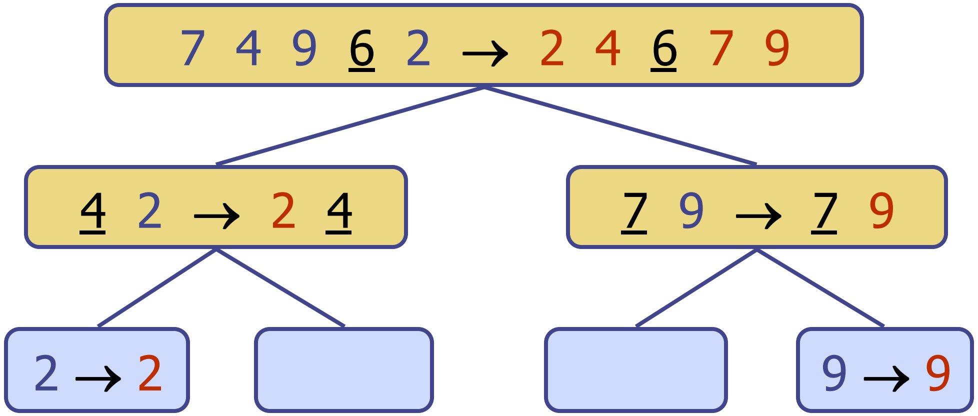
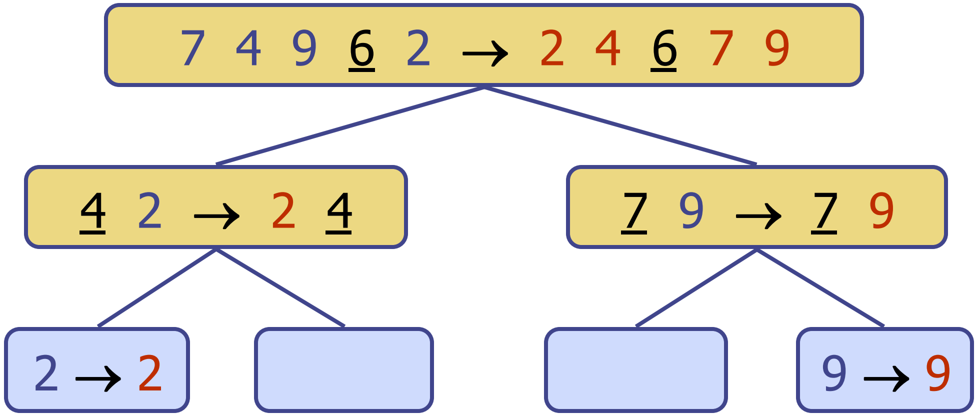
Chapter 4: Sorting



What We'll Do

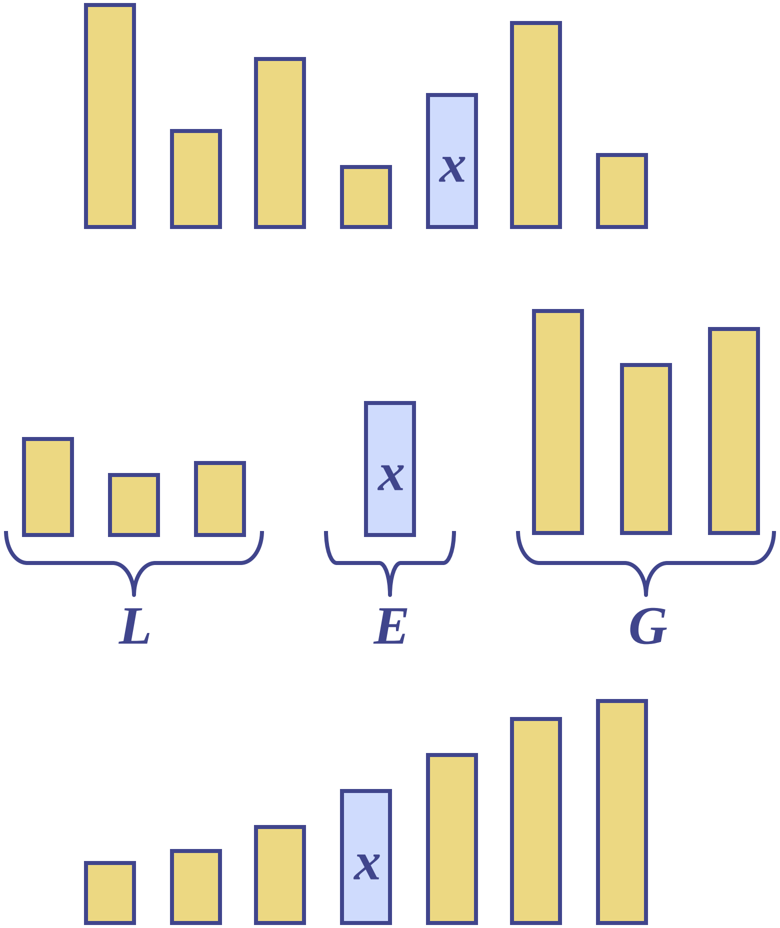
- Quick Sort
- Lower bound on runtimes for comparison based sort

Quick-Sort



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



