# `Optional` Extra Material

# Stability

A *stable* sorting algorithm preserves the order of equal keys.

# Stability

A *stable* sorting algorithm preserves the order of equal keys.

**Example 1**: Some possible sorts for `[1, 2, 1', 3, 1", 2', 3']` are:

`[1, 1', 1", 2, 2', 3, 3']` and `[1', 1, 1", 2', 2, 3, 3']`

preserved the original order of the1's, 2's and 3's.

preserved the original order of 3's but not the 1's or the 2's

# Stability

A *stable* sorting algorithm preserves the order of equal keys.

**Example 1**: Some possible sorts for `[1, 2, 1', 3, 1", 2', 3']` are:

`[1, 1', 1", 2, 2', 3, 3']` and `[1', 1, 1", 2', 2, 3, 3']`

**Example 2**: Sort files by *name* then by *type*.

sorted by name

| Name | Type |
|---|---|
| cat.jpg | JPEG |
| dog.jpg | JPEG |
| exam.doc | DOC |
| grades.pdf | PDF |
| lizard.jpg | JPEG |
| minutes.doc | DOC |
| quiz.doc | DOC |
| sorting.pdf | PDF |
| spendings.pdf | PDF |
| statement.doc | DOC |

sort by type →

| Name | Type |
|---|---|
| quiz.doc | DOC |
| exam.doc | DOC |
| statement.doc | DOC |
| minutes.doc | DOC |
| dog.jpg | JPEG |
| cat.jpg | JPEG |
| lizard.jpg | JPEG |
| spendings.pdf | PDF |
| grades.pdf | PDF |
| sorting.pdf | PDF |

# Stability

A *stable* sorting algorithm preserves the order of equal keys.

**Example 1**: Some possible sorts for `[1, 2, 1', 3, 1", 2', 3']` are:

`[1, 1', 1", 2, 2', 3, 3']` and `[1', 1, 1", 2', 2, 3, 3']`

**Example 2**: Sort files by *name* then by *type*.

sorted by name

| Name | Type |
|---|---|
| cat.jpg | JPEG |
| dog.jpg | JPEG |
| exam.doc | DOC |
| grades.pdf | PDF |
| lizard.jpg | JPEG |
| minutes.doc | DOC |
| quiz.doc | DOC |
| sorting.pdf | PDF |
| spendings.pdf | PDF |
| statement.doc | DOC |

sort by type →

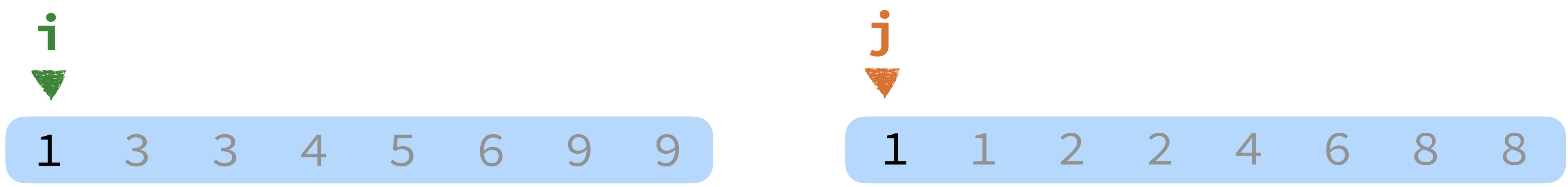
| Name | Type |
|---|---|
| quiz.doc | DOC |
| exam.doc | DOC |
| statement.doc | DOC |
| minutes.doc | DOC |
| dog.jpg | JPEG |
| cat.jpg | JPEG |
| lizard.jpg | JPEG |
| spendings.pdf | PDF |
| grades.pdf | PDF |
| sorting.pdf | PDF |

original order not preserved! (not stable!)

# Stability

A *stable* sorting algorithm preserves the order of equal keys.

