COMP2054-ADE

Lecturer: Andrew Parkes
http://www.cs.nott.ac.uk/~pszajp/

Quicksort

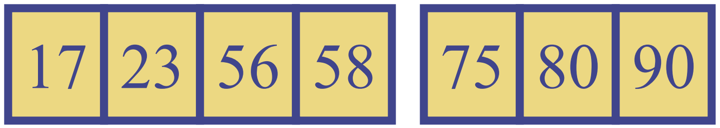
Motivations

- In merge sort the `divide' is simple, and the `merge' (relatively) complicated
- Can we swap this and make the 'merge' simple?
  - Answer: make the `divide' more complicated so that the merge becomes `concatenate'
- Analogy: sort a pack of cards by

1. divide into 'red' and 'black' cards
2. divide by suit (red into hearts and diamonds,…)
3. divide by value …

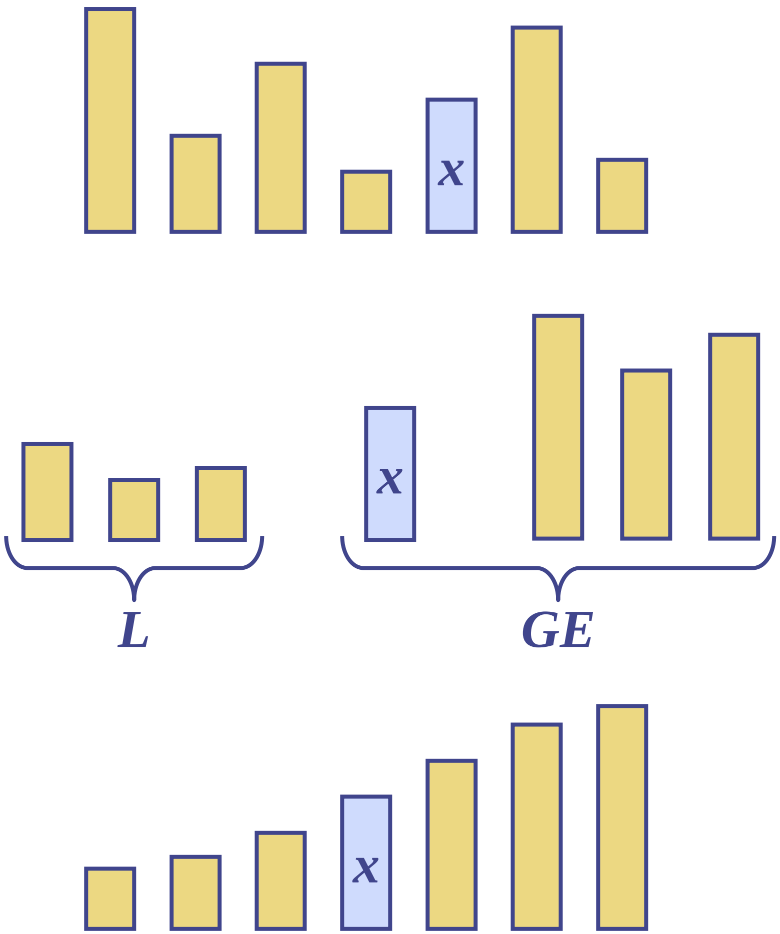
When is `merge' simple?

- When the lists A and B are sorted and known to be in disjoint ordered ranges
  - all of elements of A are smaller than all those of B
- If A and B are stored as consecutive sub-arrays, then merge needs no work at all:
  - Just "forget the boundary"



Quick-Sort

- Quick-sort is a (randomized) sorting algorithm based on the divide-and-conquer paradigm:
  - Divide: pick an element x (called pivot) and partition S into
    - L: elements less than x
      - *Have to be careful it is not empty*
    - GE: elements greater than or equal to x
    - *Pivot is often picked as a random element*
  - Recur: sort L and GE
  - Conquer: join L, GE



Partition of lists (using extra workspace)

- Suppose store L, EG as separate structures (e.g. as arrays, vectors or lists)
- 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 end) of the sequence, and hence takes $O(1)$ time
- Thus, the partition step of quick-sort takes $O(n)$ time

“In-place” or “extra workspace”?

- For sorting algorithms (and algorithms in general) an important issue can be how much extra working space they need besides the space to store the input
- “**In-place**” means they only a “little” extra space (e.g. O(1)) is used to store data elements.
  - The input array is also used for output, and only need a few temporary variables
  - Exercise: check that bubble-sort is “in-place”
  - Previous “merge” used extra O(n) array (can be made in-place, but messy and so we ignore this option)

Partitioning arrays "in-place"

- Perform the partition using two indices to do a "2-way split" of S into L and E+G.

j k

3 2 5 1 0 7 3 5 9 2 7 9 8 9 7 **6** 9 (pivot = 6)

- Repeat until j and k cross (or meet, until j >= k):
  - Scan j to the right until finding an element ≥ pivot.
  - Scan k to the left until finding an element < pivot.
  - Swap elements at indices j and k

j k

3 2 5 1 0 $\boxed{7}$ 3 5 9 $\boxed{2}$ 7 9 8 9 7 **6** 9

The scans are not done in 'lock step' but independently work inwards. Do some examples and make sure you understand how and why this works!!