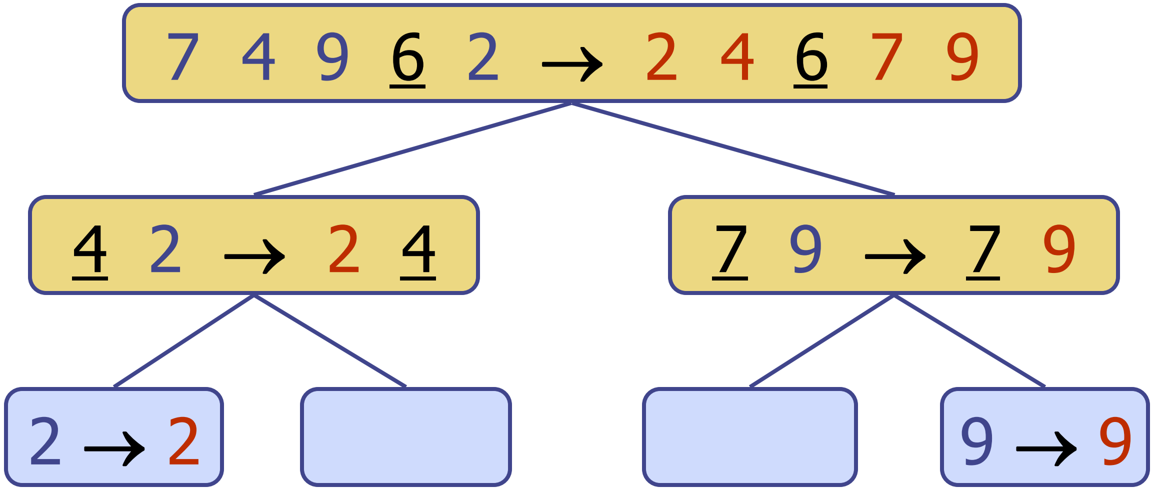
Partition

- 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.remove(p)
    while ¬S.isEmpty()
        y ← S.remove(S.first())
        if y < x
            L.insertLast(y)
        else if y = x
            E.insertLast(y)
        else { y > x }
            G.insertLast(y)