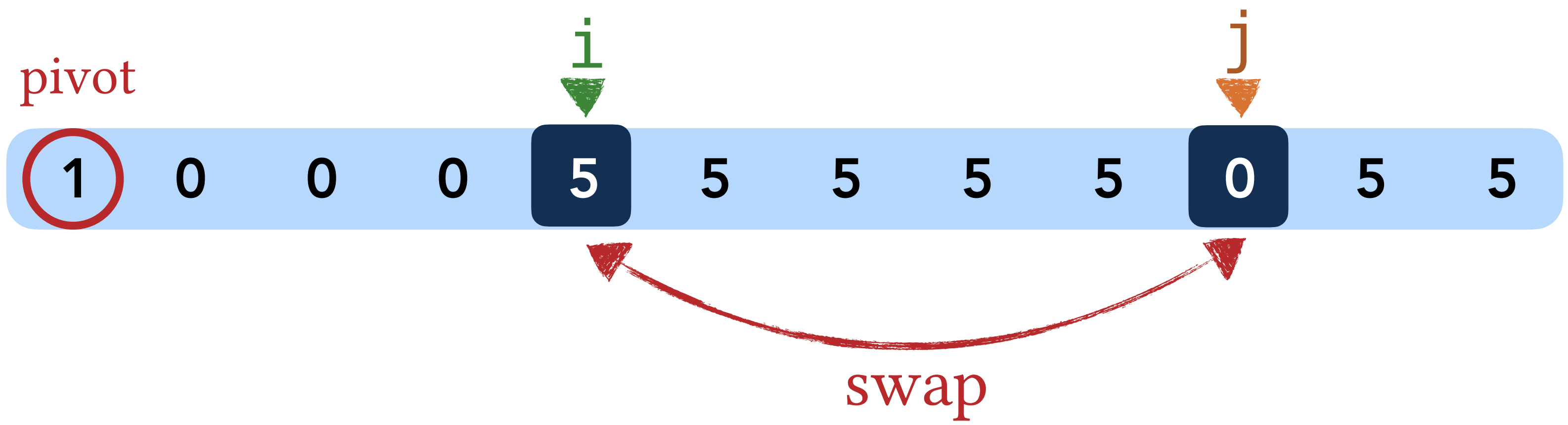
- Merge Sort is stable.
  In the merge operation: copy from the left subarray if the elements are equal.



# Stability

A *stable* sorting algorithm preserves the order of equal keys.

- Merge Sort is stable.
  In the merge operation: copy from the left subarray if the elements are equal.
- Quicksort is *not* stable.
  Partitioning does not preserve the order of equal elements.

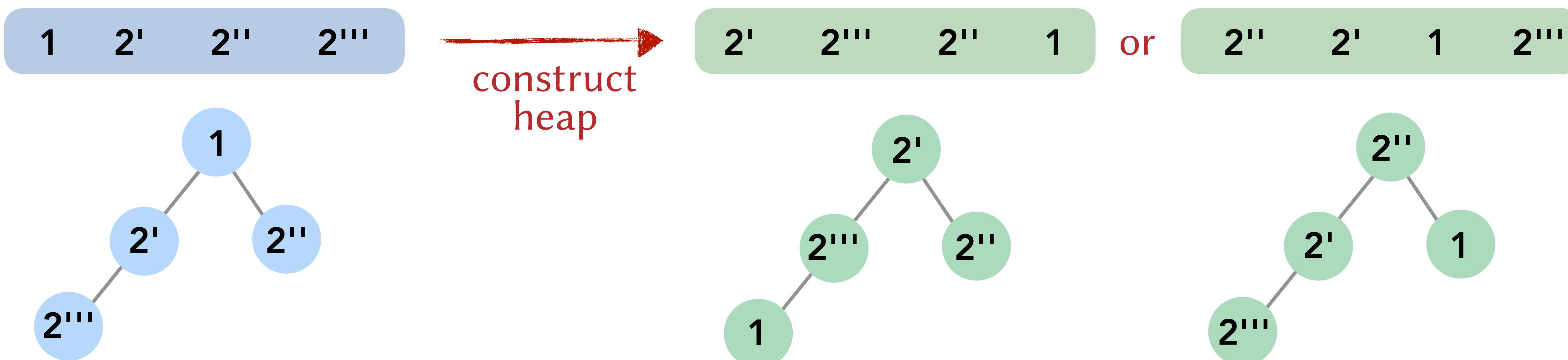



the first 5 is not the first 5 anymore!

# Stability

A *stable* sorting algorithm preserves the order of equal keys.

- Merge Sort is stable.
  In the merge operation: copy from the left subarray if the elements are equal.
- Quicksort is *not* stable.
  Partitioning does not preserve the order of equal elements.
- Heapsort is *not* stable.
  Sinking does not preserve the order of equal elements



# Sorting Lower Bound

Proposition. Any comparison-based sorting algorithm performs at least $\sim n \log_2 n$ compares in the worst case.

# Sorting Lower Bound

Proposition. Any comparison-based sorting algorithm performs at least $\sim n \log_2 n$ compares in the worst case.

are there sorting algorithms that
are not comparison-based?
**Yes!** (e.g. Radix Sort)

# Sorting Lower Bound

Proposition. Any comparison-based sorting algorithm performs at least $\sim n \log_2 n$ compares in the worst case.

proposition holds in the worst case only. E.g. Insertion sort does $\Theta(n)$ comparisons in the best case.

