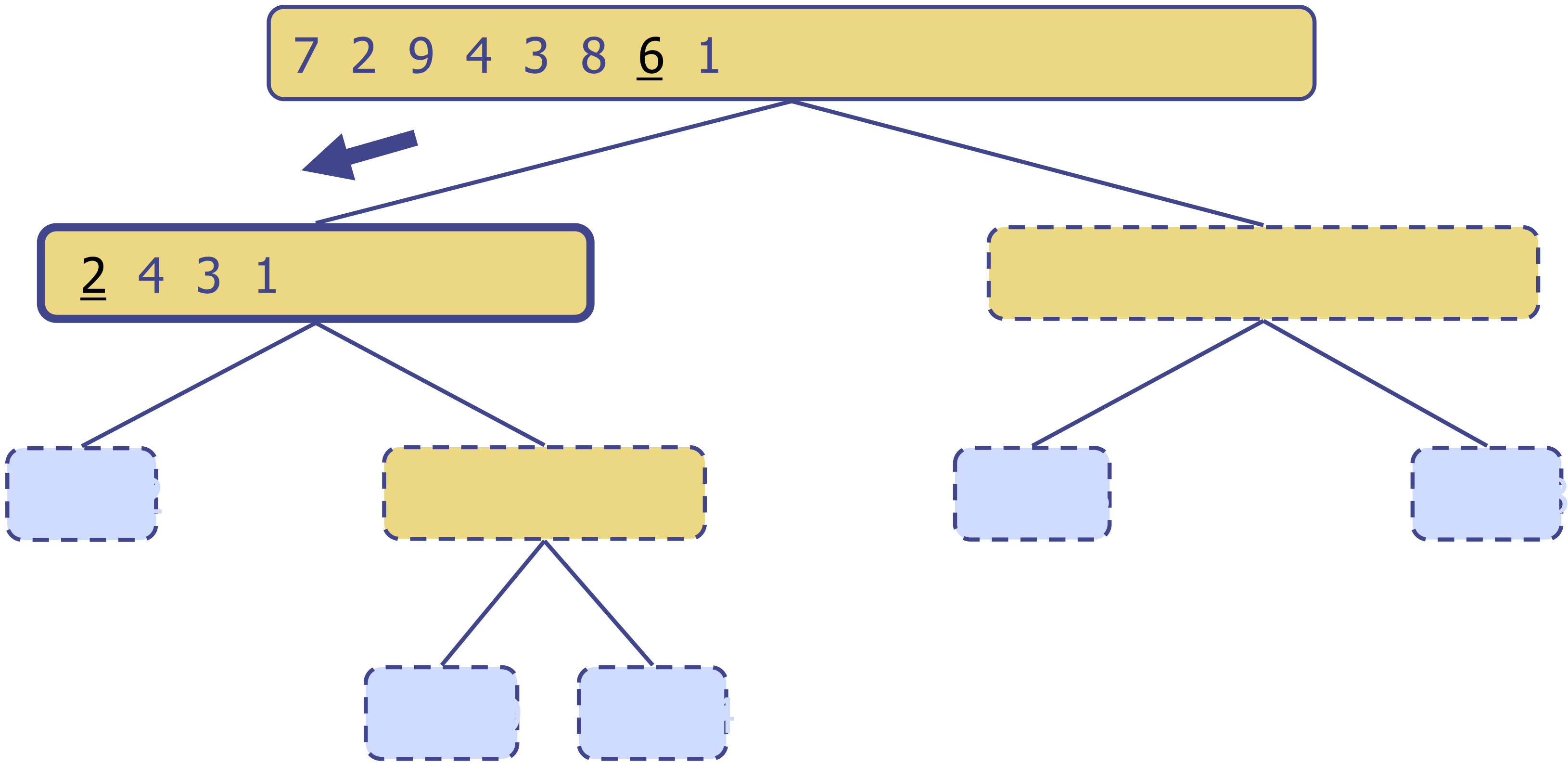
Execution Example

- Pivot selection



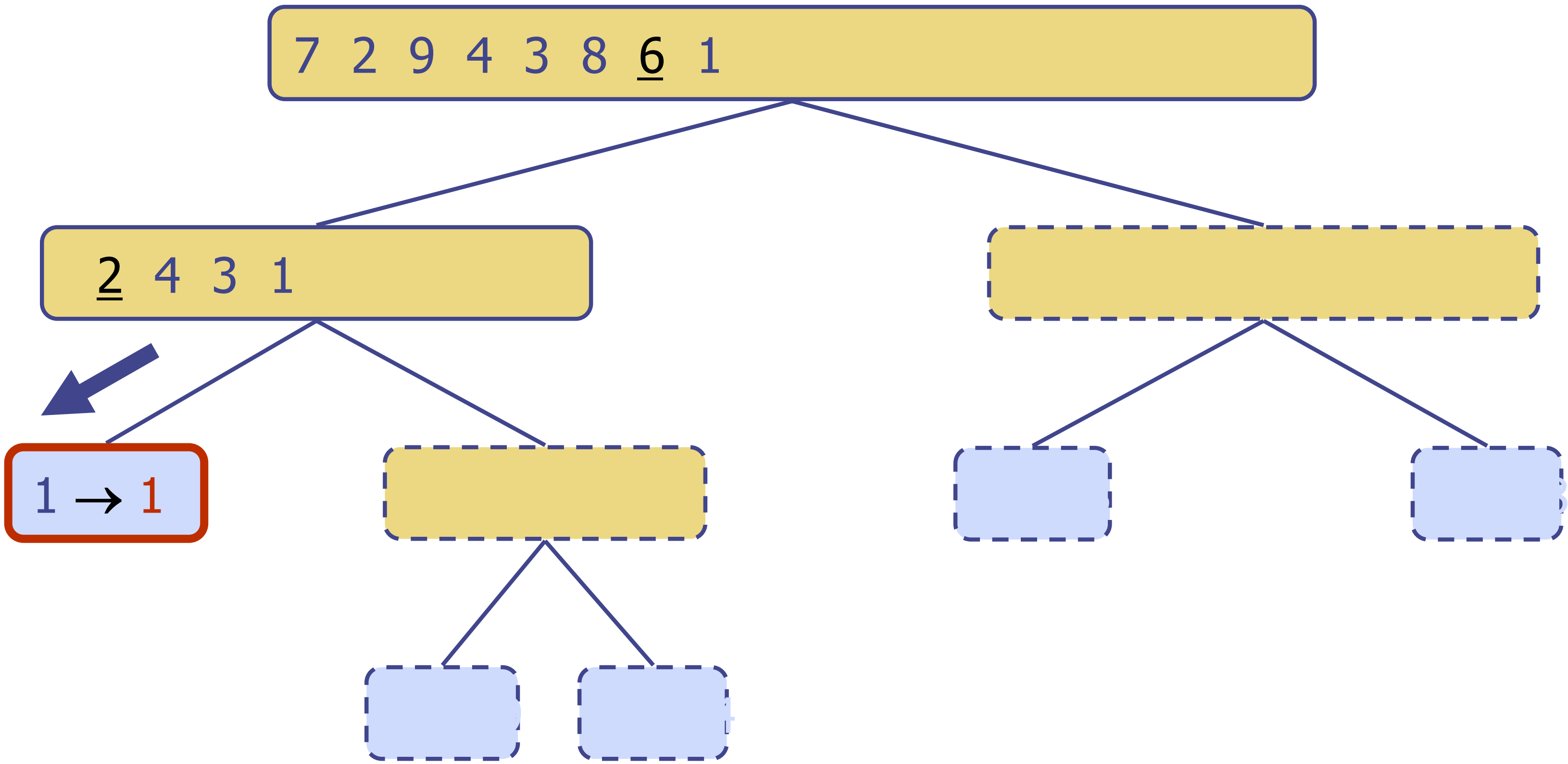
Execution Example (cont.)