    return L, E, G
```

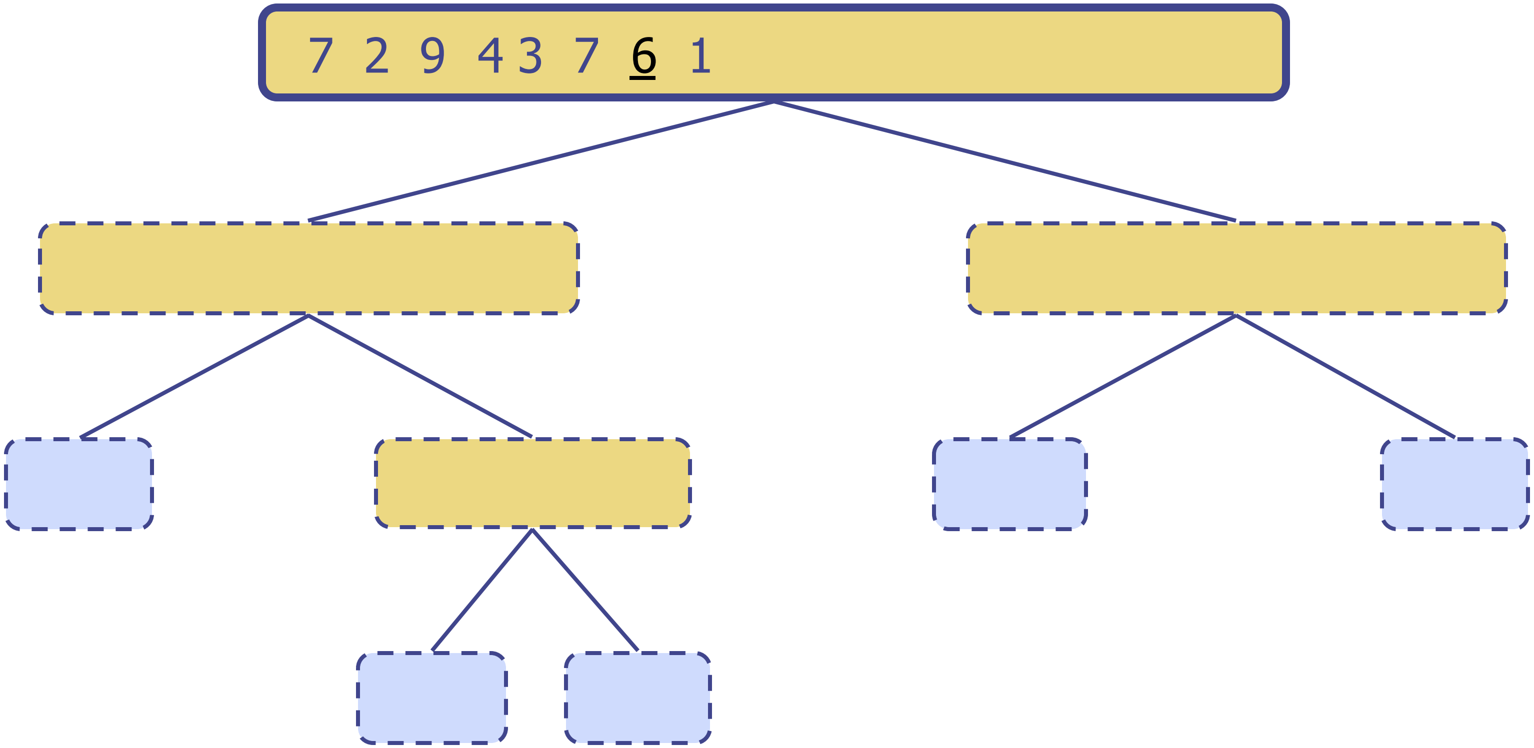
Quick-Sort Tree

- 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



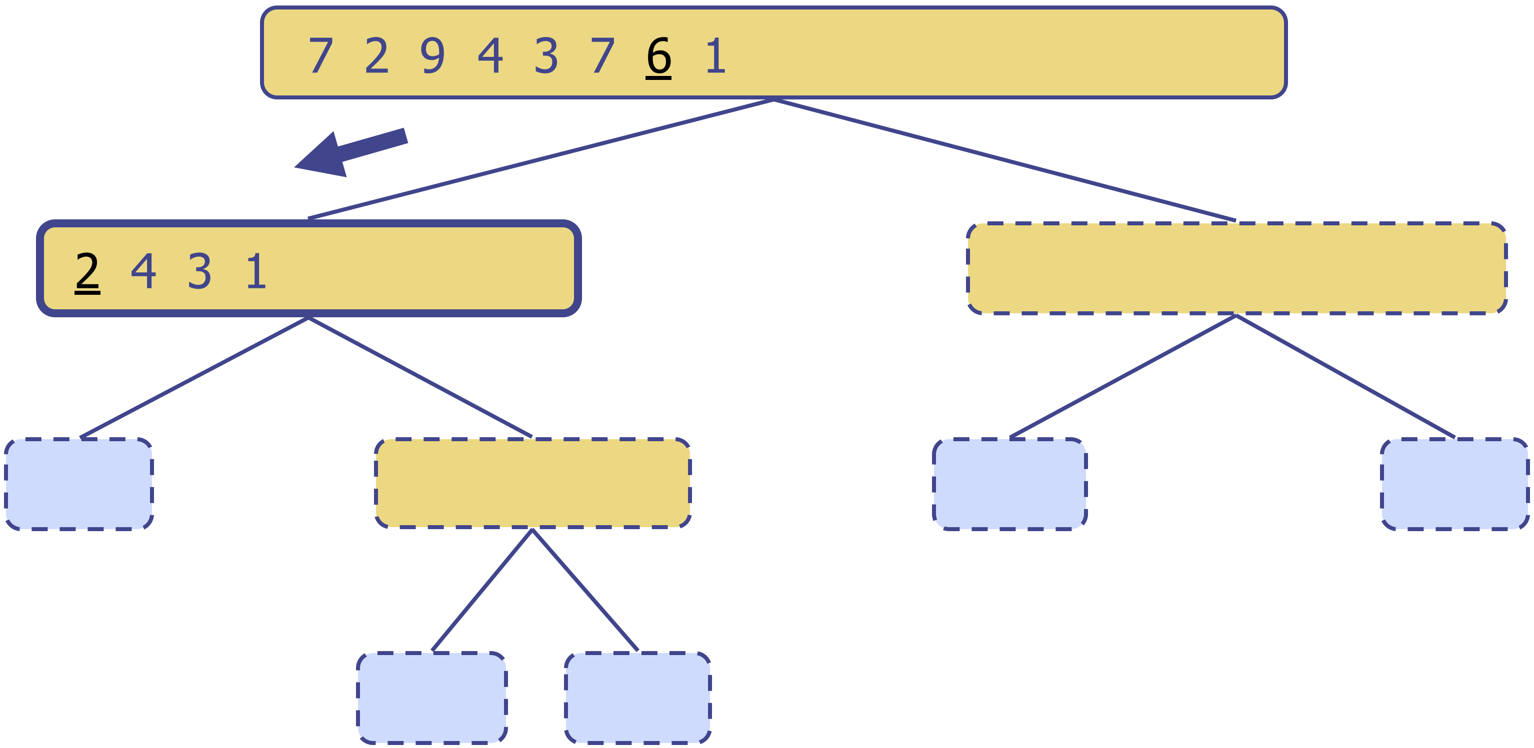
Execution Example

- Pivot selection



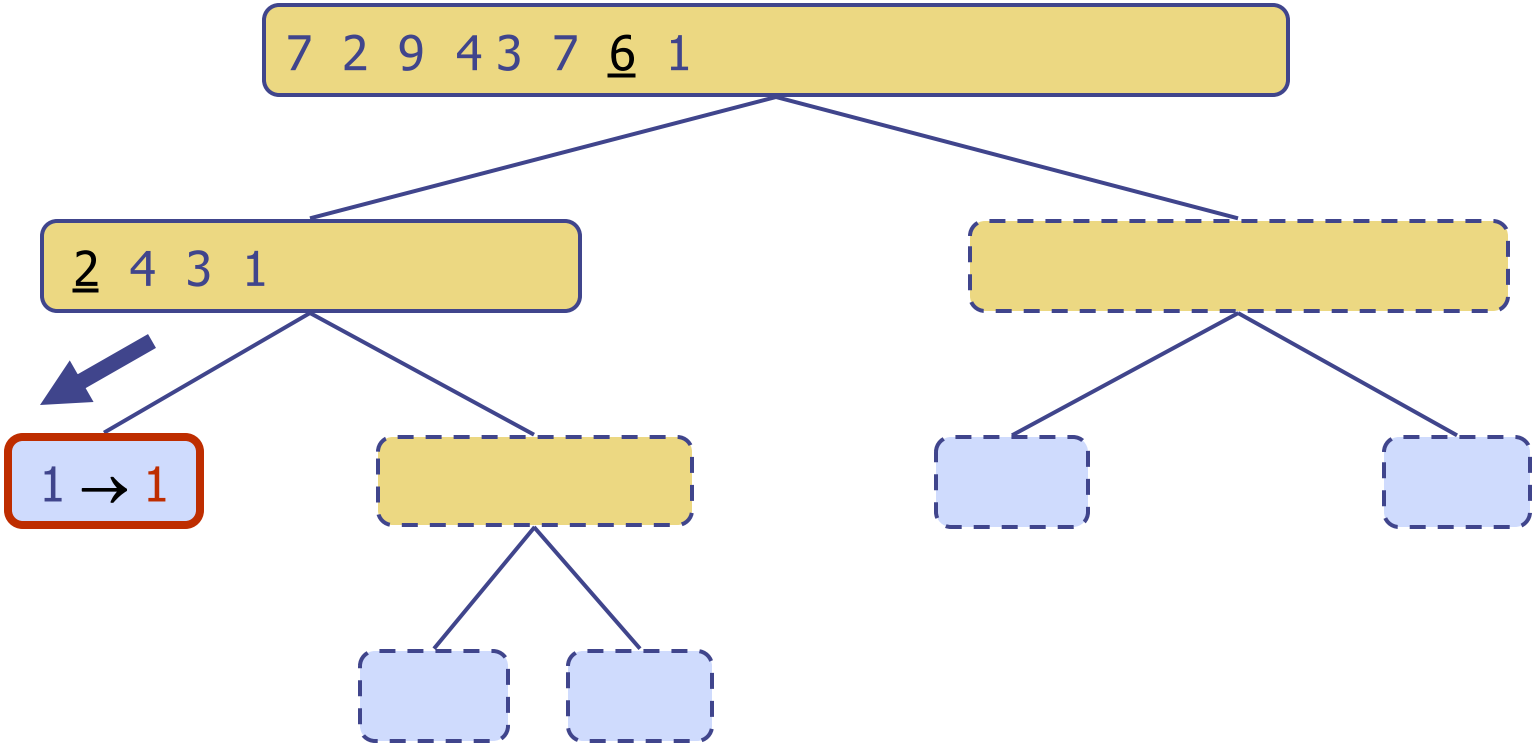
Execution Example (cont.)

