POLITECNICO
DI TORINO

Dipartimento
di Automatica e Informatica

# Sorting Algorithms

Paolo Camurati

# Problem definition

Sorting:

- Input: symbols $<a_1, a_2, \ldots, a_n>$ belonging to a set having an order relation $\leq$
- Output: permutation $<a'_1, a'_2, \ldots, a'_n>$ of the input for which the order relation holds $a'_1 \leq a'_2 \leq \ldots \leq a'_n$.

# Order relation $\leq$

Binary relation between items of a set A satisfying the following properties:

- reflexivity $\forall\, x \in A\ x \leq x$
- antisymmetry $\forall\, x, y \in A\ \ x \leq y \wedge y \leq x \Rightarrow x = y$
- transitivity $\forall\, x, y, z \in A\ \ x \leq y \wedge y \leq z \Rightarrow x \leq z$

A is a partially ordered set (poset). If relation $\leq$ holds $\forall\, x, y \in A$, A is totally ordered set.

Examples of order relations $\leq$ :

- (total) relation $\leq$ on natural, relative, rational and real numbers (sets N, Z, Q, R)
- (partial) relation: divisibility on natural numbers, excluding 0.

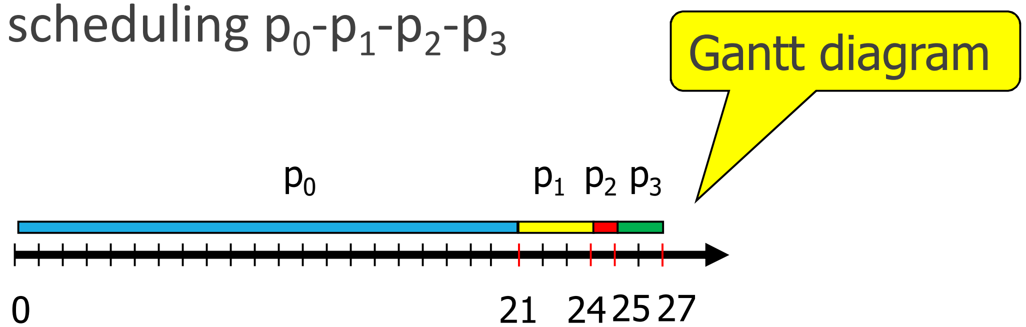
# On the importance of sorting

- 30% of CPU time is spent on sorting data
- CPU scheduling: how to select among the processes ready for execution the next one to be run on the CPU.
  - Simple solution: queue, thus *First-come First-Served* policy: the first request that arrives is the first one to be served
  - problem: *minimize average waiting time*: it turns into an optimization problem.

Example: processes $p_i$ with duration:

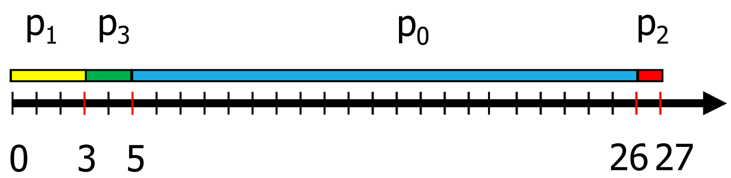
$p_0$ 21, $p_1$ 3, $p_2$ 1, $p_3$ 2

- scheduling $p_0$-$p_1$-$p_2$-$p_3$



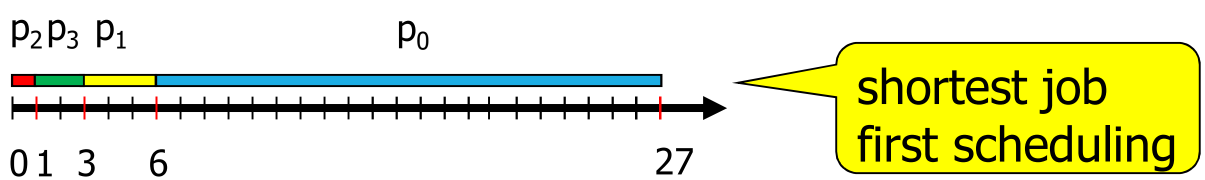

average waiting time (0+21+24 +25)/4 =17,5

- scheduling $p_1$-$p_3$-$p_0$-$p_2$




average waiting time (0+3+5 +26)/4 =8,5

- scheduling (sorted by increasing duration) $p_2$-$p_3$-$p_1$-$p_0$




average waiting time (0+1+3 +6)/4 =2,5

# Applications of sorting

- Sorting a list of names
- Organizing an MP3 library
- Displaying  Google PageRank results
- …