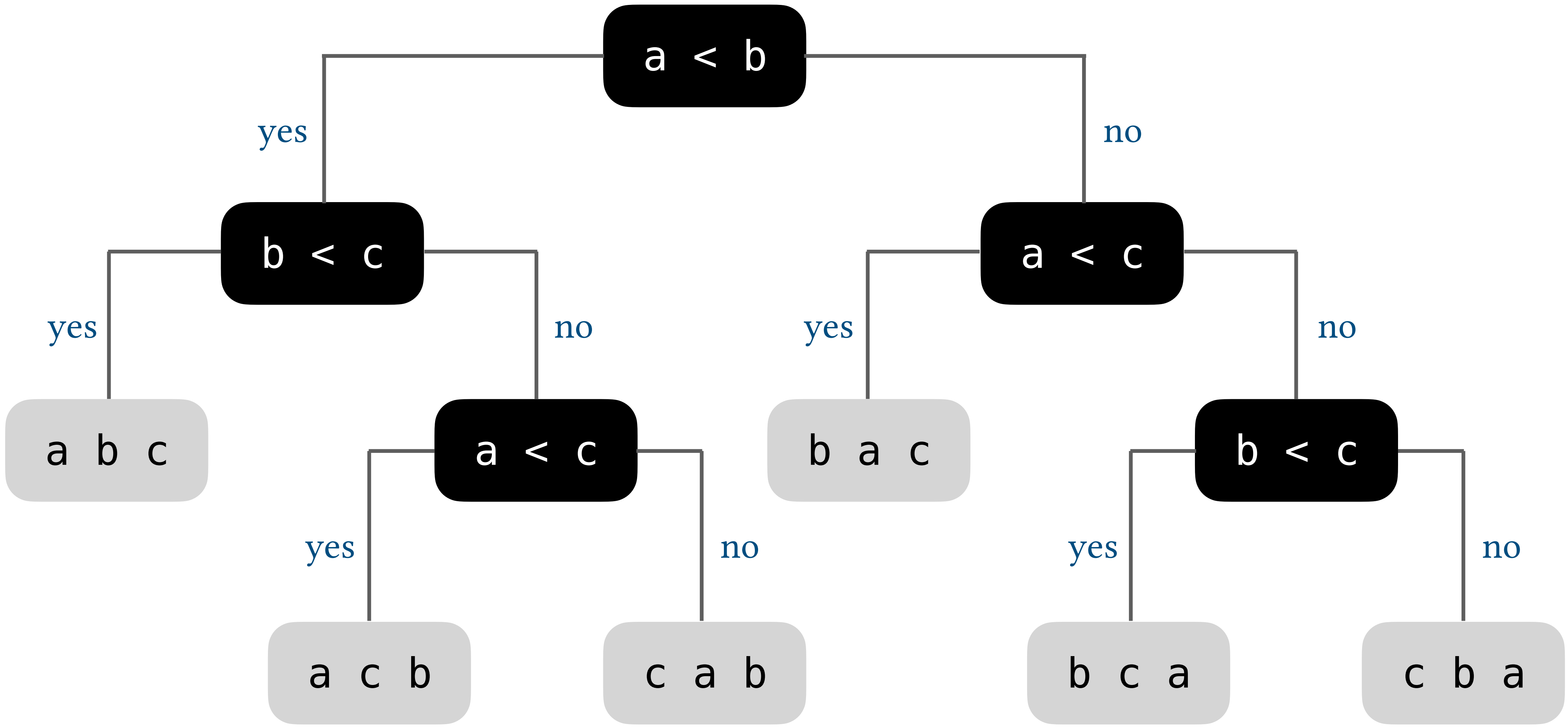
# Sorting Lower Bound

Proposition. Any comparison-based sorting algorithm performs at least $\sim n \log_2 n$ compares in the worst case.

Put another way. For any comparison-based sorting algorithm, there must be at least one sequence of elements for which the sorting algorithm needs $\sim n \log_2 n$ comparisons to sort.