- Partition, recursive call, pivot selection



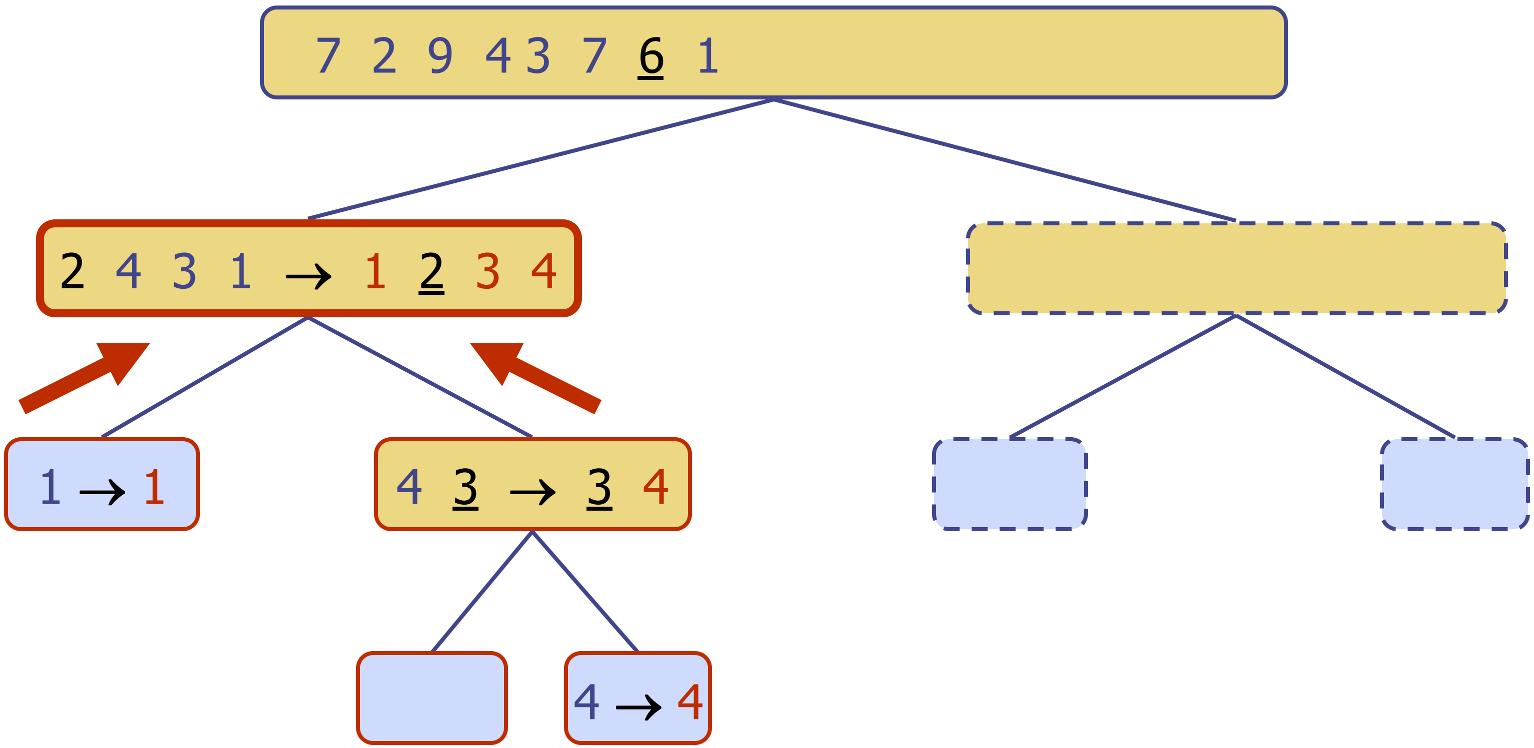
Execution Example (cont.)

- Partition, recursive call, base case



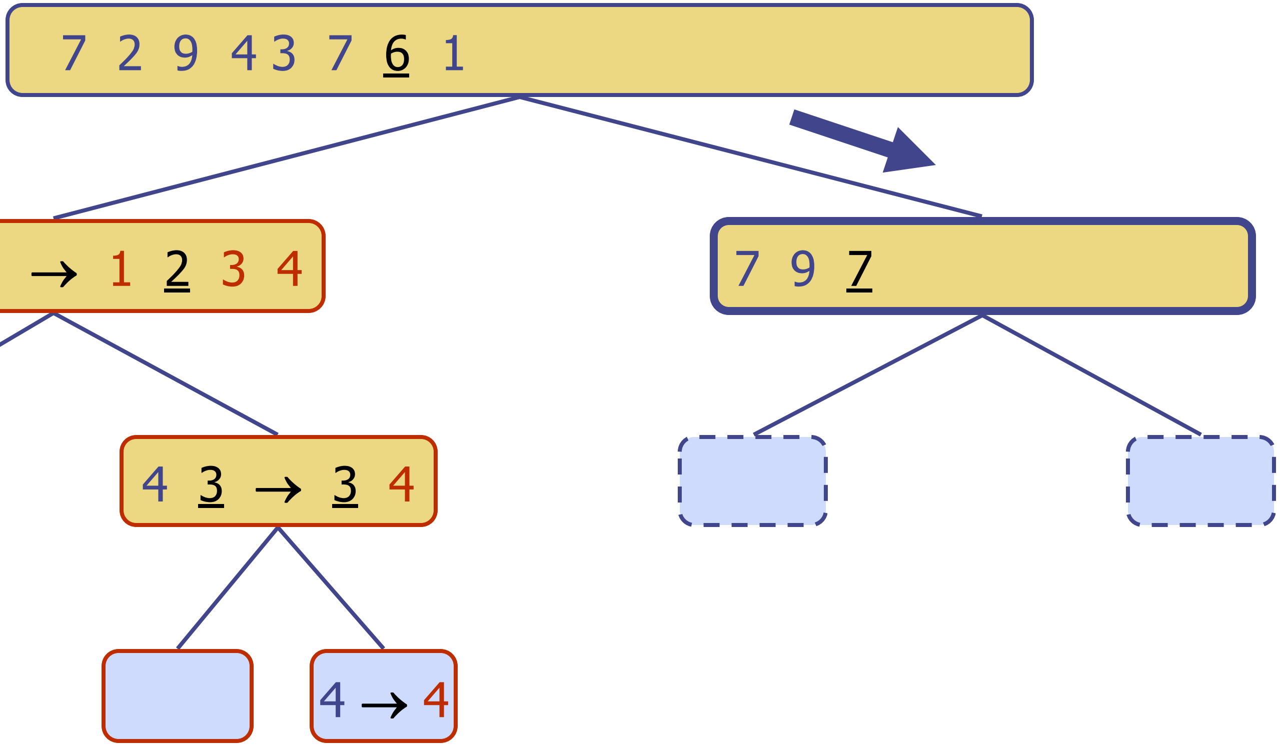
Execution Example (cont.)