Partitioning

- Partitioning is actually more subtle
- Have to make sure that we make progress – specifically that neither of the partitions are empty
  - GE always contains the pivot, so is okay.
  - But L could be empty
    - So the pivot should not be a minimum element
- Or simpler is to do a “3-way” split:
  - check and move the pivot to the left and do a 3-way split into L, {pivot}, E+G
  - And so always ensure a one copy of the pivot (there could be many) is moved to the middle.

Exercise (offline)

- Write Java code to do the partition and check that it works on some examples
  - (It is only 10-20 lines of code, but will greatly help clarify the algorithm.)
- Investigate and explore some versions of how to do it effectively and efficiently

Quicksort Overall Implementation

With the previous (2-way) split:

```
public static void recQuickSort(int[] arr, int left, int right) {
   if (right - left <= 0) return;
   else {
      int border = partition(arr, left, right); // “crossing position”
      recQuickSort(arr, left, border);
      recQuickSort(arr, border+1, right);
   }
}
```

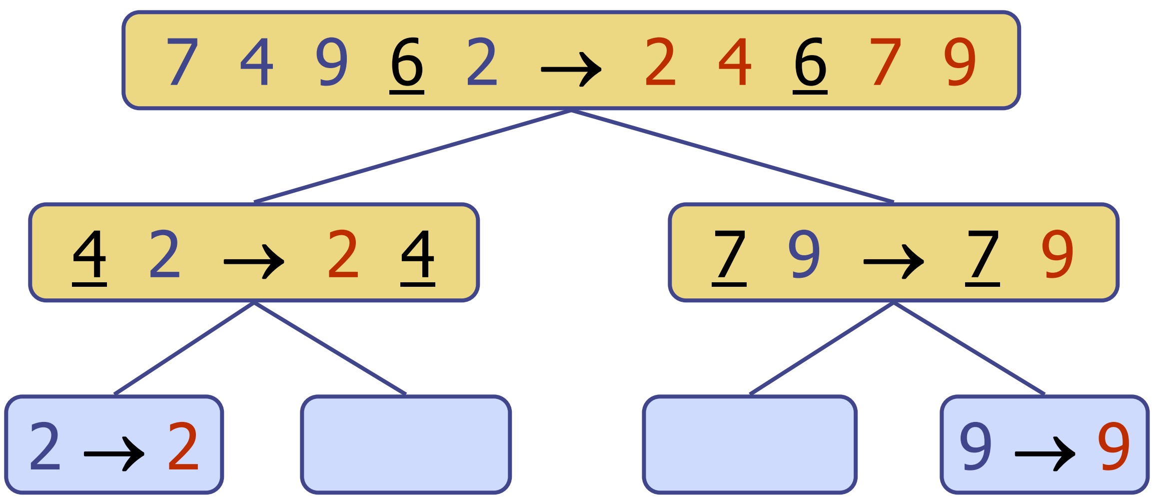
Quicksort Overall Implementation

With a 3-way split (assuming just one copy of the pivot):

```
public static void recQuickSort(int[] arr, int left, int right) {
   if (right - left <= 0) return;
   else {
      int border = partition(arr, left, right); // pivot position
      recQuickSort(arr, left, border-1);
      recQuickSort(arr, border+1, right);
   }
}
```