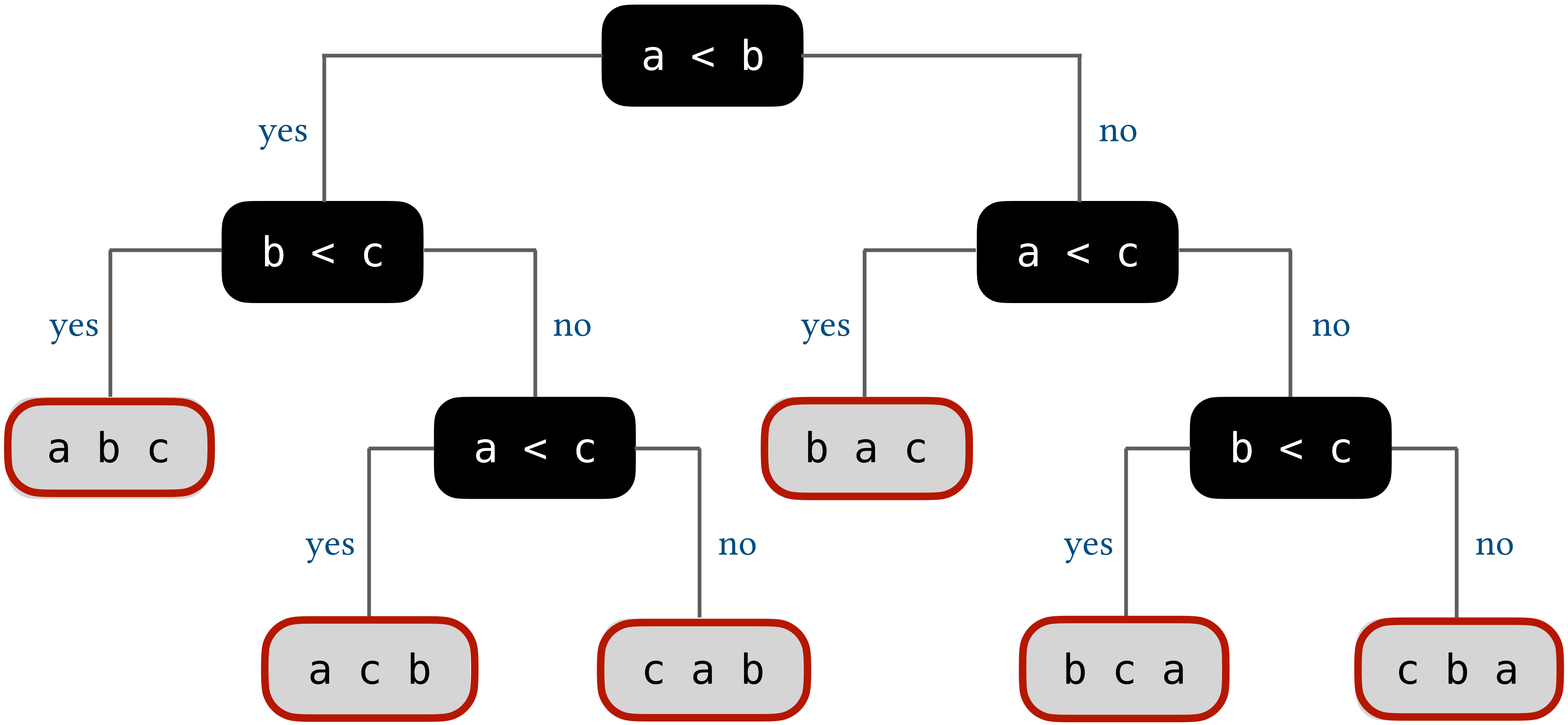
# Sorting Lower Bound

A comparison tree for three distinct keys (a, b and c)



# Sorting Lower Bound

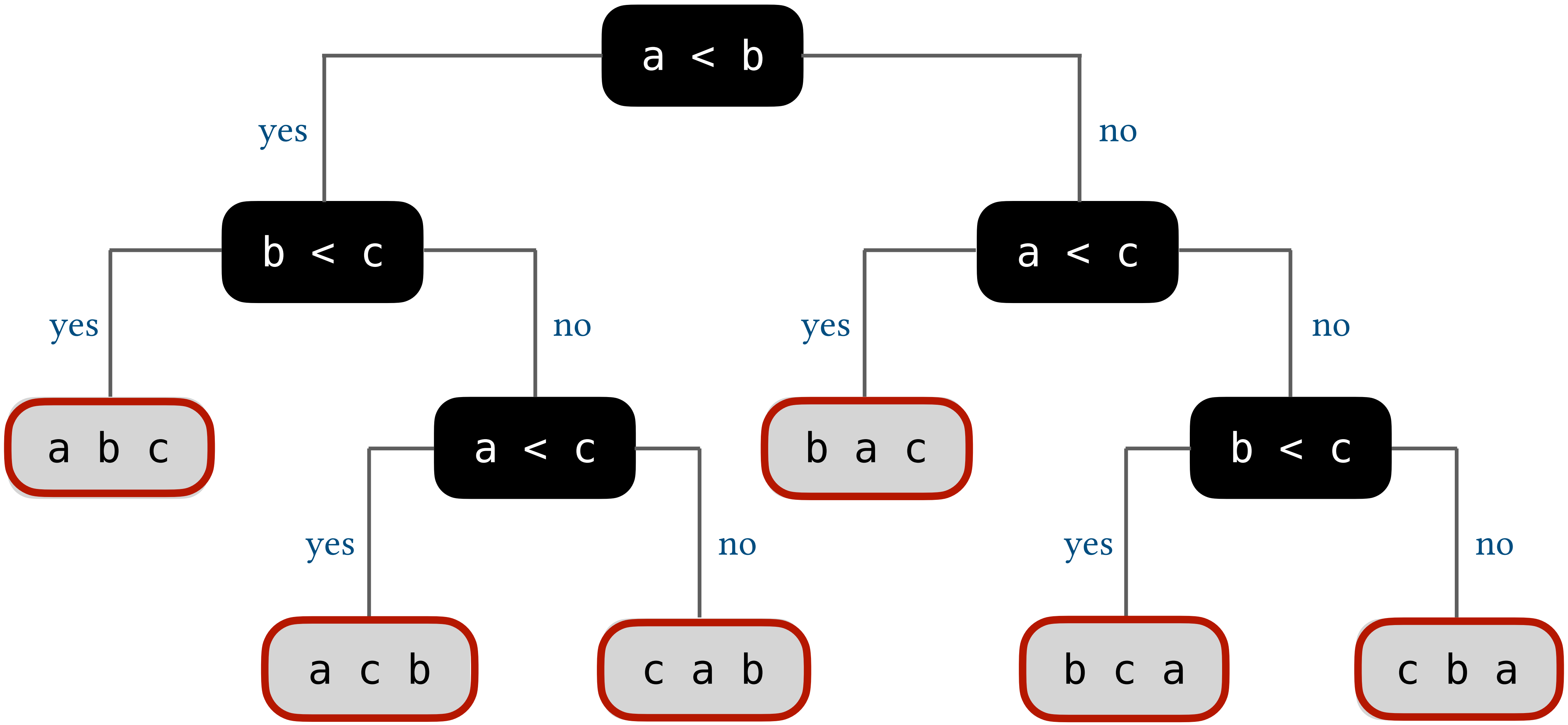
A comparison tree for three distinct keys (a, b and c)