- Recursive call, …, base case, join



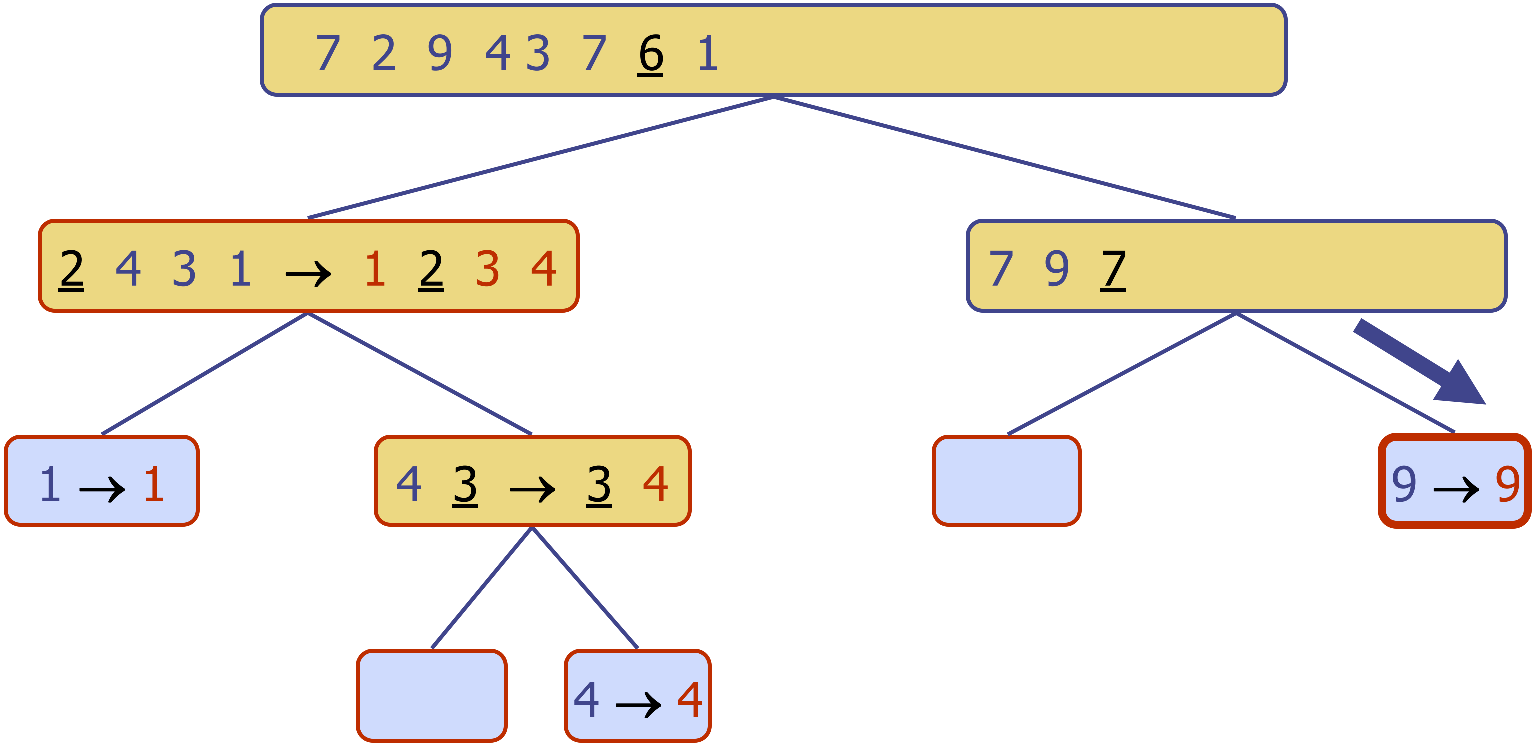
Execution Example (cont.)