Trivial applications

- find the median
- binary search in a database
- find duplicates in a mailing list
- …

Simple problems if data are sorted

- data compression
- computer graphics (eg. convex hull)
- computational biology
- …

**Non trivial applications**

# Classification: internal/external sorting

- Internal sorting  only
  - data are in main memory
  - direct access to items
- External sorting
  - data are on mass memory
  - sequential access to items

# Classification: in place / stable

- In place sorting
  n data in array + constant number of auxiliary memory locations
- Stable sorting
  for data with duplicated keys the relative ordering is unchanged: the output relative ordering is the same as the relative input ordering

# Example

Struct with 2 keys: name (the first letter is the key) and group (the key is an integer)

First sorting by first letter:

| | |
|---|---|
| Andrea | 3 |
| Barbara | 4 |
| Chiara | 3 |
| Fabio | 3 |
| Franco | 1 |
| Giada | 4 |
| Lucia | 3 |
| Roberto | 2 |

Second sorting by group:

keeping the original ordering
stable --> yes / non stable --> no (different order)

NON stable algorithm

| | |
|---|---|
| Franco | 1 |
| Roberto | 2 |
| Chiara | 3 |
| Fabio | 3 |
| Andrea | 3 |
| Lucia | 3 |
| Giada | 4 |
| Barbara | 4 |

Stable algorithm

| | |
|---|---|
| Franco | 1 |
| Roberto | 2 |
| Andrea | 3 |
| Chiara | 3 |
| Fabio | 3 |
| Lucia | 3 |
| Barbara | 4 |
| Giada | 4 |

# Classification: complexity

- $O(n^2)$:
  - simple, iterative, based on comparison
  - Insertion sort, Selection sort, Exchange/Bubble sort
- $O(n^x)$ with $x \leq 2$
  - Evolution of the simple ones, iterative, based on comparison
  - Shell sort, $O(n^2)$, $O(n^{3/2})$, $O(n^{4/3})$ depending on the sequence
- $O(n \log n)$:
  - more complex, recursive, based on comparison. Will be dealt with in second year Course
  - Merge sort, Quick sort, Heap sort
    iterative
- $O(n)$:
  - applicable only with restrictions on data, based on computation
- Counting sort, Radix sort, Bin/Bucket sort

A mode detailled analysis is possible, distinguishing between

- comparison and
- swap operations.

When data are large, moving chunks of memory (not just pointers) may be expensive.

Asymptotic complexity however doesn’t change.

# Graphs: Paths

sequence of nodes

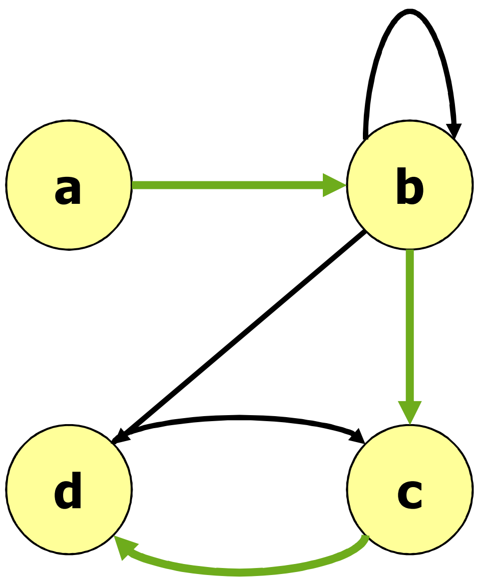
In a graph $G = (V, E)$

Path $p$: $u \rightarrow_p u'$:

$\exists(v_0,v_1,v_2,\dots,v_k) \mid u=v_0,\ u'=v_k,\ \forall i = 1,2,\dots,k\ (v_{i-1},v_i) \in E$

- $k$ = length of the path
- $u'$ is reachable from $u \Leftrightarrow \exists\, p$: $u \rightarrow_p u'$. In an undirected graph it is also true that $u$ is reachable from $u' \Leftrightarrow \exists\, p$: $u' \rightarrow_p u$
- simple path $p \Leftrightarrow (v_0,v_1,v_2,\dots,v_k) \in p$ distinct

# Example




G = (V, E)
p: a $\rightarrow_p$ d : (a, b), (b, c), (c, d)
k = 3
d is reachable from a (not necessarily vice-versa)
p is simple.