- Partition, recursive call, pivot selection



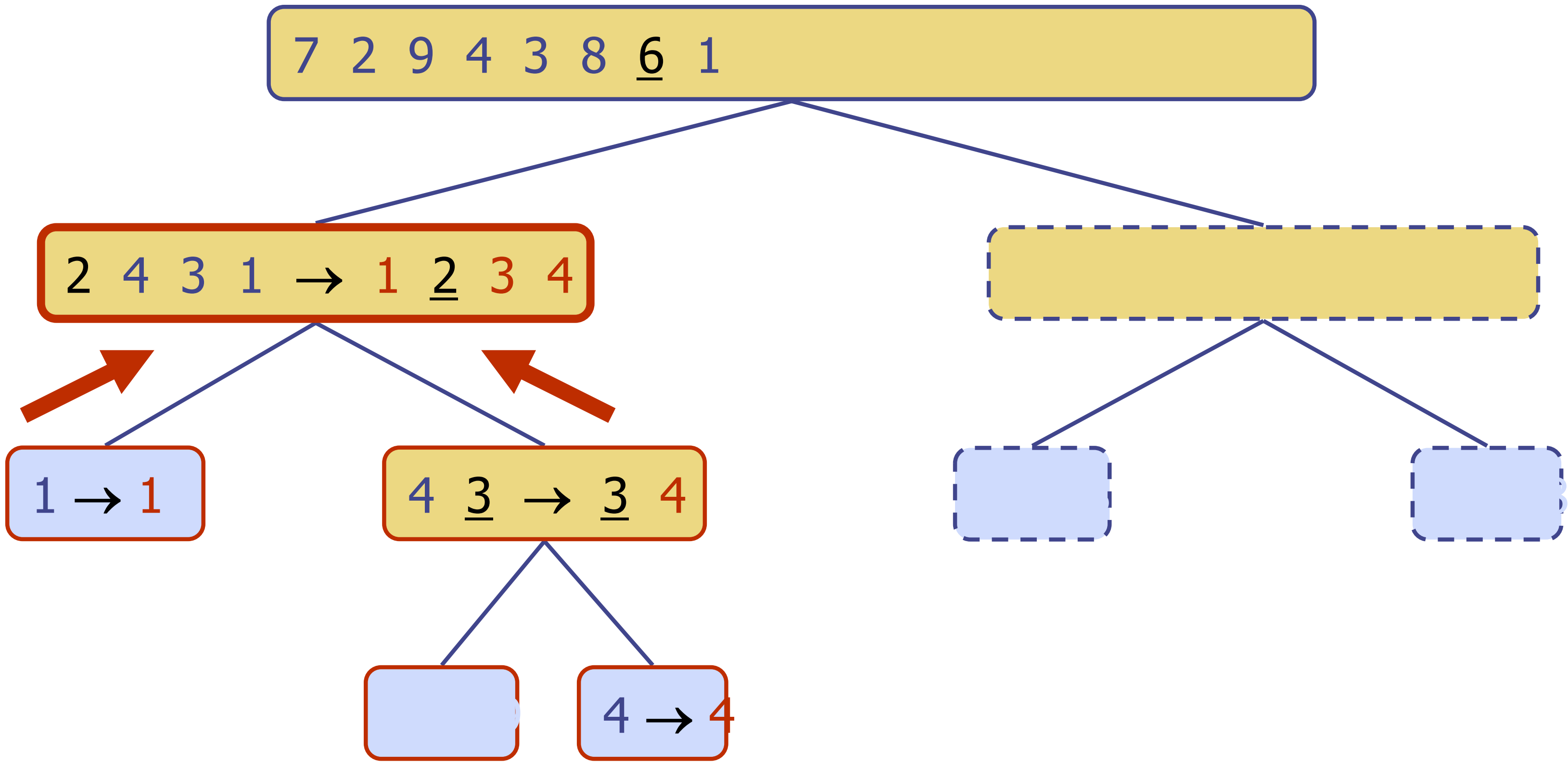
Execution Example (cont.)