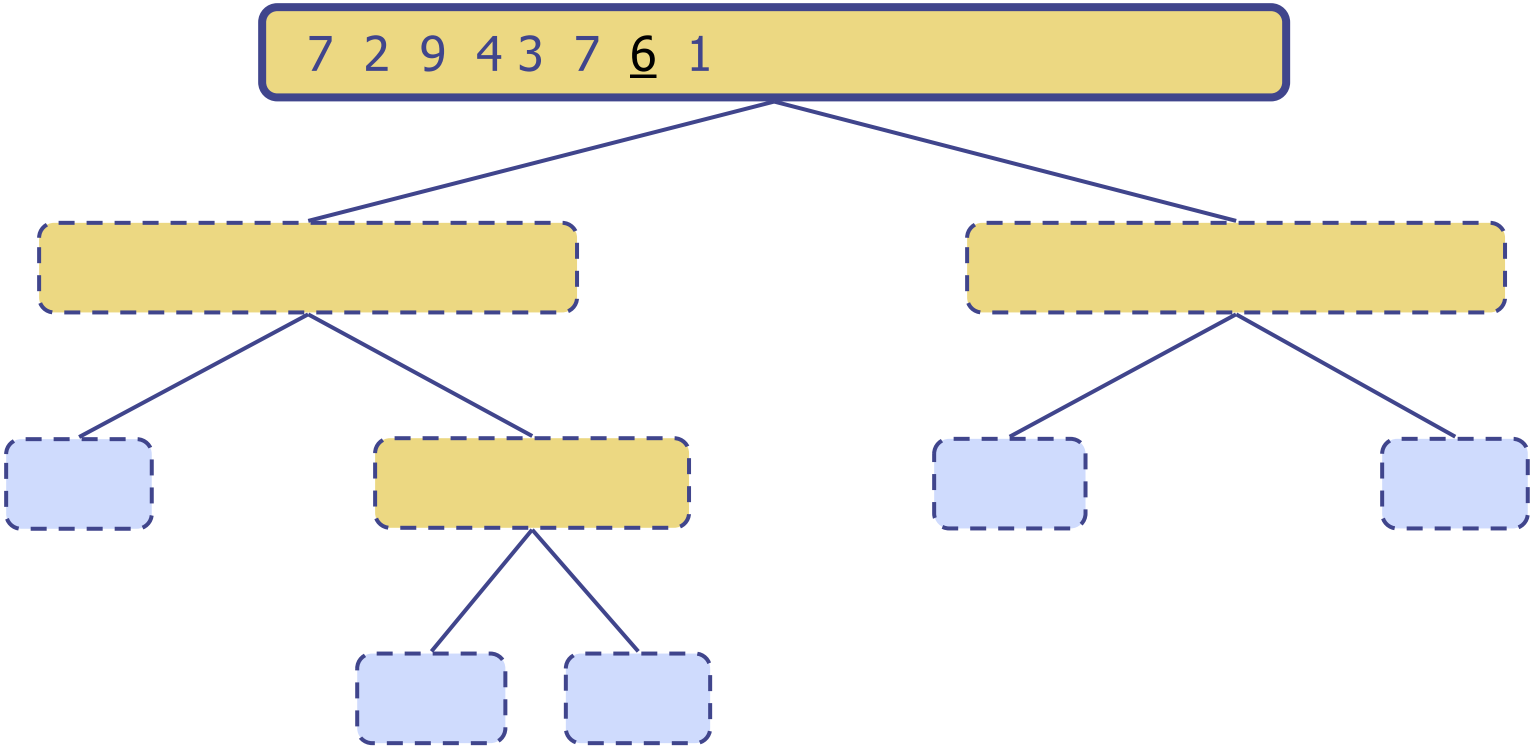
Quick-Sort Tree

- 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
  - Example shows 3-way split (one copy of the pivot is `removed' on each partition).