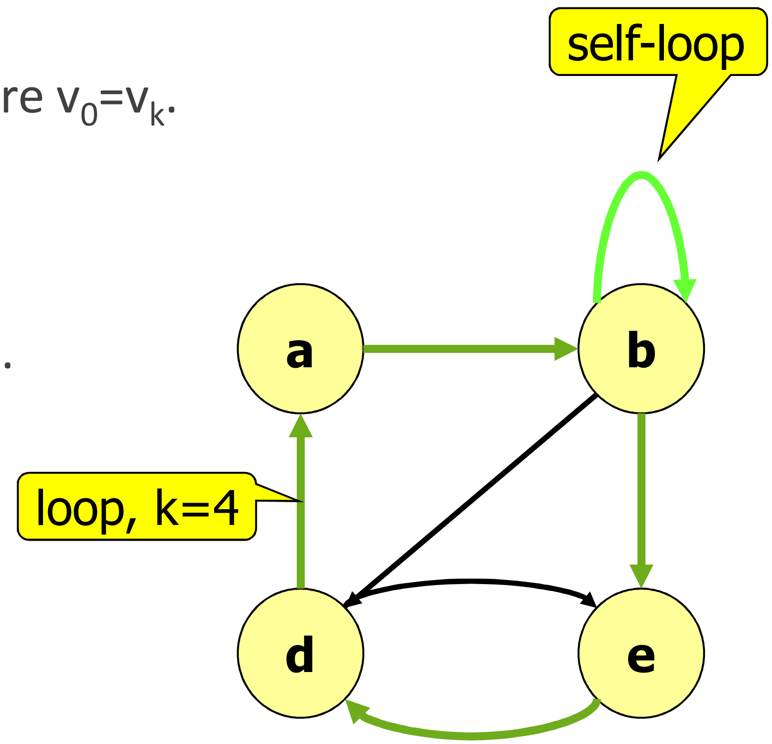
# Graphs: Loops

Loop = path where $v_0=v_k$.

Simple loop = simple path where $v_0=v_k$.

Self-loop = unit-length loop.

A graph without loops is acyclic.



# Undirected Graphs: Connection

Connected undirected graph G = (V, E):

alwyas find a path between each pair of vertices

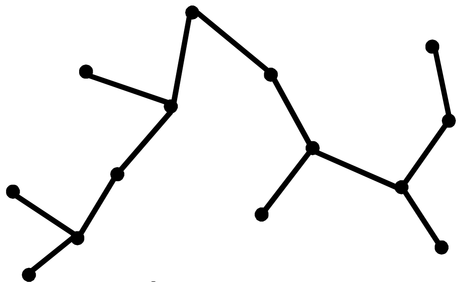
$\forall v_i, v_j \in V \quad \exists p \quad v_i \rightarrow_p v_j$

Connected component: maximal connected subgraph (= $\nexists$ subsets for which the property holds that include it).

Connected undirected graph: only one connected component.

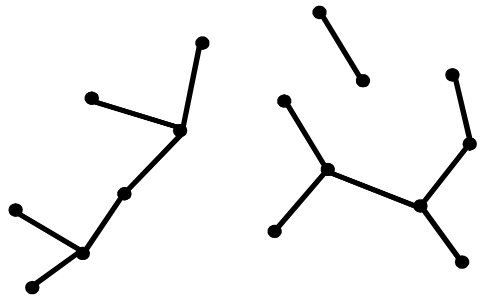
# Non rooted (free) trees

Non rooted tree = undirected, connected, acyclic graph



Forest = undirected acyclic graph

not connected --> set of free trees



# Properties

for a free tree

$G = (V, E)$ undirected graph $|E|$ edges, $|V|$ nodes:

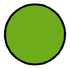
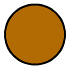
- G = non rooted tree
- Every pair of nodes is connected by a single simple path
- G connected, removing an edge disconnects the graph

  because it has the minimum number of edges
- G connected and $|E| = |V| - 1$
- G acyclic and $|E| = |V| - 1$

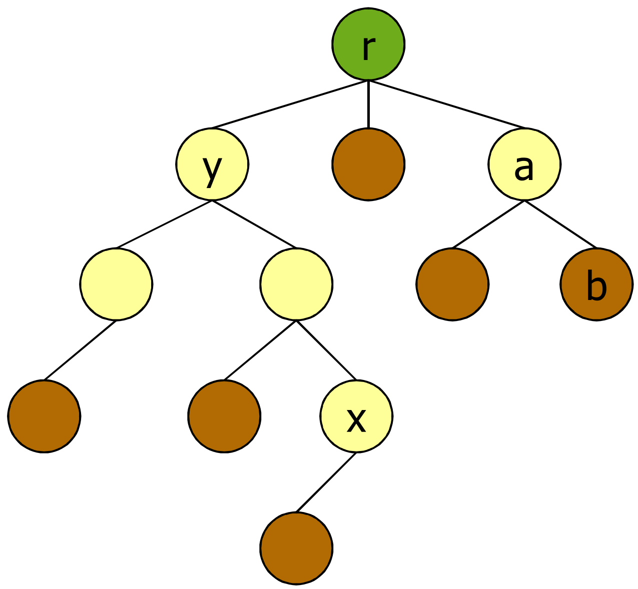
  |x| --> the number of x
- G acyclic, adding an edge introduces a loop.