There are $n!$ unique orderings making $n!$ leaves

# Sorting Lower Bound

A comparison tree for three distinct keys (a, b and c)

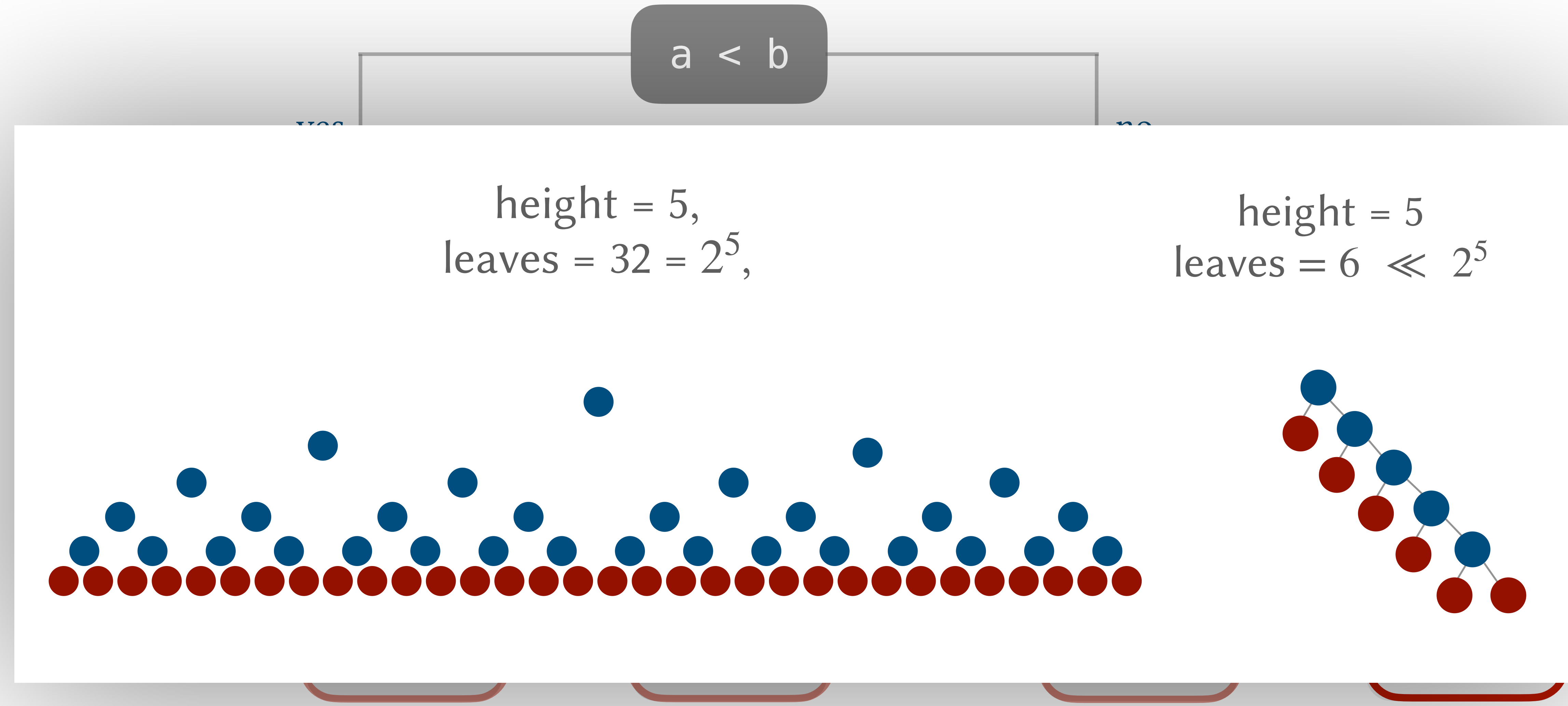



There are $n!$ unique orderings making $n!$ leaves

$$\# \text{ of leaves} \leq 2^{\text{height}}$$

