Comparison Sort

A **Comparison Sort** is a sorting algorithm where the final order is determined only by comparisons between the input elements.

- In **Insertion Sort**, the proper position to insert the current element is found by comparing the current element with the elements in the sorted sub-array.
- In **Heap Sort**, the Heapify procedure determines where to place items based on their comparisons with adjacent items (parent-child) in the tree.
- In **Merge Sort**, the merge procedure chooses an item from one of two arrays after comparing the top items from both arrays.
- In **Quicksort**, the Partition procedure compares each item of the subarray, one by one, to the pivot element to determine whether or not to swap it with another element.

Summary for Comparison Sort Algorithms

Sorting Methods	Worst Case	Best Case	Average Case	Applications
InsertionSort	n^2	n	n^2	Very fast when $n < 50$
MergeSort	$nlgn$	$nlgn$	$nlgn$	Need extra space; good for linked lists.
HeapSort	$nlgn$	$nlgn$	$nlgn$	Good for real-time appl.
QuickSort	n^2	$nlgn$	$nlgn$	Practical and fast

Decision Tree Model

- Each comparison sort algorithm can be viewed abstractly in terms of a **decision tree**.

- It is a rooted binary tree where internal nodes represent comparisons between two keys and leaves represent sorted outputs.

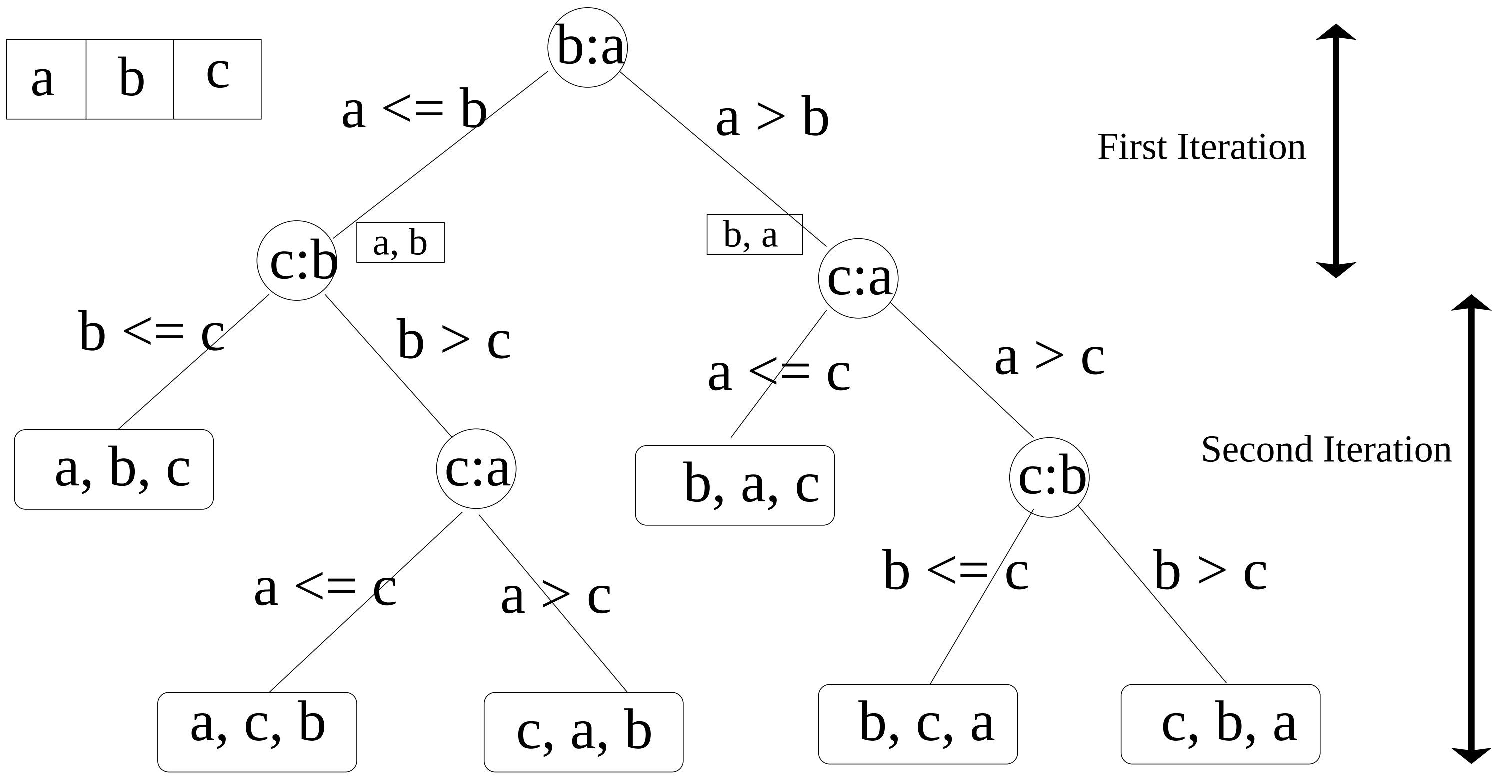
A comparison Sort algorithm + an input size n

$\longleftrightarrow$ a decision tree.