# Rooted trees

∃ a node r called root special node in the tree with parent/child rel.

- parent/child relationship
  - y ancestor of x if y belongs to the path from r to x. x descendant of y
  - proper ancestor/descendant if $x \neq y$
  - parent/child: adjacent nodes
- root: no parent 
- leaves: no children 

# Example




r root

y proper ancestor of di x

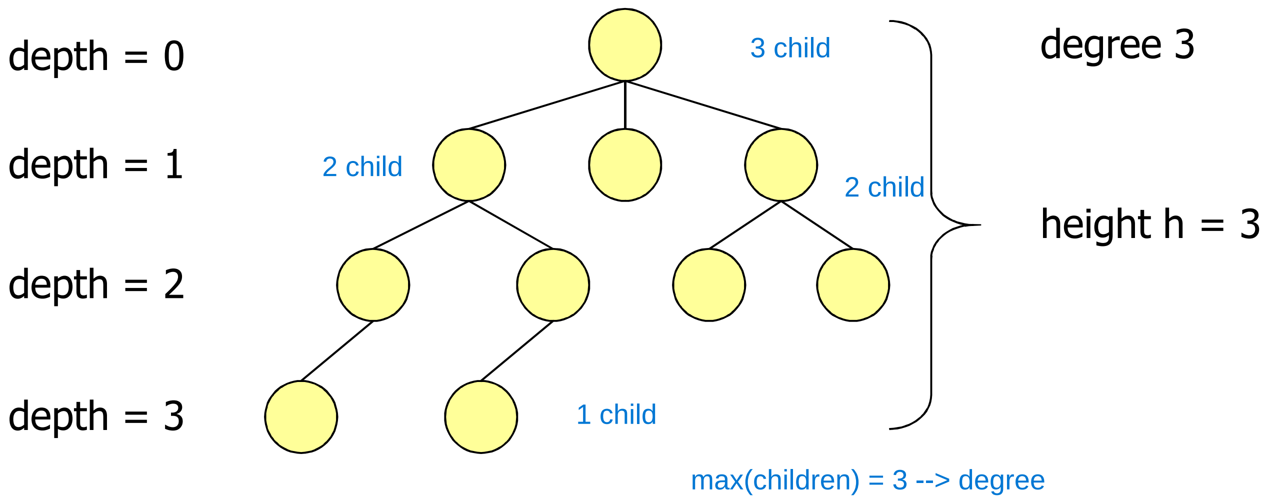
x proper descendant of y

a parent of b

b child of a

# Properties of a tree T

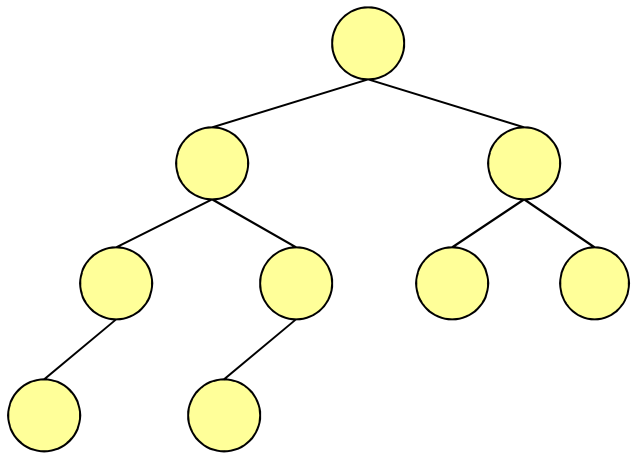
- degree(T) = maximum number of children
- depth(x) = length of the (unique) path from r to x
- height(T) = maximum depth.