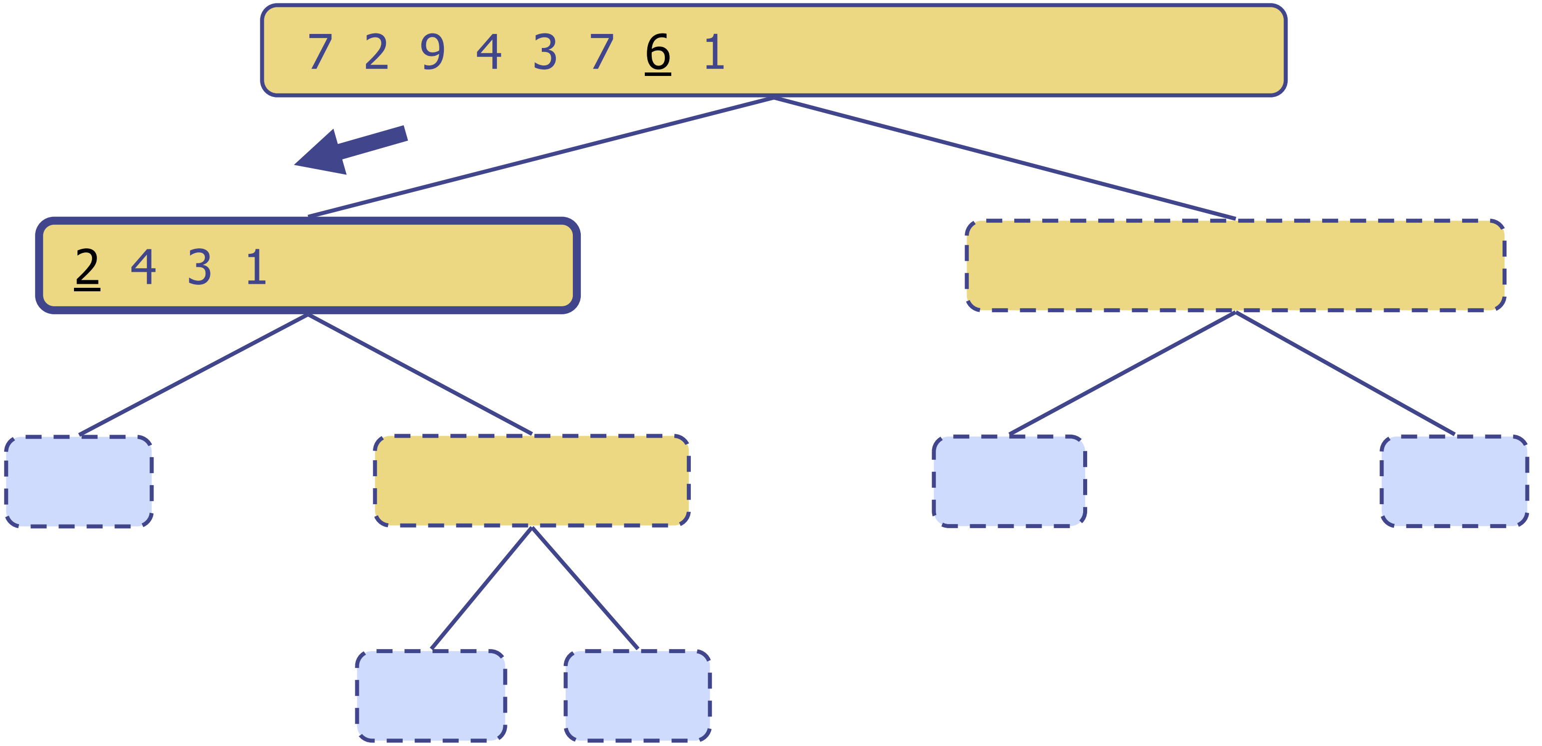
Execution Example

- Pivot selection



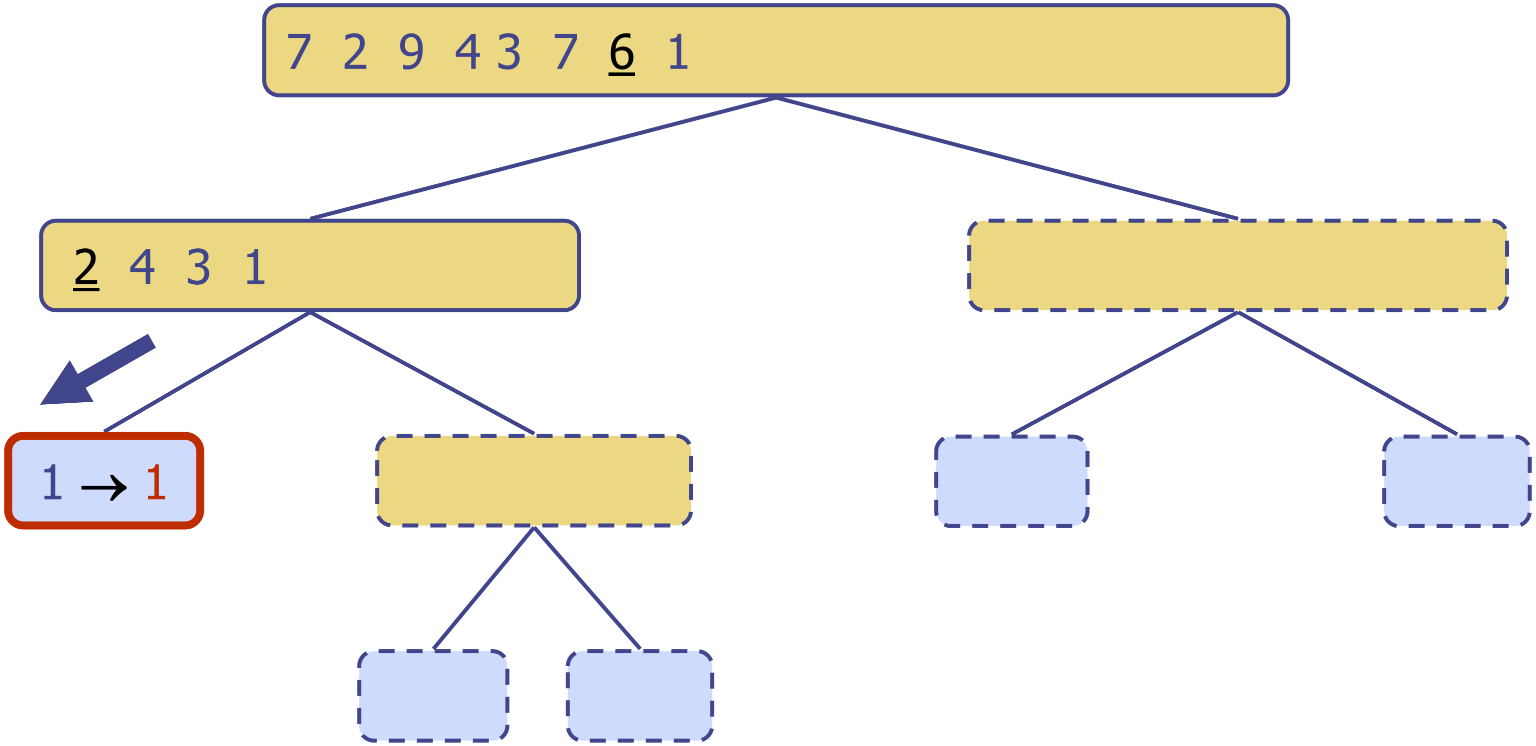
Execution Example (cont.)