Insertion Sort Algorithm

```
INSERTION-SORT(A)
    for j:=2 to length of A do
        key := A[j]
        /* put A[j] into A[1..j-1] */
        i := j -1
        while ( i > 0 AND A[i] > key)
            A[i+1] := A[i]
            i:=i - 1
        A[i+1] := key
```

The Decision Tree Corresponding to Insertion Sort (n = 3)



Another Example: Bubble Sort

Bubble elements to their proper place in the array by comparing elements i and $i+1$, and swapping if $A[i] > A[i+1]$.

- last position has the largest element (loop invariant).

- then **bubble** every element except the last one towards its correct position.

- then repeat until done or until the end of quarter.

- whichever comes first ...

Illustration of Bubble Sort

Input: *4 2 5 3*

4 2 5 3

2 4 5 3

2 4 5 3

2 4 3 **5**

2 4 3 **5**

2 4 3 **5**

2 3 **4 5**

2 3 **4 5**

2 **3 4 5**

2 3 4 5

Another input: *3 2 1*

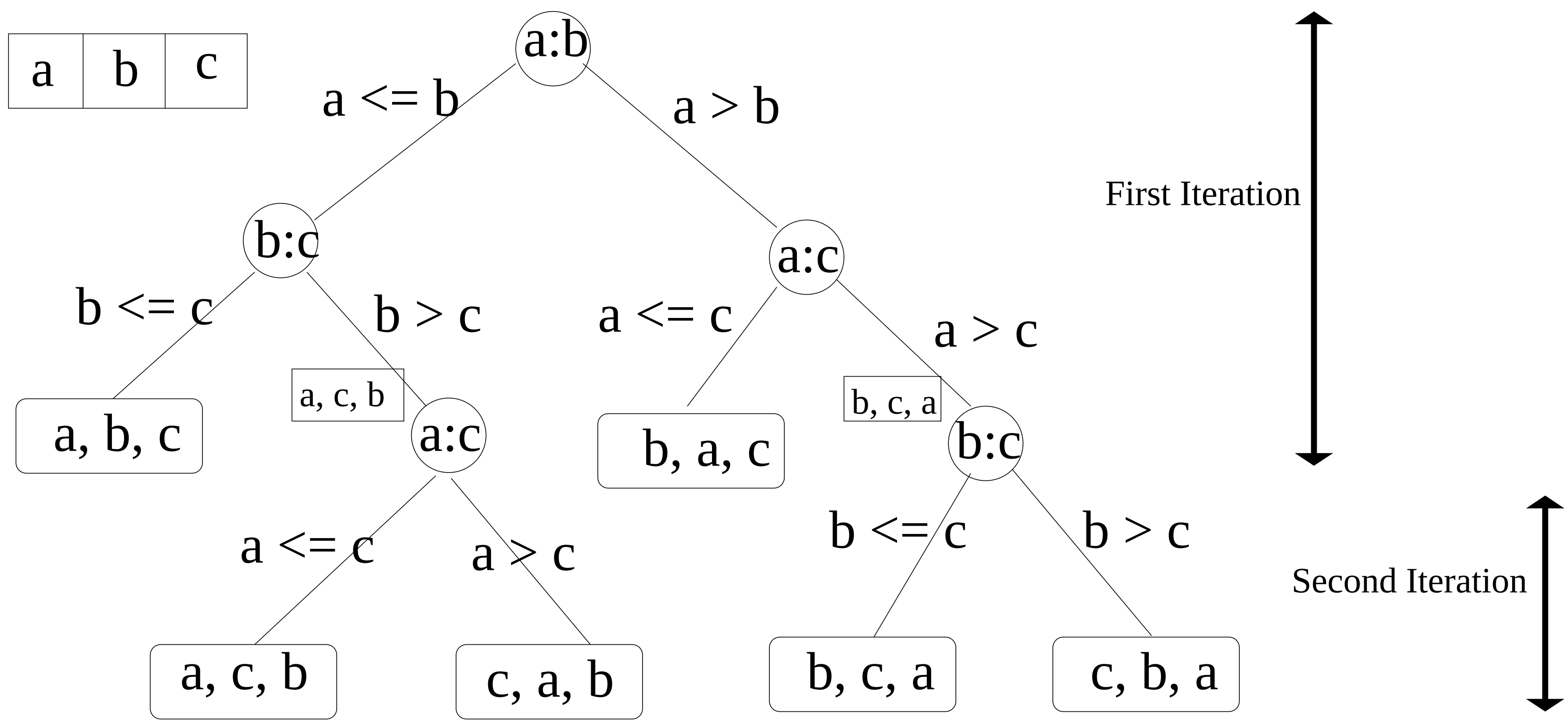
3 2 1

2 3 1

2 1 **3**

1 2 3

The Decision Tree Corresponding to Bubble Sort (n = 3)