# Binary Tree

Definition:

- Tree of degree 2: each node has 0, 1 or 2 children
- There is also a recursive definition (topic of the 2nd-year course)

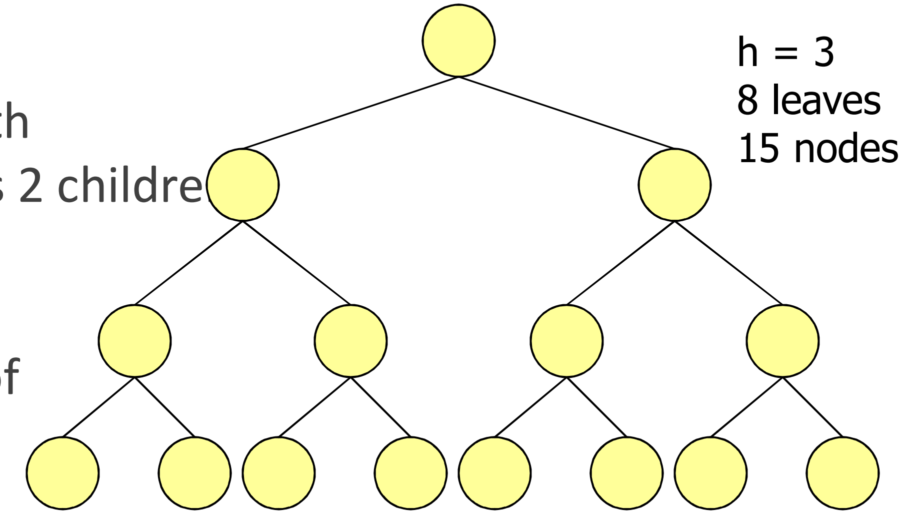


# Perfectly balanced (full) binary tree

Two conditions:

- All the leaves have the same depth
- Each node is either a leaf or it has 2 childre

leaves have the same distance (=3) to the root




Perfectly balanced (full) binary tree of height h:

- Number of leaves: $2^h$
- Number of nodes: $\Sigma_{0 \le i \le h} 2^i = 2^{h+1} - 1$

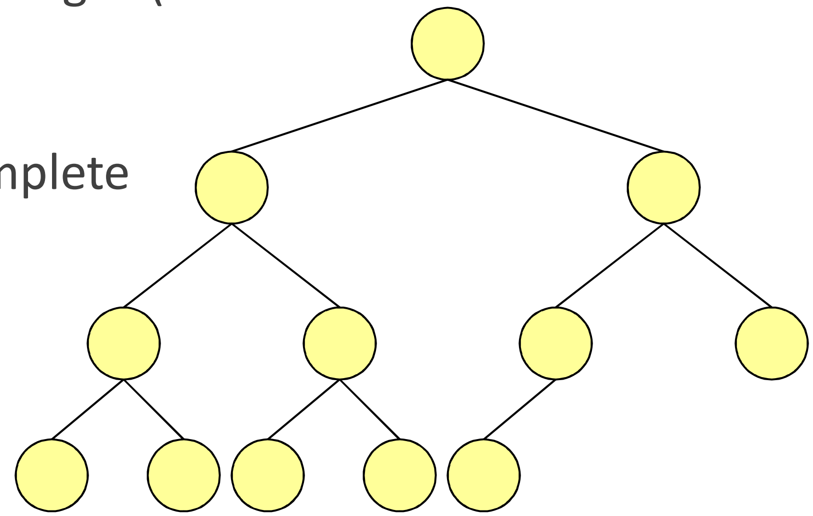
Finite geometric progression with ratio =2


Hyphaene Compressa - Doum Palm
© Shlomit Pinter

# Complete (to the left) binary tree

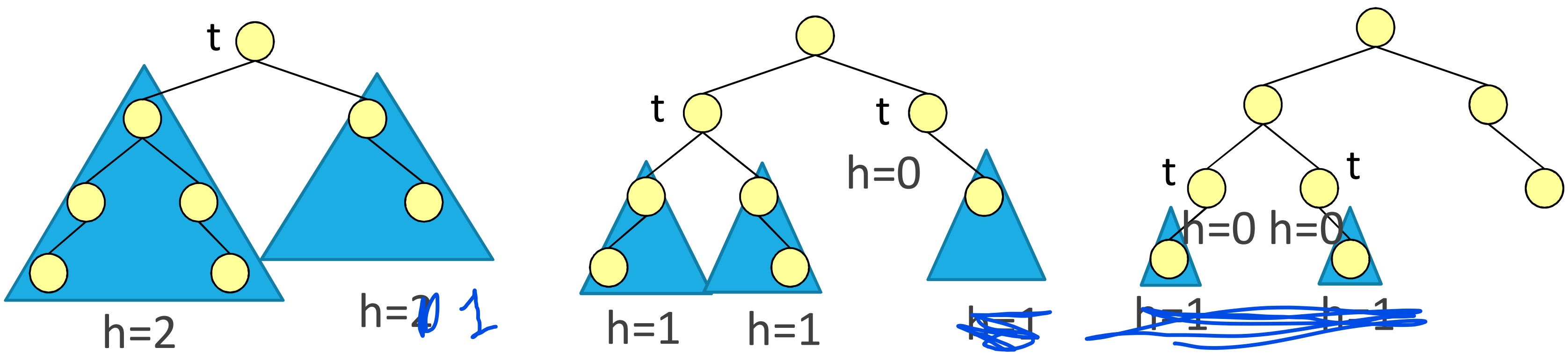
Each level is filled with all possible nodes for that level, possibly except the last one, filled from left to right (all nodes are as far left as possible).