- Partition, recursive call, base case



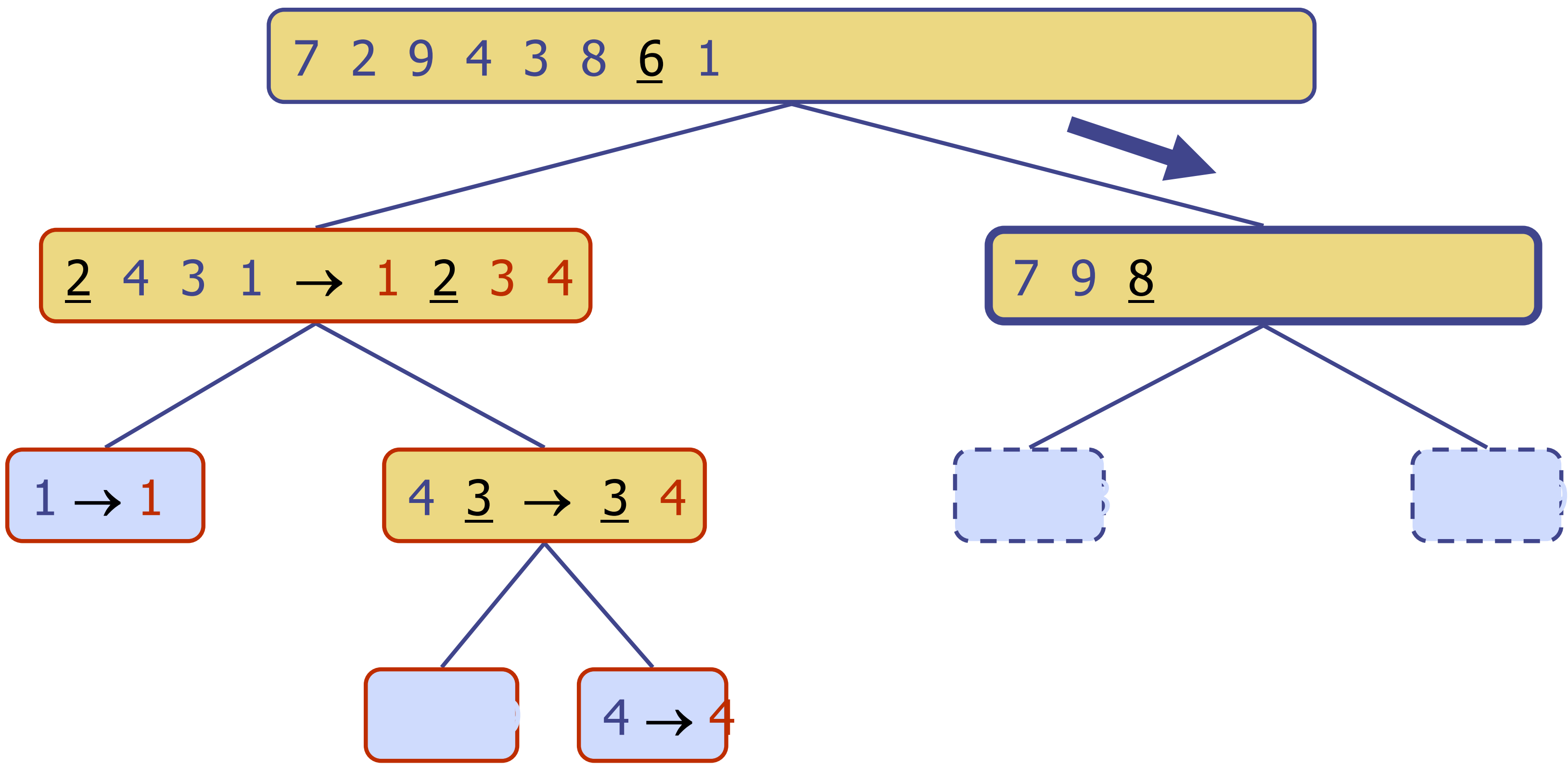
Execution Example (cont.)