- Recursive call, pivot selection



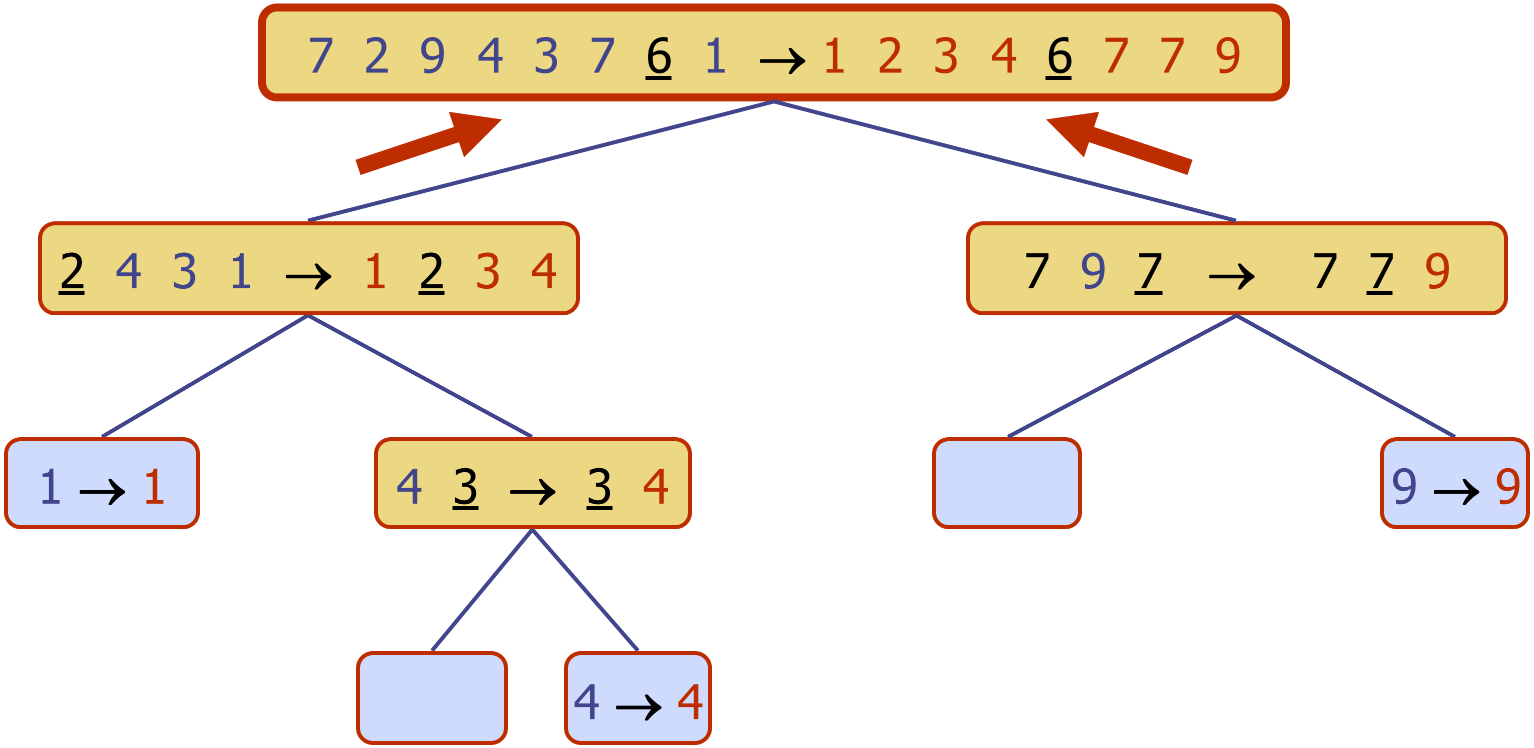
Execution Example (cont.)

- Partition, …, recursive call, base case



Execution Example (cont.)

- Join, join

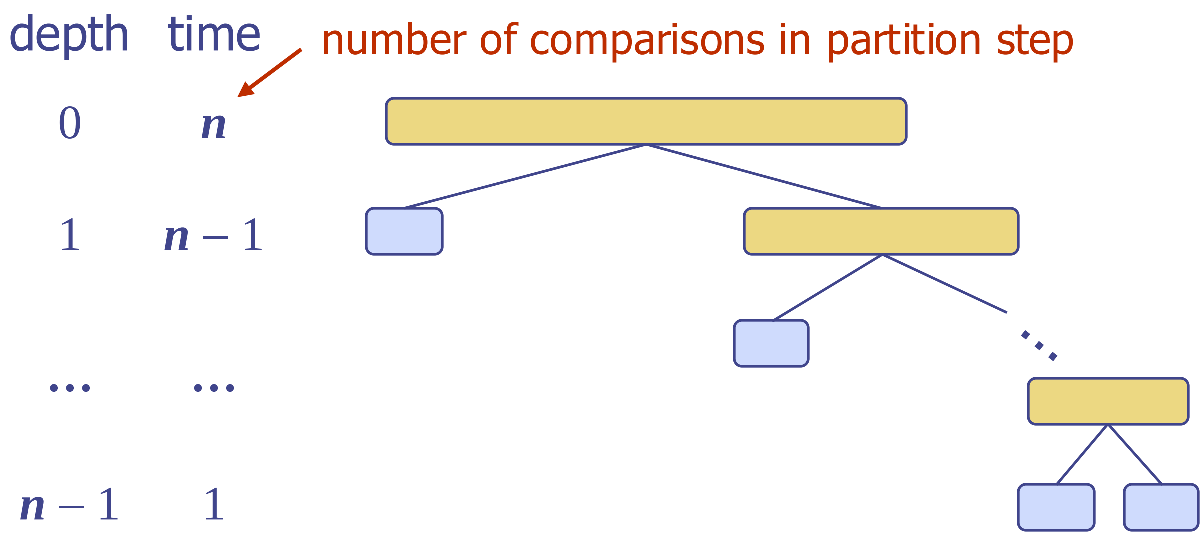


Worst-case Running Time

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

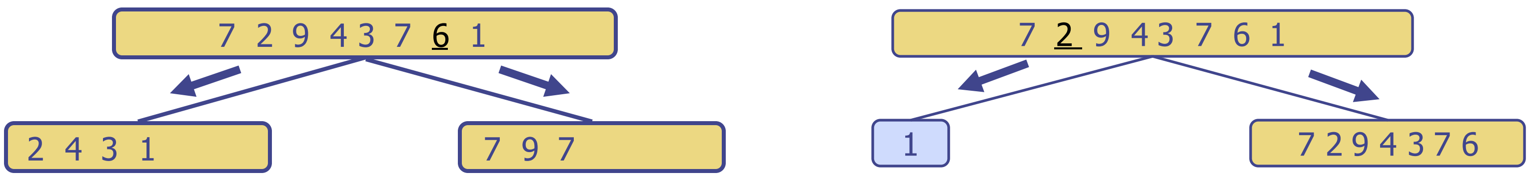
$$n + (n - 1) + \ldots + 2 + 1$$

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



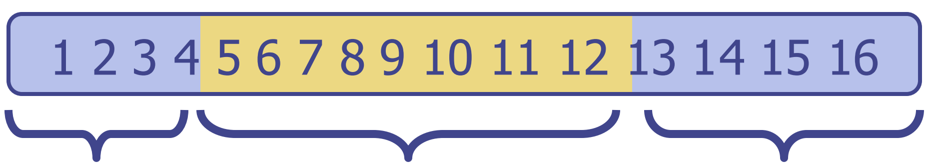
Expected Running Time

- 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 $3s/4$
  - **Bad call:** one of L and G has size greater than $3s/4$



Good call **Bad call**

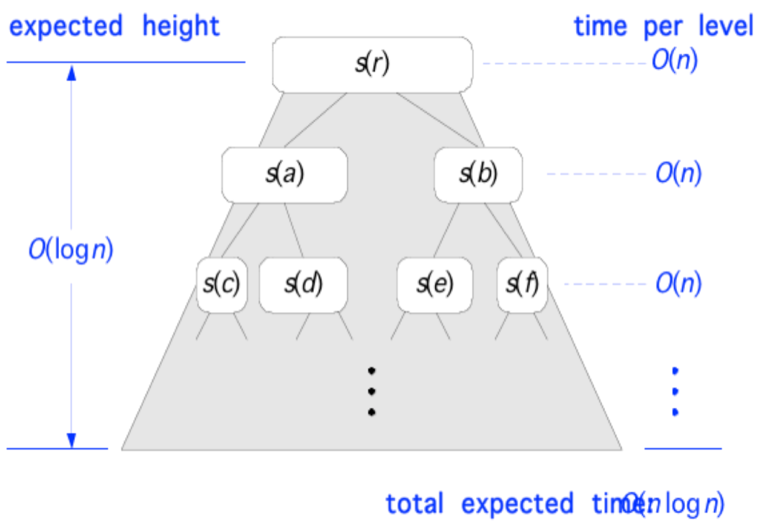
- A call is good with probability $1/2$
  - 1/2 of the possible pivots cause good calls:



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

Expected Running Time, Part 2

- Probabilistic Fact: The expected number of coin tosses required in order to get k heads is $2k$
- For a node of depth i, we expect