Given the number of nodes n, the complete (o the left) binary tree always exists and is unique.

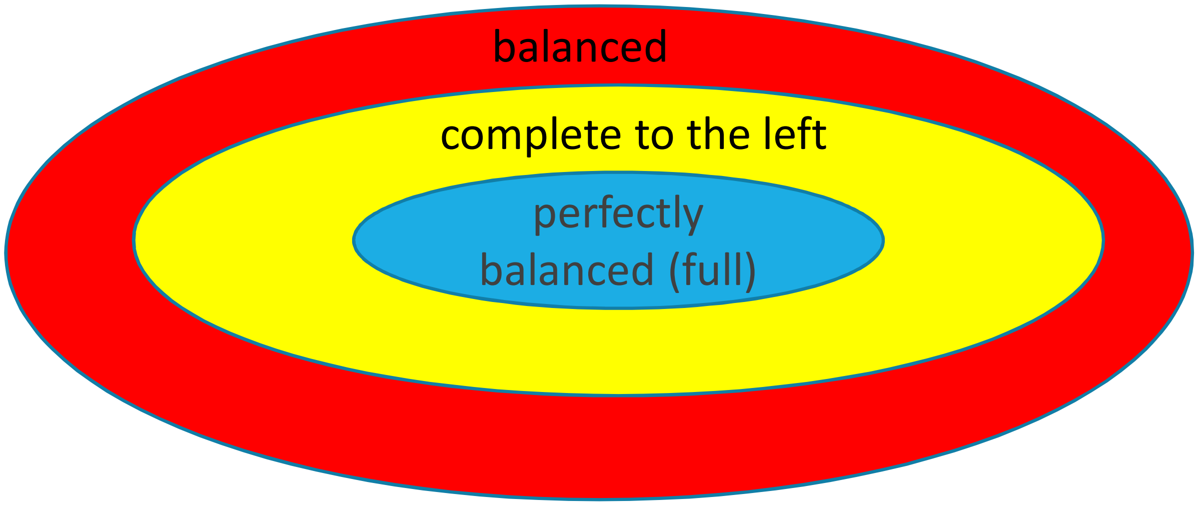


# Height-balanced binary tree

A binary tree is **height-balanced** (or **balanced**) if and only if, for every subtree t rooted in one of its nodes the heights of the left and right subtrees differ for at most 1.

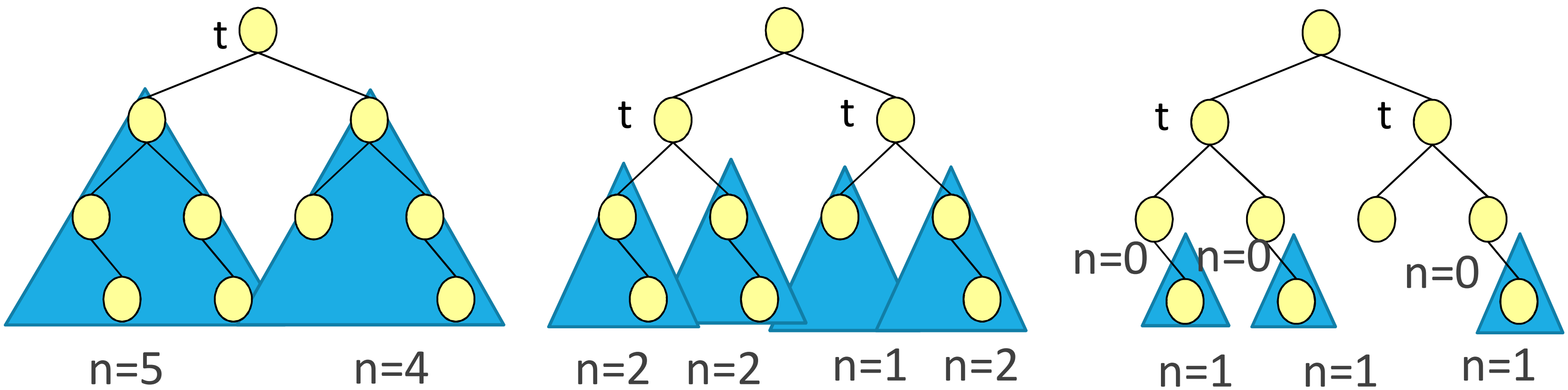


Perfectly balanced (full) binary trees are a proper subset of the complete (to the left) binary trees. In turn, complete (to the left) binary trees are a proper subset of balanced trees.