- Recursive call, …, base case, join



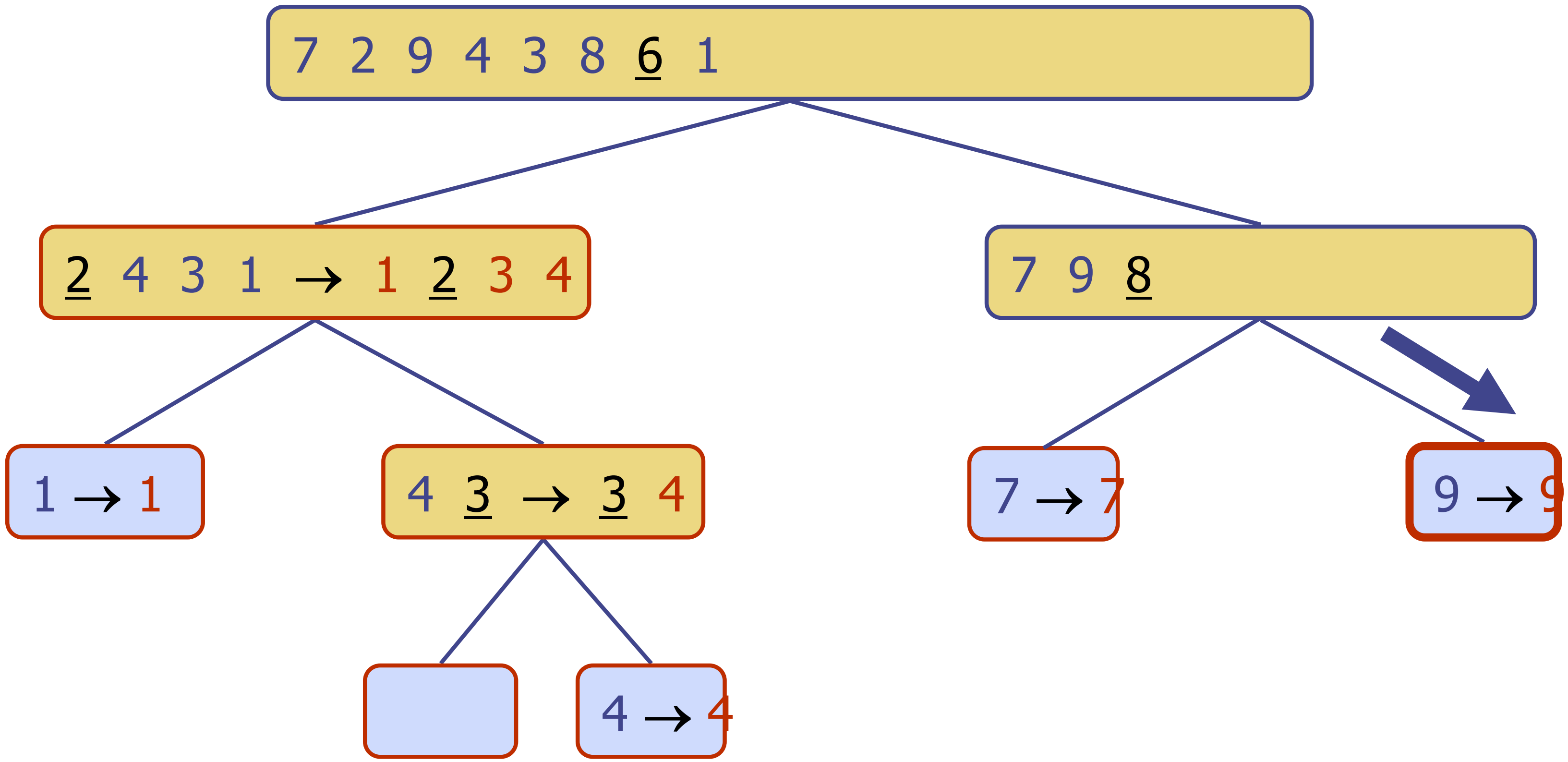
Execution Example (cont.)