  - $i/2$ ancestors are good calls
  - The size of the input sequence for the current call is at most $(3/4)^{i/2}n$
    - Since each **good** call shrinks size to at most 3/4 of previous size
- Therefore, we have
  - For a node of depth $2\log_{4/3}n$, the expected input size is one
  - The expected height of the quick-sort tree is $O(\log n)$
- The amount or work done at the nodes of the same depth is $O(n)$
- Thus, the expected running time of quick-sort is $O(n \log n)$



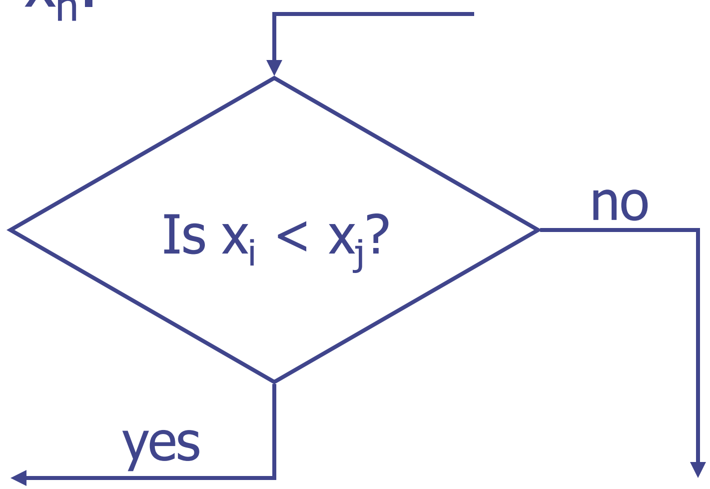
$$\left(\frac{3}{4}\right)^{\frac{i}{2}} n = 1 \quad \Rightarrow \quad i = 2\log_{\frac{4}{3}} n$$

Sorting Lower Bound

Comparison-Based Sorting

- Many sorting algorithms are comparison based.
  - They sort by making comparisons between pairs of objects
  - Examples: bubble-sort, selection-sort, insertion-sort, heap-sort, merge-sort, quick-sort, ...
- Let us therefore derive a lower bound on the running time of any algorithm that uses comparisons to sort a set S of n elements, $x_1, x_2, \ldots, x_n$.