# Node-balanced binary tree

A binary tree is **node-balanced** if and only if, for each subtree t rooted at one of its nodes, the numbers of nodes in the left and right subtrees differ at most by 1.



# Lower Bound($\Omega$)

Goal: to find a lower bound on worst-case asymptotic complexity for ALL sorting algorithms based on comparison.

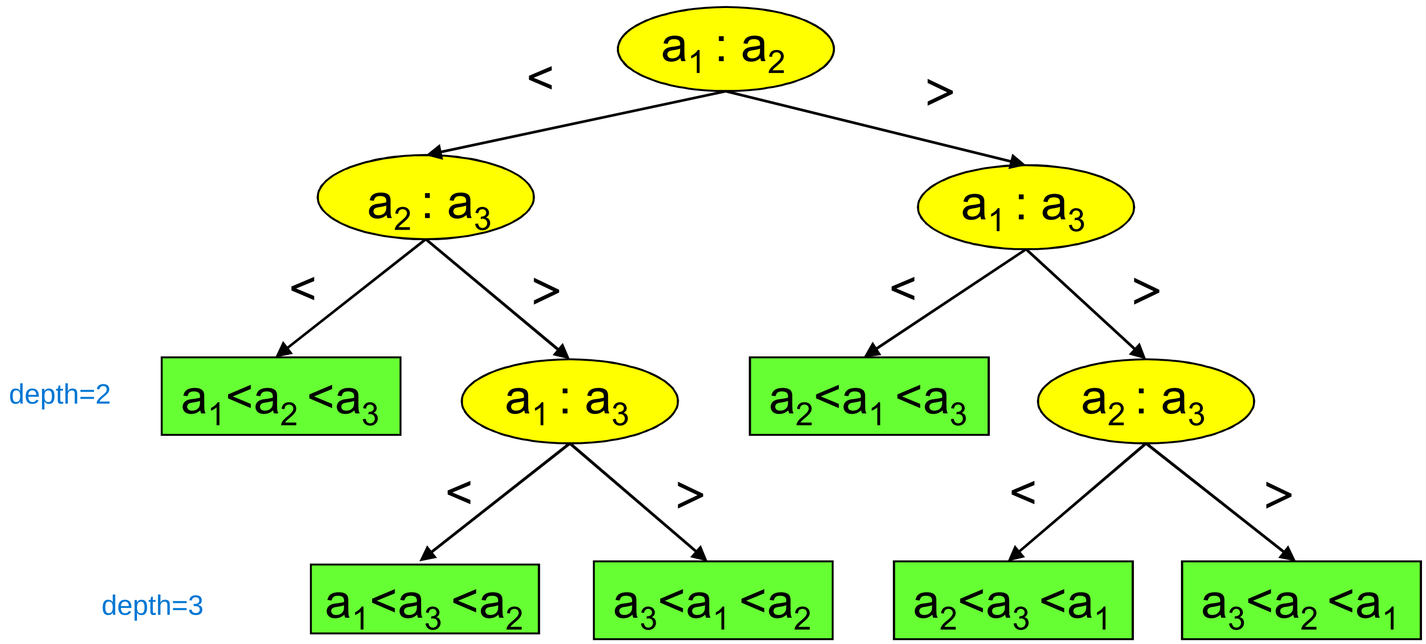
Demonstration is algorithm-INDEPENDENT.

Basic operation: comparison between 2 items $a_i : a_j$
Outcome: decision ($a_i > a_j$ or $a_i \leq a_j$), shown on a binary tree called decision tree.

# Example

Find the complexity of sorting 3-item array A with distinct items $a_1$, $a_2$, $a_3$.

Build a decision tree where each node is labelled with the current comparison ($a_i : a_j$) and the 2 edges with the outcome (> or <). Keep on comparing till a solution is found (leaf).




--> worst case complexity

Complexity is related to the number of comparisons.