- Recursive call, pivot selection



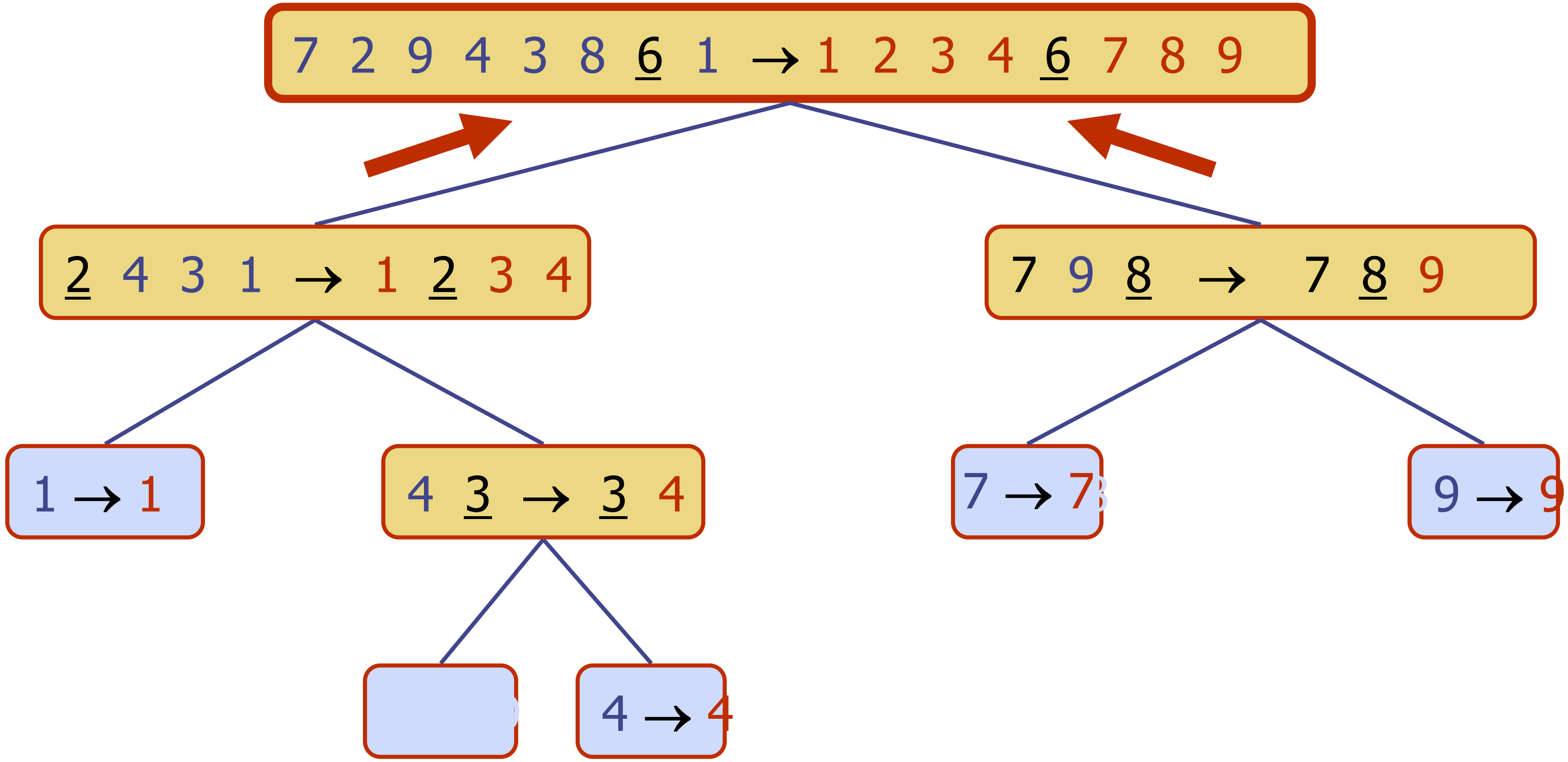
Execution Example (cont.)