# Sorting Lower Bound

A comparison tree for three distinct keys (a, b and c)

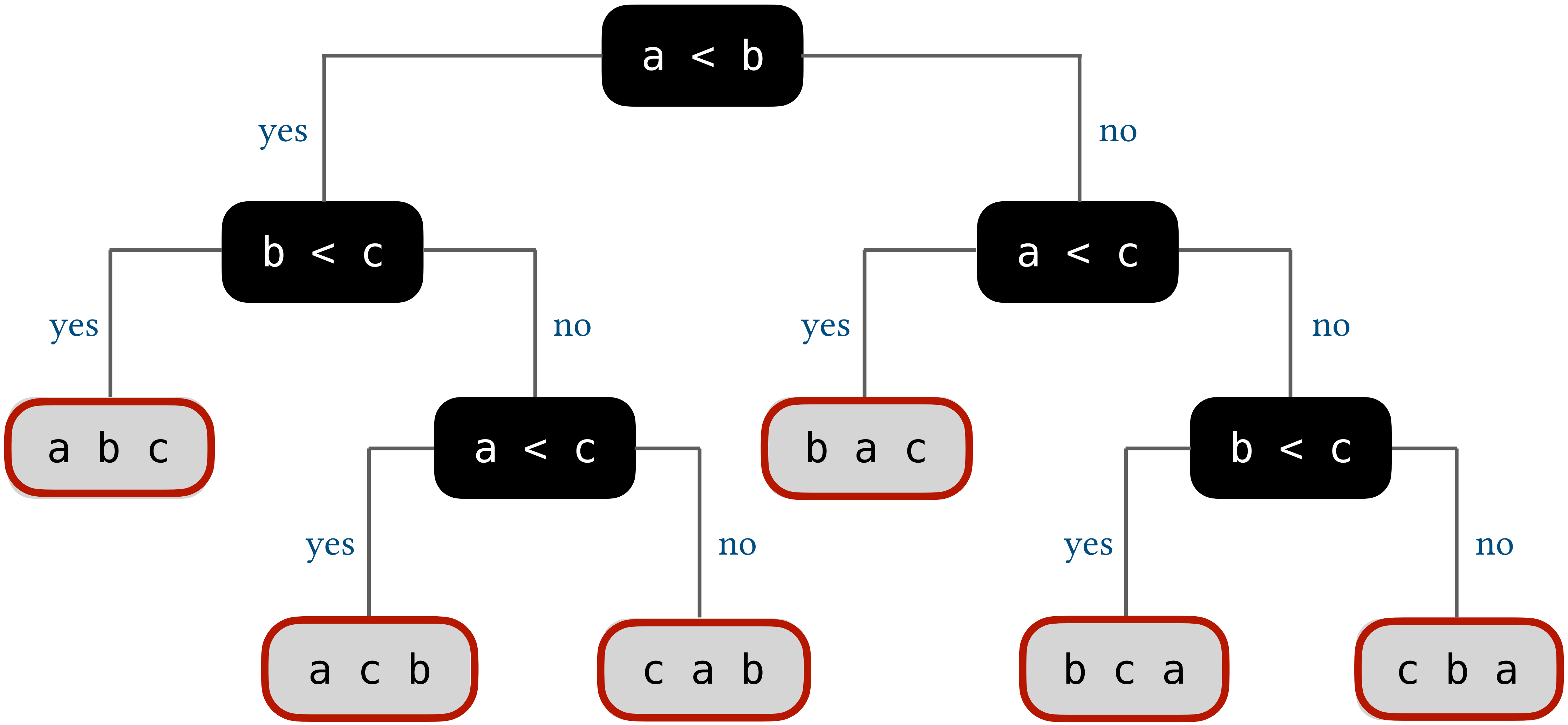



There are $n!$ unique orderings making $n!$ leaves

# of leaves $\leq$ $2^{\text{height}}$

# Sorting Lower Bound

A comparison tree for three distinct keys (a, b and c)

