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

<table>
<thead>
<tr>
<th>Sorting Methods</th>
<th>Worst Case</th>
<th>Best Case</th>
<th>Average Case</th>
<th>Applications</th>
</tr>
</thead>
<tbody>
<tr>
<td>InsertionSort</td>
<td>$n^2$</td>
<td>$n$</td>
<td>$n^2$</td>
<td>Very fast when $n &lt; 50$</td>
</tr>
<tr>
<td>MergeSort</td>
<td>$n \log n$</td>
<td>$n \log n$</td>
<td>$n \log n$</td>
<td>Need extra space; good for linked lists.</td>
</tr>
<tr>
<td>HeapSort</td>
<td>$n \log n$</td>
<td>$n \log n$</td>
<td>$n \log n$</td>
<td>Good for real-time appl.</td>
</tr>
<tr>
<td>QuickSort</td>
<td>$n^2$</td>
<td>$n \log n$</td>
<td>$n \log n$</td>
<td>Practical and fast</td>
</tr>
</tbody>
</table>
- 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 $n$ ↔ a decision tree.
**Insertion Sort Algorithm**

**INSERTION-SORT(A)**

1. for j:=2 to length of A do
2.   key := A[j]
4.   i := j - 1
5.   while ( i > 0 AND A[i] > key)
7.     i:=i - 1
8.   A[i+1] := key
The Decision Tree Corresponding to Insertion Sort (n = 3)

The decision tree for insertion sort with 3 elements (a, b, c) is shown. The tree branches out based on the comparison of elements:

- **First Iteration:**
  - **First Comparison:** a <= b
    - If a <= b, the next comparison is a > c.
    - If a > b, the next comparison is b <= c.
- **Second Iteration:**
  - **Comparison:** a <= c
    - If a <= c, the next comparison is a > c.
    - If a > c, the next comparison is b <= c.

The tree structure reflects the sorting process, with each comparison leading to a decision on the next element to be compared.
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

Another input: 3 2 1
The Decision Tree Corresponding to Bubble Sort (n = 3)

First Iteration

Second Iteration
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.

![Decision Tree Diagram]
Theorem: Any decision tree that sorts $n$ elements has height $\Omega(n \log n)$.

- 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:
  
  $$2^h \geq l \geq n! \Rightarrow h \geq \log(n!).$$

- Claim: $\log(n!) = \Theta(n \log n)$ (see hw 1).
- Therefore $h \geq \Theta(n \log n) \Rightarrow h = \Omega(n \log n)$. 