What is the minimum number of comparisons in the worst case? 3

The decision tree has height h=3. The minimum number of comparisons executed in the worst case equals height h.

$\Downarrow$

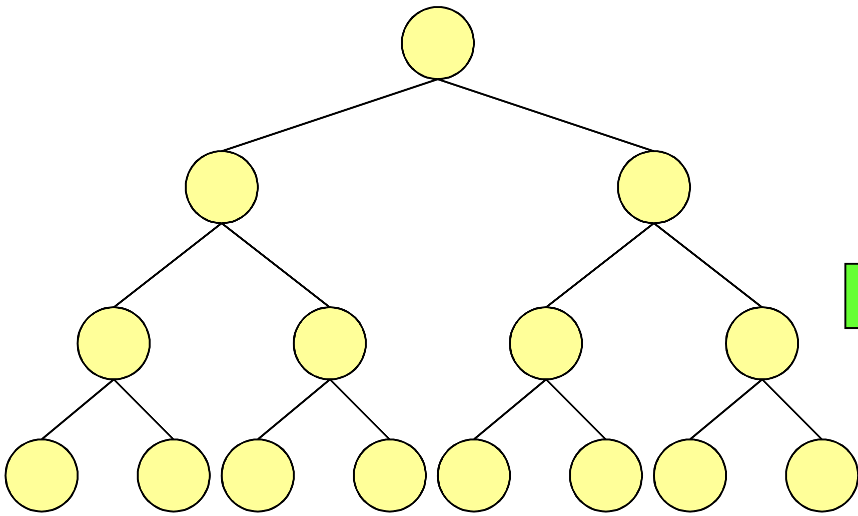
Minimum worst-case complexity is O(h). Complexity is a function of height h, not of the number of items n.

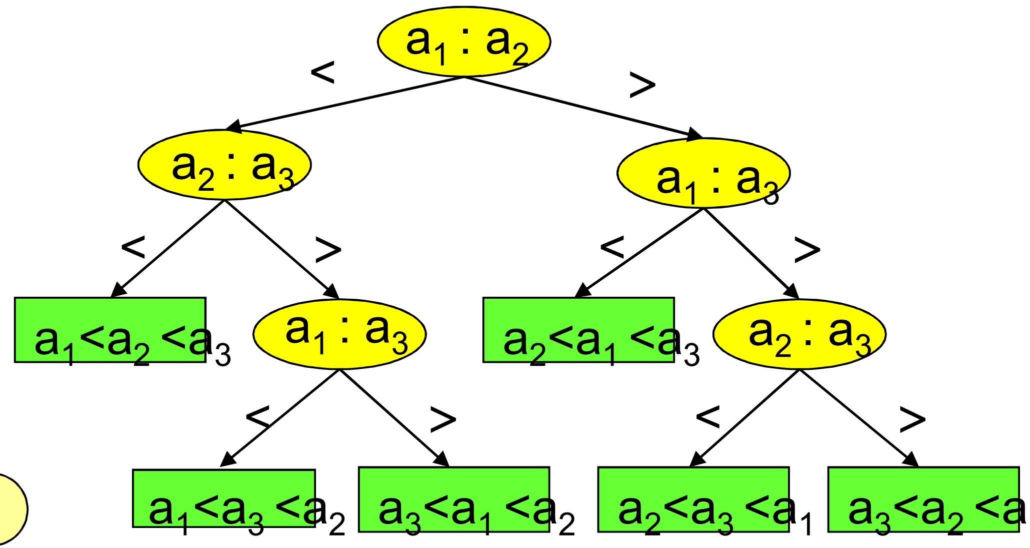
$\Downarrow$

What is the height of a decision tree able to store decisions for an n-item array?

A perfectly balanced (full) binary tree of height h has:

- $2^h$ leaves
- $\Sigma_{0\leq i \leq h} 2^i = 2^{h+1} - 1$ nodes





For n distinct data: the number of possible orderings is the number of permutations $n!$ (number of leaves)

Orderings are stored in tree leaves, so there must be at least as many leaves as the number of orderings

$$2^h \geq n!$$

(number of leaves in terms of height)

Resorting to Stirling's approximation $n! > (n/e)^n$

$$2^h \geq n! > (n/e)^n$$

Taking the log of both sides

$$h > \lg(n/e)^n = n \lg n - \underbrace{n \lg e}_{\text{constant}} = \Omega(n \lg n)$$

Comparison-based sorting algorithms whose worst-case asymptotic complexity is better than linearithmic do not exist.

Comparison-based sorting algorithms whose complexity is $\Omega(n \lg n)$ are **OPTIMAL**.