- Partition, …, recursive call, base case



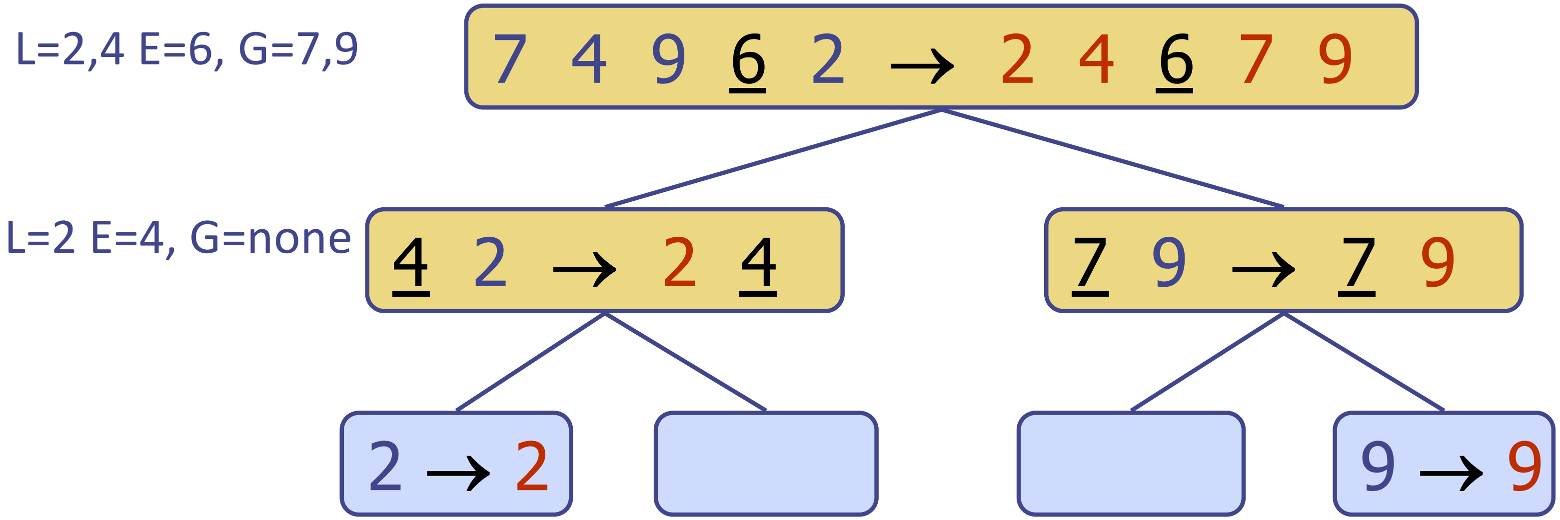
Execution Example (cont.)

- Join, join



Quick-Sort Tree

- An execution of quick-sort is depicted by a binary tree
  - Each node represents a recursive call of quick-sort and stores
    - Unsorted sequence before the execution and its pivot
    - Sorted sequence at the end of the execution
  - The root is the initial call
  - The leaves are calls on subsequences of size 0 or 1



Partition

- We partition an input sequence as follows:
  - We remove, in turn, each element y from S and
  - We insert y into L, E or G, depending on the result of the comparison with the pivot x
- Each insertion and removal is at the beginning or at the end of a sequence, and hence takes $O(1)$ time
- Thus, the partition step of quick-sort takes $O(n)$ time

```
Algorithm partition(S, p)
    Input sequence S, position p of pivot
    Output subsequences L, E, G of the
        elements of S less than, equal to,
        or greater than the pivot, resp.
    L, E, G ← empty sequences
    x ← S.erase(p)
    while ¬S.empty()
        y ← S.eraseFront()
        if y < x
            L.insertBack(y)
        else if y = x
            E.insertBack(y)
        else { y > x }
            G.insertBack(y)
    return L, E, G
```

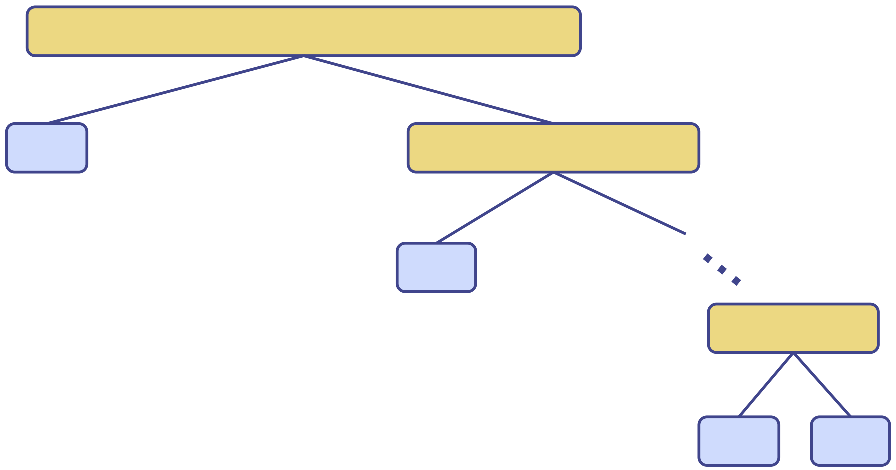
Worst-case Running Time

- The worst case for quick-sort occurs when the pivot is the unique minimum or maximum element
- One of L and G has size $n - 1$ and the other has size 0
- The running time is proportional to the sum