- Partition, recursive call, pivot selection



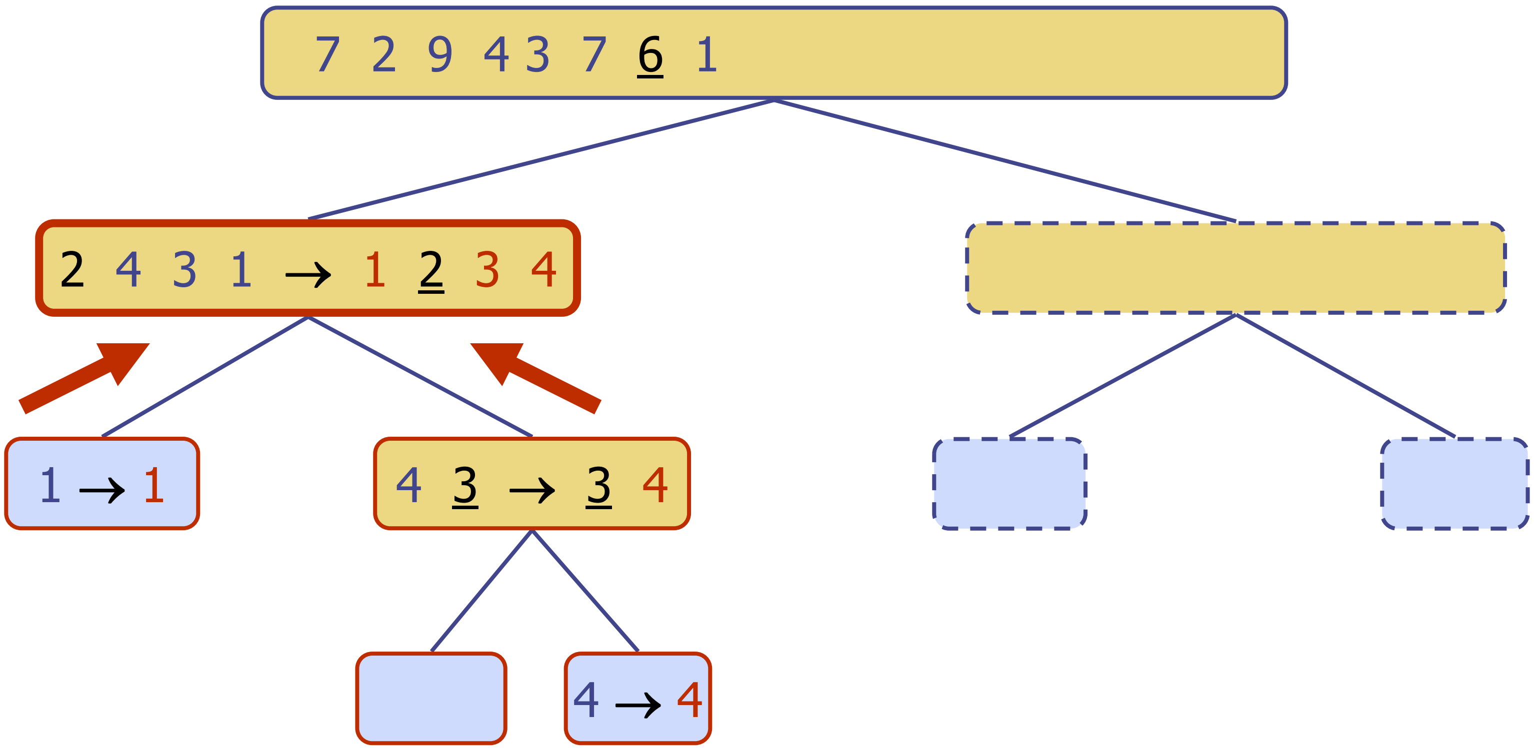
Execution Example (cont.)

