Algorithms represented as Decision Trees

Internal nodes represent comparison of two elements
Algorithms represented as Decision Trees

Internal nodes represent comparison of two elements $i, j$

Branches represent outcome of comparison

Left: $i \leq j$
Right: $i > j$
Algorithms represented as Decision Trees

Internal nodes represent comparison of two elements $i, j$

Branches represent outcome of comparison

Left: $i \leq j$
Right: $i > j$

Example: sort $a_1, a_2, a_3$
Algorithms represented as Decision Trees

**Internal nodes** represent comparison of two elements \( i, j \)
**Branches** represent outcome of comparison
- **Left**: \( i \leq j \)
- **Right**: \( i > j \)

**Example**: sort \( a_1, a_2, a_3 \)

Each leaf is a possible output.
Algorithms represented as Decision Trees

Internal nodes represent comparison of two elements $i, j$.

Branches represent outcome of comparison:
- Left: $i \leq j$
- Right: $i > j$

Example: sort $a_1, a_2, a_3$

Each leaf is a possible output:
- $a_3 > a_2 > a_1$
- $a_2 > a_3 > a_1$
- $a_1 > a_2 > a_3$
- $a_1 > a_3 > a_2$
- $a_3 > a_1 > a_2$
- $a_2 > a_1 > a_3$
- $a_1 > a_2 > a_3$
- $a_2 > a_3 > a_1$
Algorithms represented as Decision Trees

Internal nodes represent comparison of two elements i,j
Branches represent outcome of comparison
Left: \( i \leq j \)
Right: \( i > j \)

Example: sort \( a_1, a_2, a_3 \)

Verify on 9, 4, 6
\( a_3 > a_2 > a_1 \)

Each leaf is a possible output
Algorithms represented as Decision Trees

Internal nodes represent comparison of two elements \(i, j\)
Branches represent outcome of comparison
- Left: \(i \leq j\)
- Right: \(i > j\)

Example: sort \(a_1, a_2, a_3\)

Verify on \(9, 4, 6\)
\(a_1, a_2, a_3\)

Each leaf is a possible output
Each root-to-leaf path represents an execution of algo.
Algorithms represented as decision trees

Internal nodes represent comparison of two elements $i,j$
Branches represent outcome of comparison
- Left: $i \leq j$
- Right: $i > j$

Example: sort $a_1, a_2, a_3$

Each leaf is a possible output
Each root-to-leaf path represents an execution of algo.

Verify on 9,4,6
$a, a_2, a_3$

$a_3 > a_2 > a_1$

Any decision-based algorithm can be encoded as a decision tree.
If you are designing a decision tree
  it's up to you to avoid comparing the same elements many times.
If you are designing a decision tree, it's up to you to avoid comparing the same elements many times. The worst-case run-time is precisely the longest root-leaf path.
If you are designing a decision tree, it’s up to you to avoid comparing the same elements many times. The worst-case run-time is precisely the longest root-leaf path. You shouldn’t compare $a_i:a_j$ twice on one path. 

$\downarrow$ so max path length $= (n)$

For a good algo

-for sorting
If you are designing a decision tree, it's up to you to avoid comparing the same elements many times. The worst-case run-time is precisely the longest root-leaf path.

\[ \text{so max path length} = \binom{n}{2} \]

Why not write all algorithms this way? (so much prettier than pseudocode)
If you are designing a decision tree, it's up to you to avoid comparing the same elements many times. The worst-case run-time is precisely the longest root-leaf path. You shouldn't compare $a_i:a_j$ twice on one path. 

\[ \text{so max path length} = \binom{n}{2} \]

Why not write all algorithms this way? (so much prettier than pseudocode)

\[ \hookrightarrow \text{It's huge and repetitive.} \]

\[ \hookrightarrow \text{It really lists every possible execution of algo.} \]
If you are designing a decision tree, it's up to you to avoid comparing the same elements many times. The worst-case run-time is precisely the longest root-leaf path. You shouldn't compare \( a_i : a_j \) twice on one path. \( \Downarrow \) so max path length = \( \binom{n}{2} \)

Why not write all algorithms this way? (so much prettier than pseudocode)

\( \Downarrow \) It's huge and repetitive.
   It really lists every possible execution of algo.
   You actually might need a different tree for each \( n \).
If you are designing a decision tree, it's up to you to avoid comparing the same elements many times. The worst-case run-time is precisely the longest root-leaf path. You shouldn't compare \( a_i : a_j \) twice on one path.

\[ \Rightarrow \text{so max path length} = (n) \]

Why not write all algorithms this way? (so much prettier than pseudocode)

\[ \Rightarrow \text{It's huge and repetitive.} \]

It really lists every possible execution of algo. You actually might need a different tree for each \( n \).

What is the shortest possible tree for comparison-sort?
A correct decision tree for sorting must have every possible output represented at a leaf node.

#leaves » ?
A correct decision tree for sorting must have every permutation of the input represented at a leaf node.

#leaves ?
A correct decision tree for sorting must have every permutation of the input represented at a leaf node.