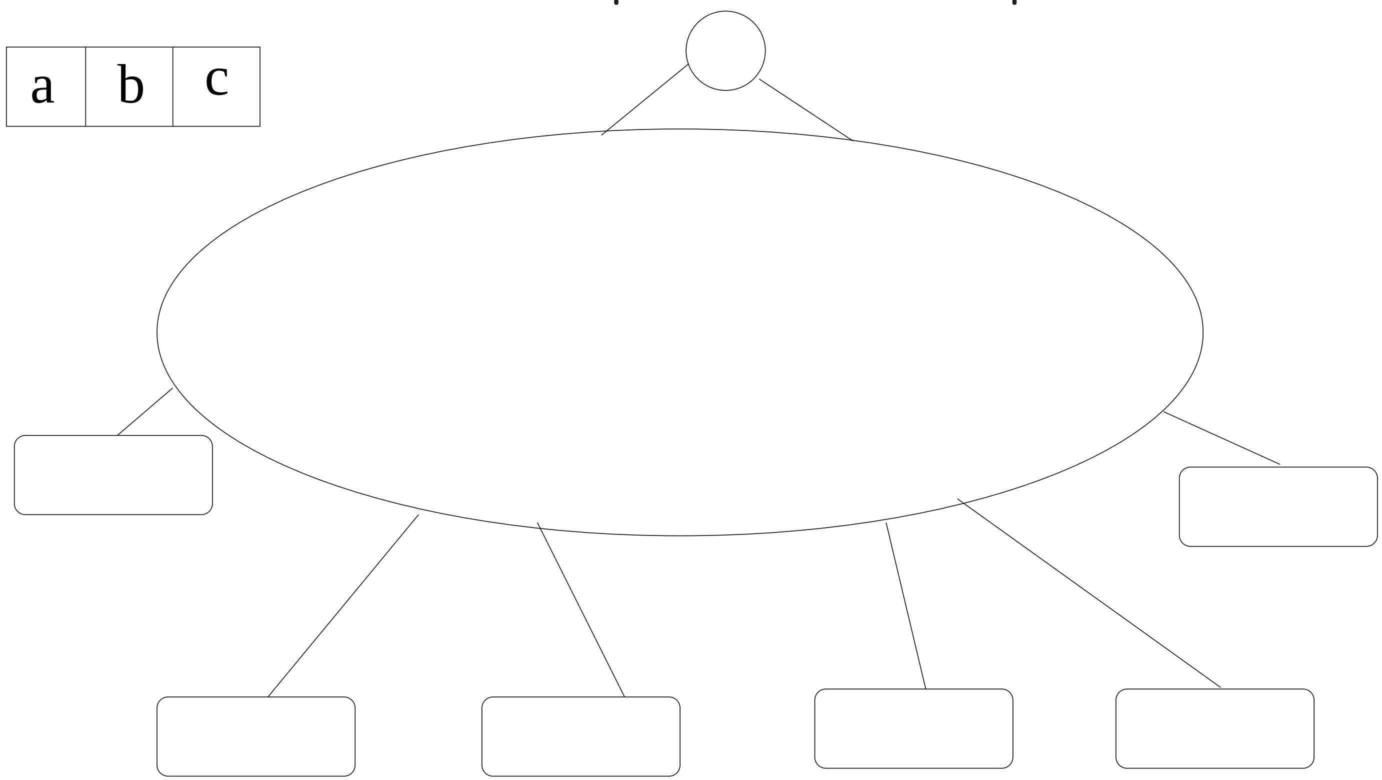
Lower Bound for Comparison-based Sorting

The decision trees of comparison-based sorting algorithms:

- Each internal node contains a comparison.
- Each leaf contains a permutation. All the leaf nodes produce a correctly sorted sequence.
- Algorithm execution = a path from the root to a leaf.
- Worst-case number of comparisons = height of tree.
- **Idea**: If we find a lower bound on the height of the decision tree, we will have a lower bound on the running time of any comparison-based sorting algorithm.

A Necessary Condition for Any Correct Comparison Sorting Algorithm

Given an input size n, there are $n!$ possible orderings of the elements, so the corresponding decision tree needs to have at least **n!** leaf nodes to produce these permutations.



Lower Bound on the Height of a Decision Tree

Theorem: Any decision tree that sorts n elements has height $\Omega(nlgn)$.

- Suppose the decision tree has height h.

- A binary tree of height h has at most 2^h leaves.

- The decision tree must have at least $n!$ leaves, hence:

$$2^h \geq l \geq n! \Rightarrow h \geq lg(n!).$$

- Claim: $lg(n!) = \Theta(nlgn)$ *(see hw 1)*.

- Therefore $h \geq \Theta(nlgn) \Rightarrow \mathrm{h} = \Omega(nlgn)$.