- Partition, recursive call, base case



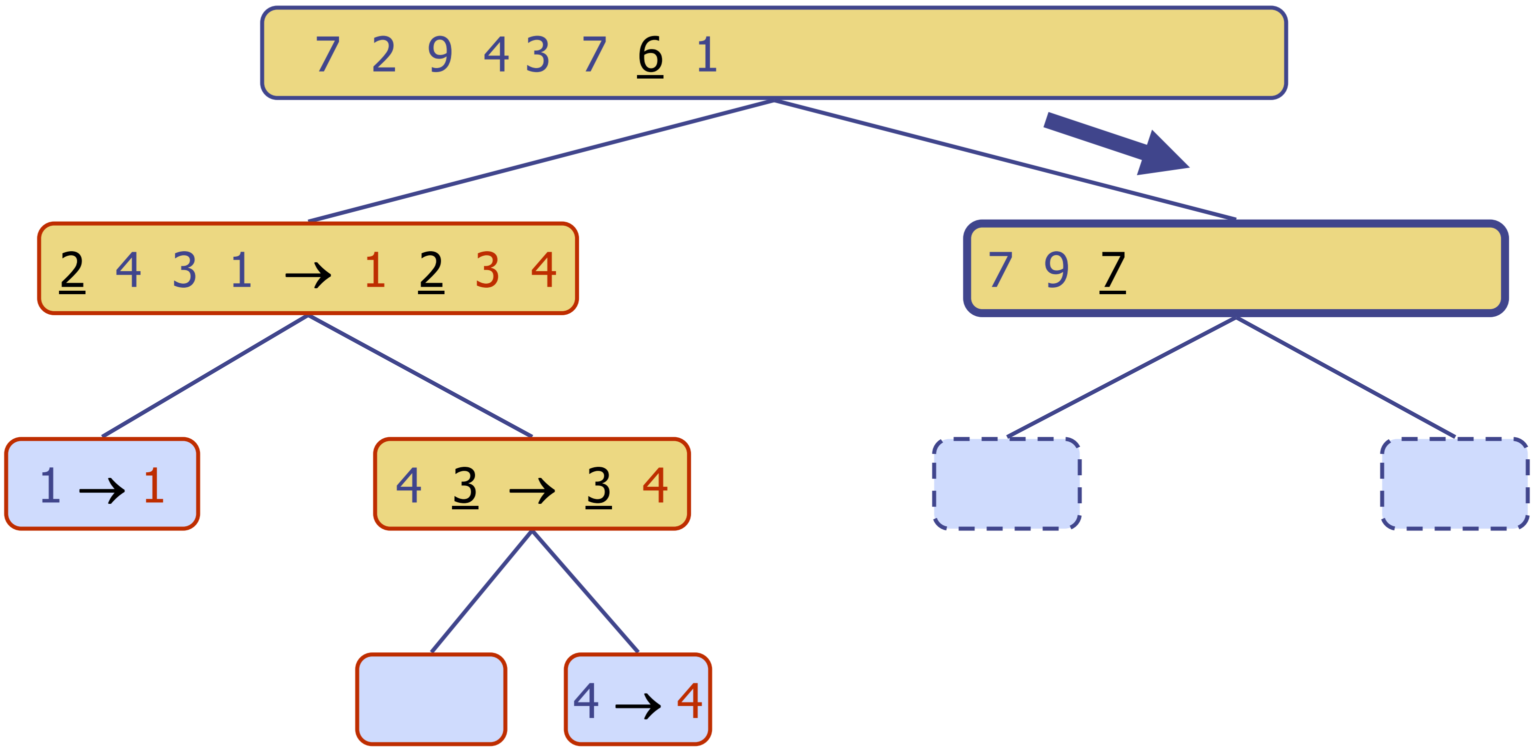
Execution Example (cont.)