\( \# \text{leaves} \geq n! \)
A correct decision tree for sorting must have every permutation of the input represented at a leaf node.

\[ \#\text{leaves} \geq n! \]

height of tree = worst case time = h \quad \Rightarrow \quad \#\text{leaves} \leq ?
A correct decision tree for sorting must have every permutation of the input represented at a leaf node. \(\#\text{leaves} \geq n!\)

Height of tree = worst case time = \(h\) \(\implies\) \(\#\text{leaves} \leq 2^h\)

[Binary tree; every node has 2 children]
A correct decision tree for sorting must have every permutation of the input represented at a leaf node.

\[ \#\text{leaves} \geq n! \]

height of tree = worst case time = \( h \) \( \implies \#\text{leaves} \leq 2^h \) [binary tree; every node has 2 children]

so, \( n! \leq \#\text{leaves} \leq 2^h \)
A correct decision tree for sorting must have every permutation of the input represented at a leaf node.

\[ \#\text{leaves} \geq n! \]

Height of tree = worst case time = \( h \) \( \Rightarrow \#\text{leaves} \leq 2^h \) [binary tree; every node has 2 children]

So, \( n! \leq \#\text{leaves} \leq 2^h \) \( \Rightarrow \log n! \leq \log 2^h \)
A correct decision tree for sorting must have every permutation of the input represented at a leaf node.

\[ \#\text{leaves} \geq n! \]

height of tree = worst case time = \( h \) \( \Rightarrow \) \#leaves \( \leq 2^h \)

[binary tree; every node has 2 children]

so, \( n! \leq \#\text{leaves} \leq 2^h \) \( \Rightarrow \) \( \log n! \leq \log 2^h \) \( \Rightarrow \) \( h \geq \log n! \)
A correct decision tree for sorting must have every permutation of the input represented at a leaf node.

- \( \# \text{leaves} \geq n! \)

- Height of tree = worst case time = \( h \) \( \Rightarrow \) \( \# \text{leaves} \leq 2^h \)  
  [binary tree; every node has 2 children]

So, \( n! \leq \# \text{leaves} \leq 2^h \) \( \Rightarrow \) \( \log n! \leq \log 2^h \) \( \Rightarrow \) \( h \geq \log n! \)

Stirling's formula: \( n! \geq (\frac{n}{e})^n \)
A correct decision tree for sorting must have every permutation of the input represented at a leaf node.

\[ \#\text{leaves} \geq n! \]

height of tree = worst case time = \( h \) \( \Rightarrow \) \( \#\text{leaves} \leq 2^h \)  

[binary tree; every node has 2 children]

so, \( n! \leq \#\text{leaves} \leq 2^h \) \( \Rightarrow \) \( \log n! \leq \log 2^h \) \( \Rightarrow \) \( h \geq \log n! \)

Stirling's formula: \( n! \approx (\frac{n}{e})^n \)

\( h \approx \log(\frac{n}{e})^n = n \cdot \log \frac{n}{e} \)
A correct decision tree for sorting must have every permutation of the input represented at a leaf node.

\[
\text{\#leaves} \geq n!
\]

height of tree = worst case time = \( h \) \( \implies \) \#leaves \( \leq 2^h \)

[binary tree; every node has 2 children]

so, \( n! \leq \#\text{leaves} \leq 2^h \) \( \implies \) \( \log n! \leq \log 2^h \implies h \geq \log n! \)

Stirling's formula: \( n! \approx (\frac{n}{e})^n \)

\[
\log n! \geq \log (\frac{n}{e})^n = n \cdot \log \frac{n}{e}
\]

\[
= n \log n - \log e = n \log n - \Theta(n)
\]

\[
h = \Omega(n \log n)
\]

extra analysis of \( \log n! \) follows
\[ \log(n!) = O(n \log n) \]
\[ \log(n!) \leq \log(n^n) = n \log n \Rightarrow \log(n!) \leq 1 \cdot n \log n \quad \text{for } n \geq 1 \]

\[ \log(n!) = \Omega(n \log n) \]

\[ \log(n!) = \log(n \cdot (n-1) \cdot (n-2) \cdot (n-3) \cdots \cdots 3 \cdot 2 \cdot 1) \]
\[ = \log(n \cdot \underbrace{1 \cdot (n-1)}_{\text{even}} \cdot \underbrace{2 \cdot (n-2)}_{\text{even}} \cdot \underbrace{3 \cdot (n-3)}_{\text{even}} \cdot 4 \cdots \cdots \cdots \underbrace{n\cdot(n-\frac{n}{2})\cdot(n-\frac{n}{2})}_{\text{even}} ) \]
\[ \geq \log(n \cdot n \cdot n \cdot \cdots \cdots \cdots n) \]
\[ = \log(n^{n/2}) \quad \text{(assume } n: \text{even}) \Rightarrow \log(n!) \geq \frac{n}{2} \log n \]

\[ \frac{1}{2} n \log n \leq \log(n!) \leq n \log n \]

In fact, Stirling's approximation:
\[ \ln(n!) = n \cdot \ln(n) - n + O(\ln(n)) \]