$$n + (n - 1) + \ldots + 2 + 1$$

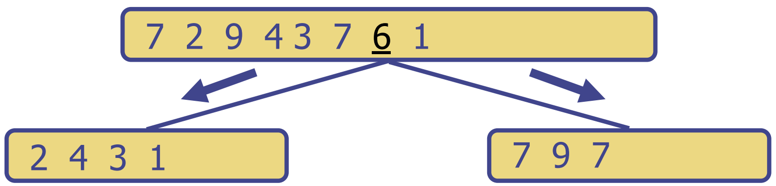
- Thus, the worst-case running time of quick-sort is $O(n^2)$

depth	time
0	n
1	$n - 1$
…	…
$n - 1$	1

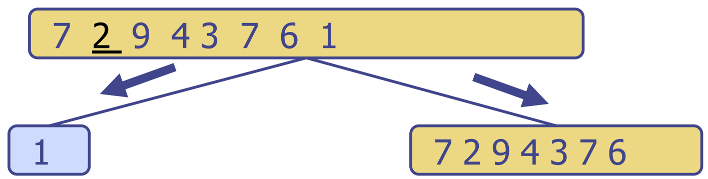


Expected Running Time (1)

- Consider a recursive call of quick-sort on a sequence of size s
  - **Good call:** the sizes of L and G are each less than $3s/4$ (“unbiased to some degree”)
  - **Bad call:** one of L and G has size greater than $3s/4$ (“biased to some degree”)

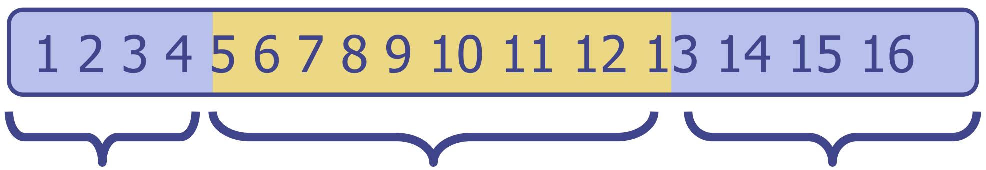


Good call



Bad call

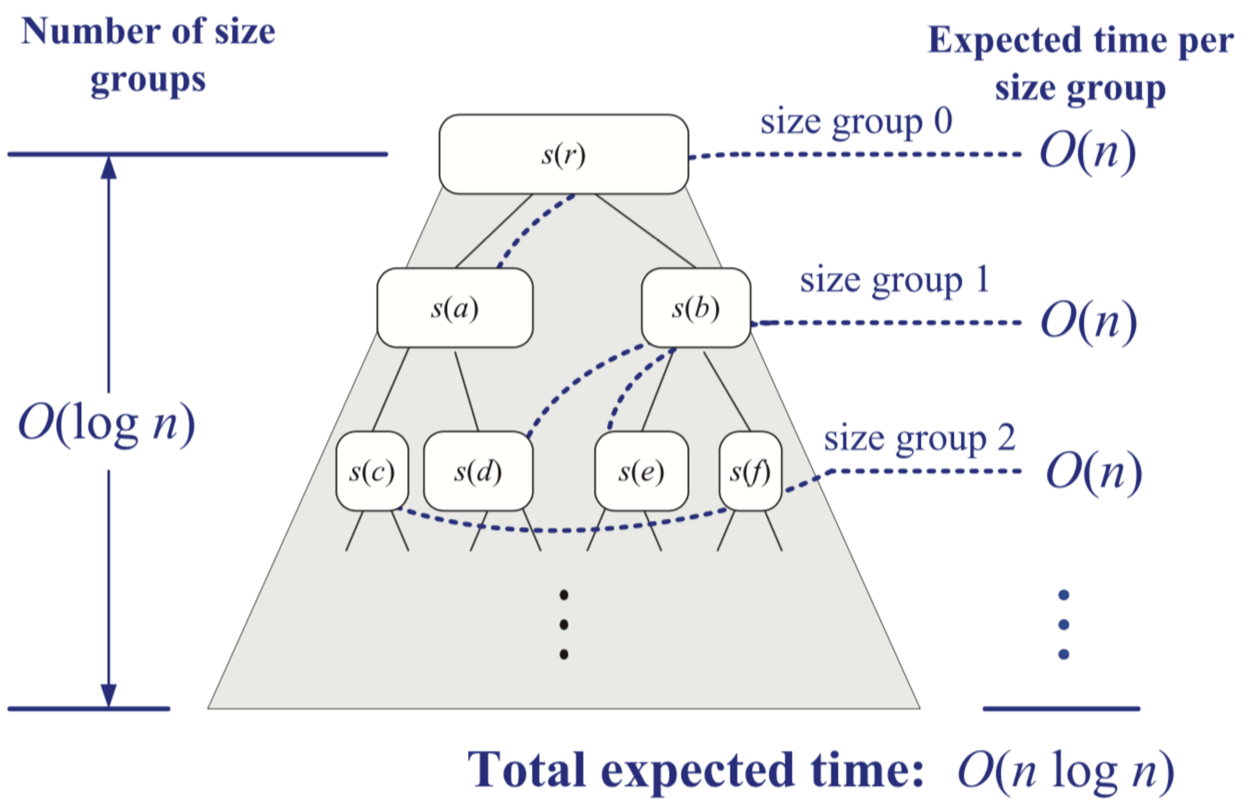
- A call is good with probability $1/2$
  - 1/2 of the possible pivots cause good calls:



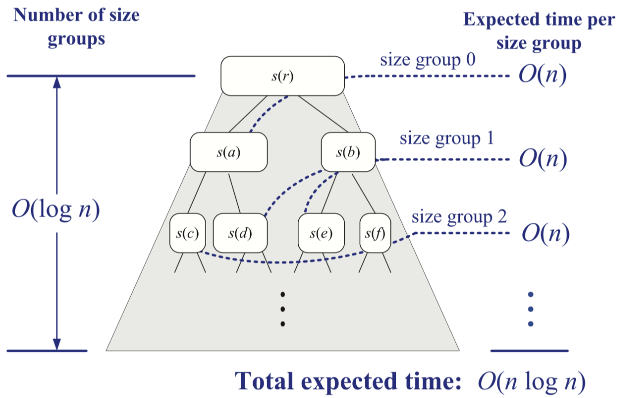
Bad pivots **Good pivots** **Bad pivots**

Expected Running Time (2)

- Consider a binary tree T used in the Quick-sort.
- Definition
  - A node v (a collection of elements) in T is said to be in size group i if $\left(\frac{3}{4}\right)^{\{i+1\}} n \leq$ the size of v's subproblem $\leq \left(\frac{3}{4}\right)^{\{i\}} n$
  - Thus, every node is in some size group (e.g., the root node is in size group 0)

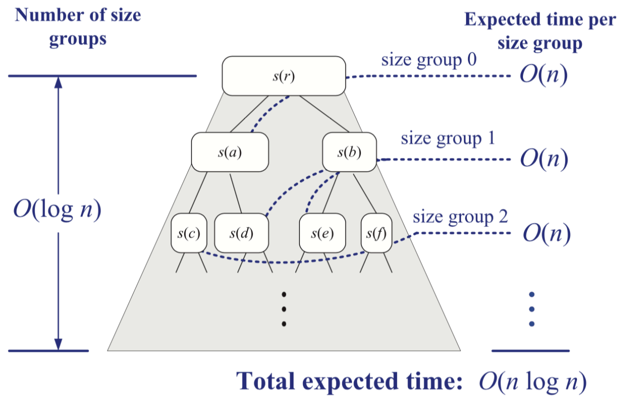


Expected Running Time (3)



- Q1. How many size groups?
  - (Ans) i, such that $\left(\frac{3}{4}\right)^{\{i\}} n = 1$, i.e., ***i = 2log$_{4/3}$n***
- Q2. What is the expected time spent working on all the subproblems for nodes in size group *i* (which we denote by T)?
  - If the answer is O(n), then we are done, because the number of size groups * expected running time for each size group = n * log n.
  - T = sum of the expected times for each node, say *v*, in size group *i* (linearity of expectation). Thus, our question is "what is the expected time for a node in size group i"?
  - v's subproblem may be either of good call or bad call.
  - (Two facts) Since a probability of good call is ½,
    - (i) The expected number of consecutive calls before a good call is 2 (i.e., constant)
    - (ii) As soon as we have a good call for node v (in size group i), its children will be in size groups higher than i. (because at least ¾ reduction of the original size happens)

Expected Running Time (4)



- Q1. How many size groups?
  - (Ans) i, such that $\left(\frac{3}{4}\right)^{\{i\}} n = 1$, i.e., ***i = 2log$_{4/3}$n***

- Q2. What is the expected time spent working on all the subproblems for nodes in size group *i* (which we denote by T)?
  - Thus, for any elements x in the input list, the expected number of nodes in size group i containing x in their subproblems is 2. (on average, constant number times of being at a bad call group and then move to the size group higher than i)
  - → Expected total size of all the subproblems in size group i is 2n
    - → Non-recursive work we perform for any subproblem is proportional to its size
    - → Expected time per each size group is O(n)

- Thus,
  - log n size groups & n computations per each size group
  - → O(n log n)

Summary of Sorting Algorithms

Algorithm	Time	Notes
selection-sort	$O(n^2)$	▪ in-place ▪ slow (good for small inputs)
insertion-sort	$O(n^2)$	▪ in-place ▪ slow (good for small inputs)
quick-sort	$O(n \log n)$ expected	▪ in-place, randomized ▪ fastest (good for large inputs)
heap-sort	$O(n \log n)$	▪ in-place ▪ fast (good for large inputs)
merge-sort	$O(n \log n)$	▪ sequential data access ▪ fast (good for huge inputs)

Questions?