- Recursive call, …, base case, join



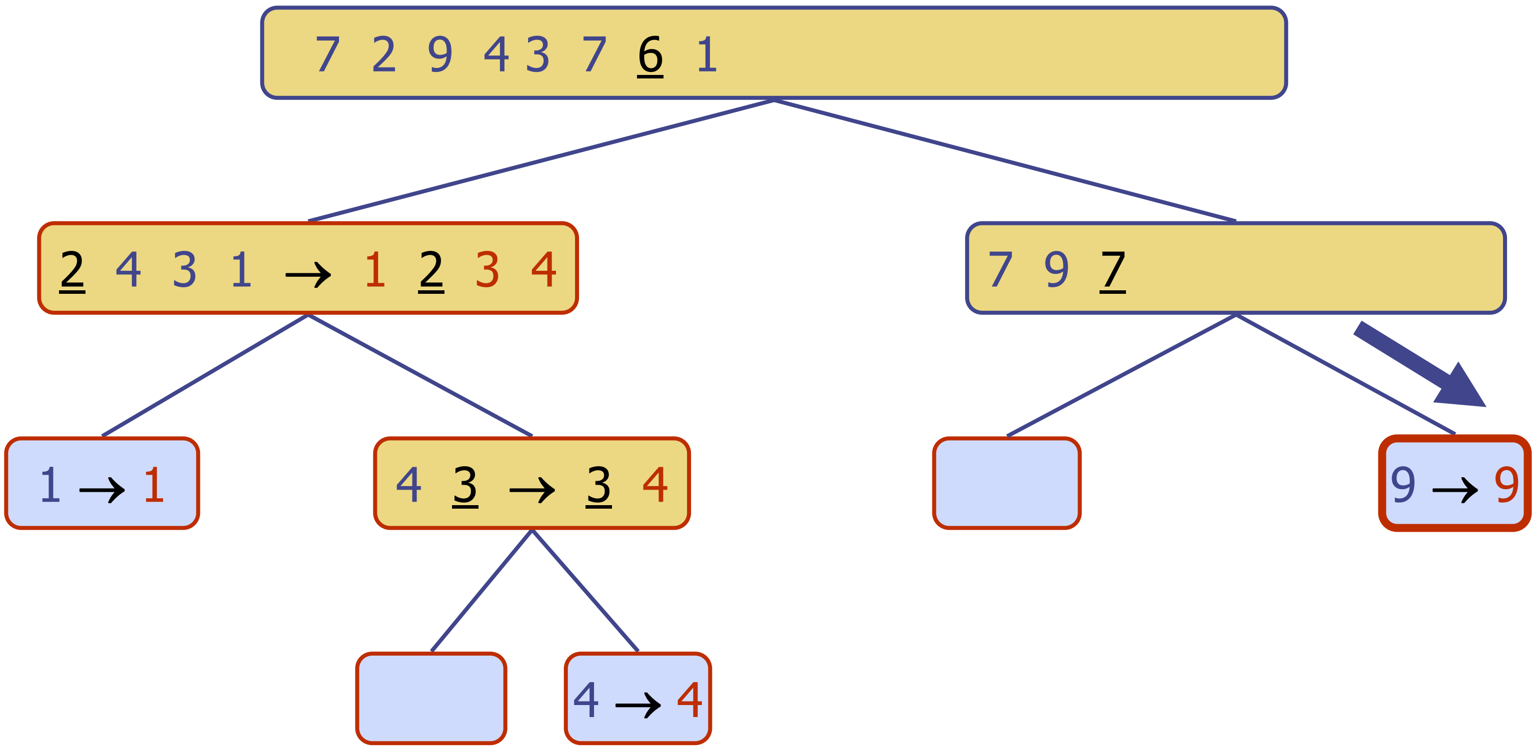
Execution Example (cont.)