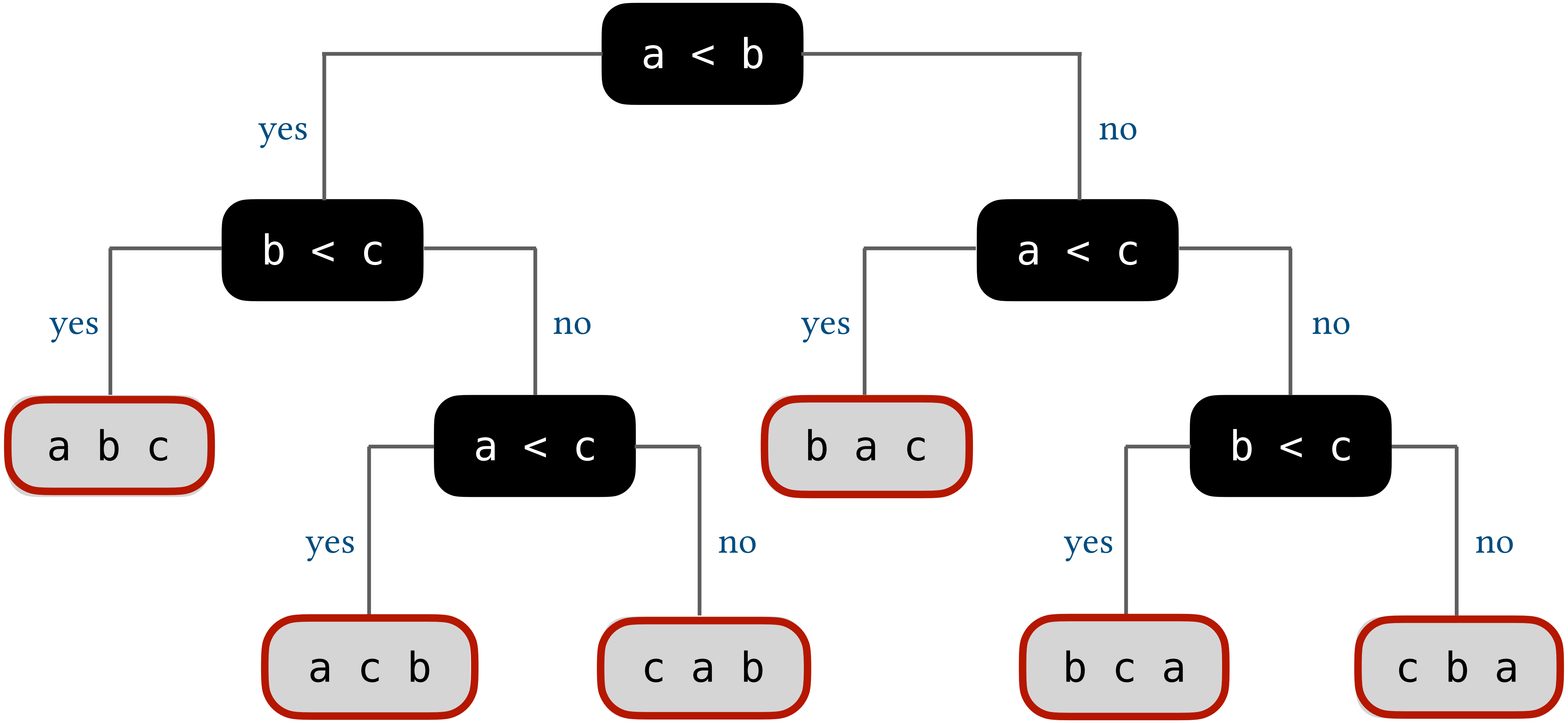



There are $n!$ unique orderings making $n!$ leaves

$$\#\text{ of leaves} \leq 2^{\text{height}}$$
$$n! \leq 2^{\text{height}}$$

# Sorting Lower Bound

A comparison tree for three distinct keys (a, b and c)