Assume that the x_i are distinct, which is not a restriction

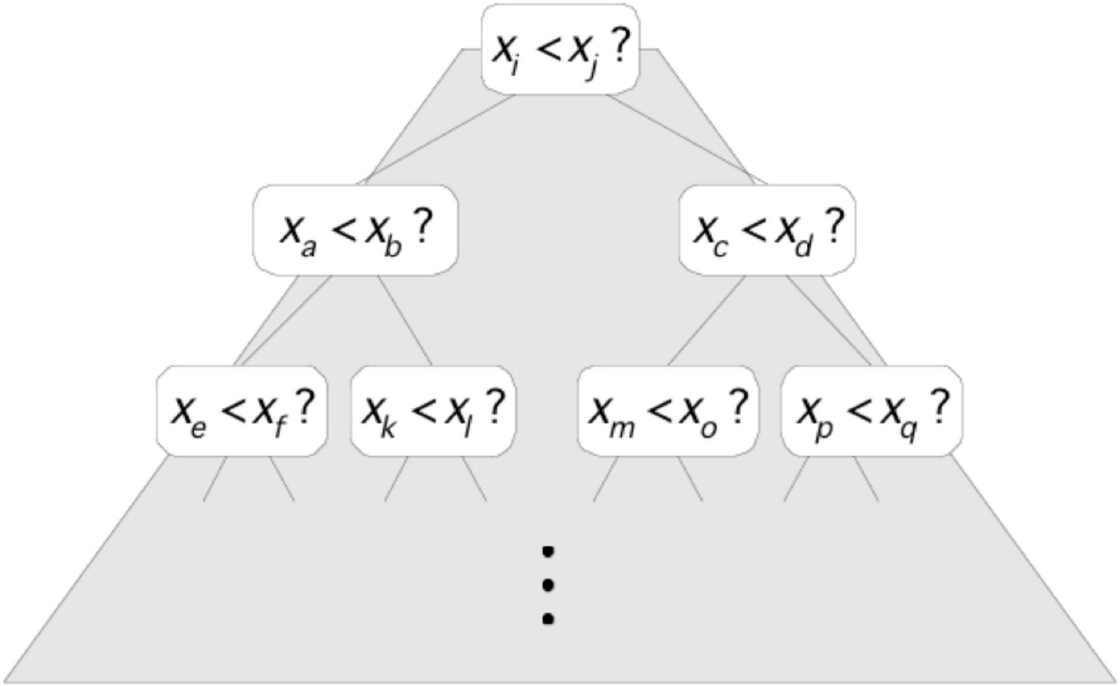


Counting Comparisons

- Let us just count comparisons then.
- First, we can map any comparison based sorting algorithm to a **decision tree** as follows:
  - Let the root node of the tree correspond to the first comparison, (is $x_i < x_j$?), that occurs in the algorithm.
  - The outcome of the comparison is either yes or no.
  - If yes we proceed to another comparison, say $x_a < x_b$? We let this comparison correspond to the left child of the root.
  - If no we proceed to the comparison $x_c < x_d$? We let this comparison correspond to the right child of the root.
  - Each of those comparisons can be either yes or no…

The Decision Tree

- **Each possible permutation of the set S** will cause the sorting algorithm to execute a sequence of comparisons, effectively traversing a path in the tree from the root to some external node

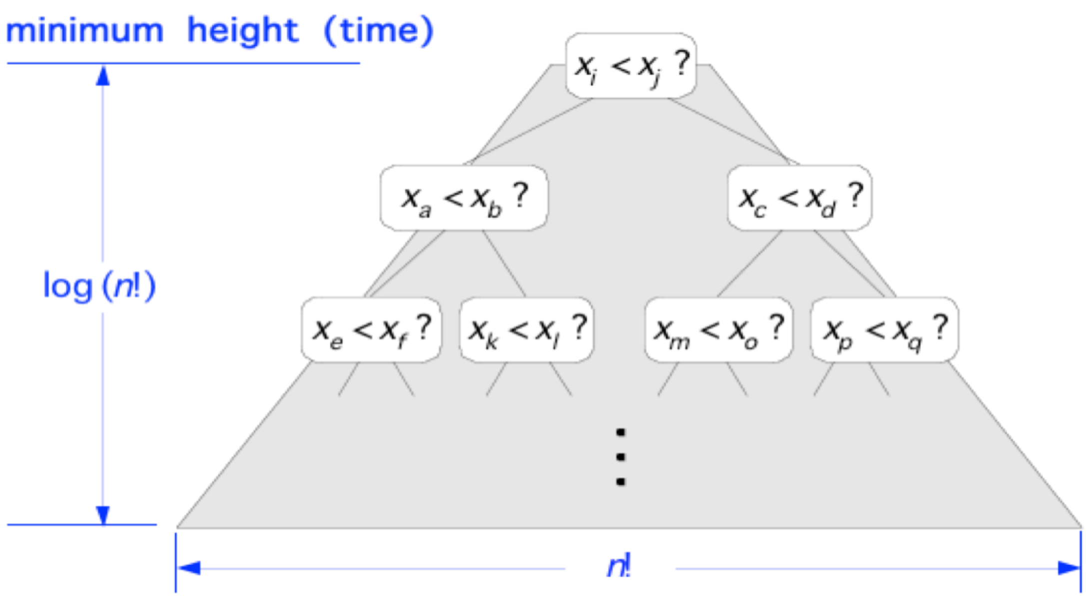


Paths Represent Permutations

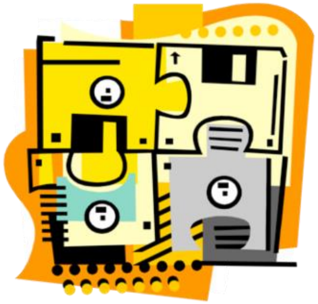
- Fact: Each external node v in the tree can represent the sequence of comparisons for exactly one permutation of S
  - If P_1 and P_2 are different permutations, then there is at least one pair x_i, x_j with x_i before x_j in P_1 and x_i after x_j in P_2
  - For both P_1 and P_2 to end up at v, this means every decision made along the way resulted in the exact same outcome.
    - We have a decision *tree*, so no cycles!
  - This cannot occur if the sorting algorithm behaves correctly, because in one permutation x_i started before x_j and in the other their order was reversed (remember, they cannot be equal)

Decision Tree Height

- The height of this decision tree is a lower bound on the running time
- Every possible input permutation must lead to a separate leaf output (by previous slide).
- There are n! permutations, so there are n! leaves.
- Since there are n!=1*2*…*n leaves, the height is at least log (n!)



The Lower Bound



- Any comparison-based sorting algorithms takes at least log (n!) time
- Therefore, any such algorithm takes time at least

$$\log(n!) \geq \log\left(\frac{n}{2}\right)^{\frac{n}{2}} = (n/2)\log(n/2).$$

  - Since there are at least n/2 terms larger than n/2 in n!
- That is, any comparison-based sorting algorithm must run no faster than O(n log n) time in the worst case.