- Recursive call, pivot selection



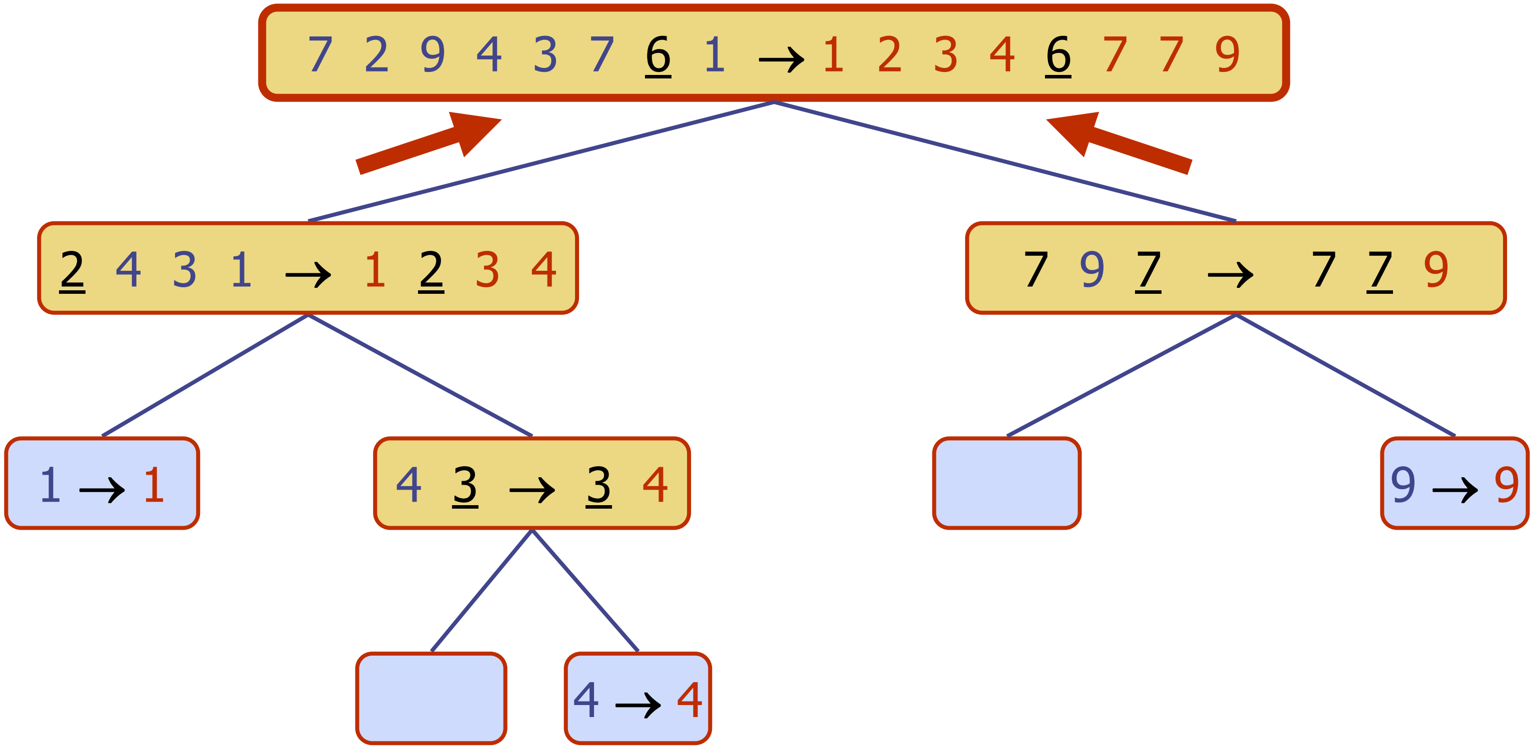
Execution Example (cont.)

- Partition, …, recursive call, base case



Execution Example (cont.)

- Join, join



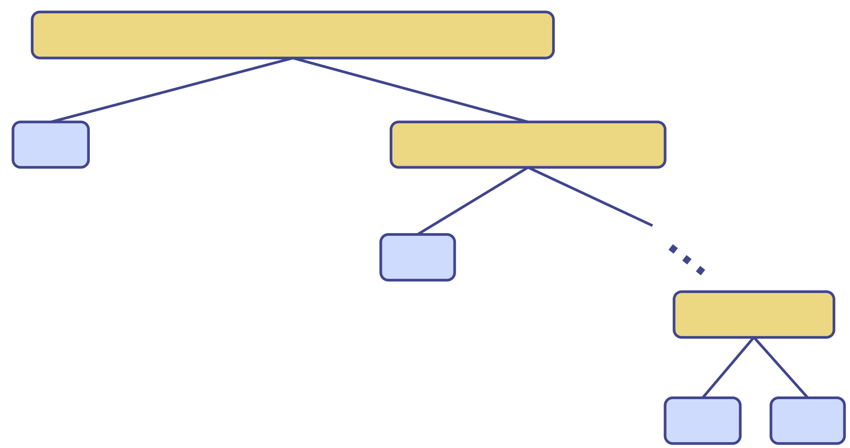
Worst-case Running Time

- The worst case for quick-sort occurs when the pivot is the unique minimum or maximum element
- One of L and E+G has size $n - 1$ and the other has size 1
  - (3-way split, so the pivot is always “removed”, so that some progress is made)
- 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



Best-case Running Time

- The best case for quick-sort occurs when the pivot is the median element
- The L and G parts are equal – the sequence is split in halves, like in merge sort
- Thus, the best-case running time of quick-sort is $O(n \log n)$

Average-case Running Time

- The average case for quick-sort: half of the times, the pivot is roughly in the middle
- Thus, the average-case running time of quick-sort is $O(n \log n)$ again
  - Detailed proof in textbook
- Basic idea: suppose that the pivot is always in the middle third
  - The both L and E+G are size at least n/3, and at most 2n/3 = n / (3/2)
  - So height of the call tree is the number of times we can divide n by (3/2)

Motivations for quicksort

- Why do we select a pivot? I.e. what advantages might quicksort ever have over mergesort?
  - Because it can be done “in-place”
    - Uses a small amount of workspace
    - Because the “merge” step is now a lot easier!!
  - The “split” is more complicated, and the merge “much” easier – but turns out that the quick-sort split is easier to do in-place than the merge-sort merge

Minimum Expectations

- For both merge- and quicksort:
  - know the algorithm and how it works on examples
  - know and be able to justify/prove their big-Oh behaviours