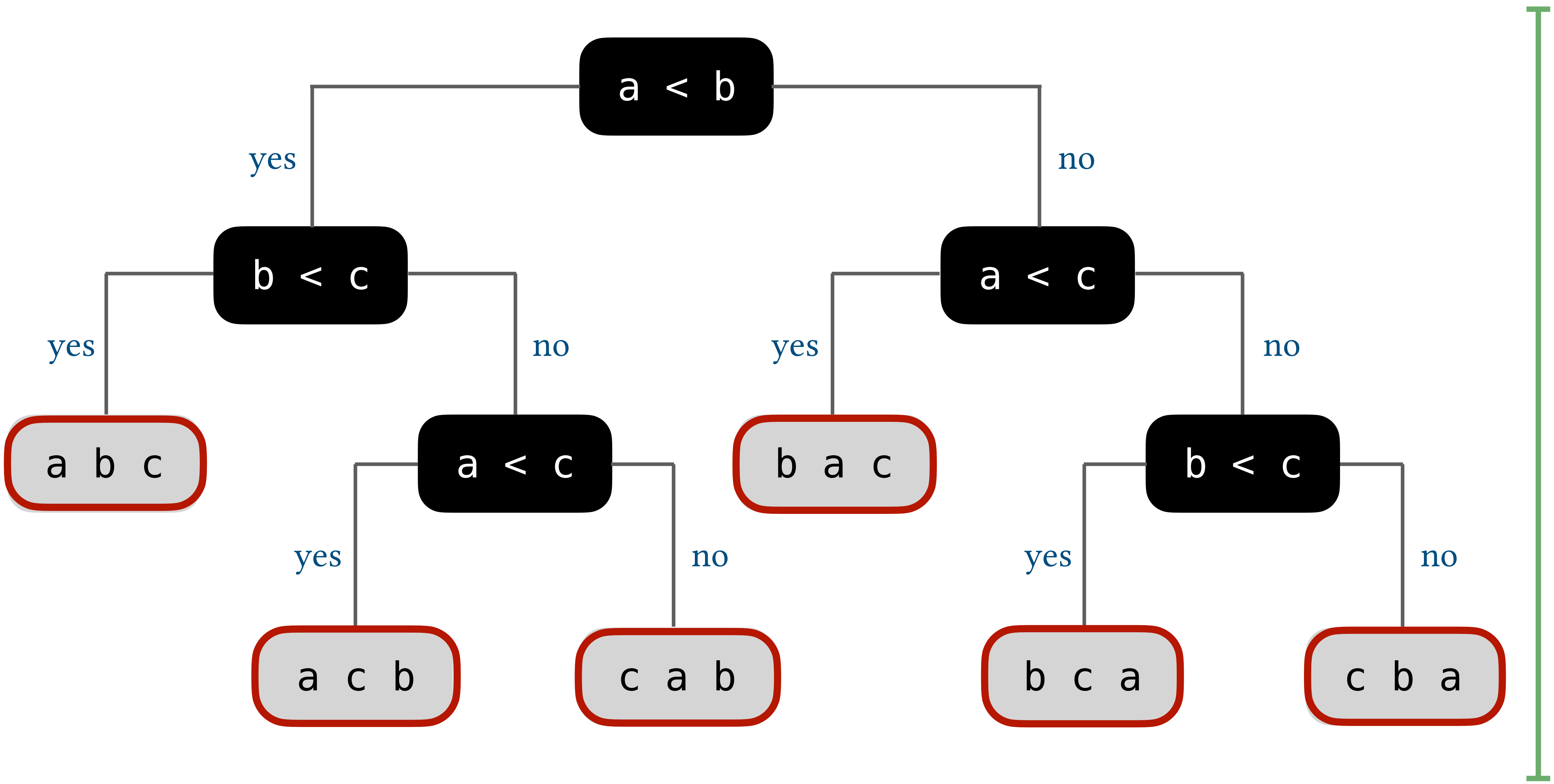



There are $n!$ unique orderings making $n!$ leaves

$$
\begin{aligned}
\#\text{ of leaves} &\leq 2^{\text{height}} \\
n! &\leq 2^{\text{height}} \\
\log(n!) &\leq \log(2^{\text{height}})
\end{aligned}
$$

# Sorting Lower Bound

A comparison tree for three distinct keys (a, b and c)




$h \geq \log_2(n!)$

There are $n!$ unique orderings making $n!$ leaves

$$\begin{aligned} \text{\# of leaves} &\leq 2^{\text{height}} \\ n! &\leq 2^{\text{height}} \\ \log(n!) &\leq \log(2^{\text{height}}) \\ \sim n\log(n) &\leq \text{height} \end{aligned}$$

← height represents # of comparisons

# Sorting Lower Bound

Proposition. Any comparison-based sorting algorithm performs at least $\sim n \log n$ compares in the worst case.

**Proof Sketch.**

- Assume the array consists of $n$ distinct values $a_1$ through $a_n$ .
- There are $n!$ unique orderings for this array.
  (any sorting algorithm must be able to distinguish between these $n!$ permutations).
- Consider a binary decision tree, where each node is labeled with a comparison between two elements ($a_i < a_j$) and each leaf is a possible ordering for the array.
  (path from the root to a leaf represents a run of a sorting algorithm).
- The tree has $n!$ leaves.
- The height of a binary tree with $n!$ leaves is $\geq \log_2(n!)$.
  (the height of the tree is $\log_2(n!)$ if it is a complete tree and possibly more if it is not).
- If the longest path in the tree is $\geq \log_2(n!)$ then there must always be a sequence of input that requires $\log_2(n!)$ comparisons to be sorted.
  (the height of a binary tree is the length of the longest path from the root to a leaf).