CS 381

Week 1
Part 3

Divide-and-Conquer

Divide-and-conquer.
- Break up problem into several parts.
- Solve each part recursively.
- Combine solutions to sub-problems into overall solution.

Most common usage.
- Break up problem of size n into two equal parts of size $\frac{1}{2}n$.
- Solve two parts recursively.
- Combine two solutions into overall solution in linear time.

Consequence.
- Brute force: n^2.
- Divide-and-conquer: $n \log n$.

Divide et impera.
Veni, vidi, vici.
- *Julius Caesar*

Mergesort

Sorting. Given n elements, rearrange in ascending order.

Obvious sorting applications.
- List files in a directory.
- Organize an MP3 library.
- List names in a phone book.
- Display Google PageRank results.

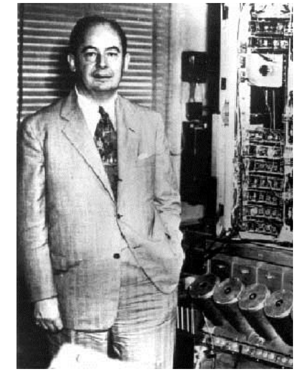
Problems become easier once sorted.
- Find the median.
- Find the closest pair.
- Binary search in a database.
- Identify statistical outliers.
- Find duplicates in a mailing list.

Non-obvious sorting applications.
- Data compression.
- Computer graphics.
- Interval scheduling.
- Computational biology.
- Minimum spanning tree.
- Supply chain management.
- Simulate a system of particles.
- Book recommendations on Amazon.
- Load balancing on a parallel computer.
- . . .

Mergesort

Mergesort.

- Divide array into two halves.
- Recursively sort each half.
- Merge two halves to make sorted whole.



Jon von Neumann (1945)

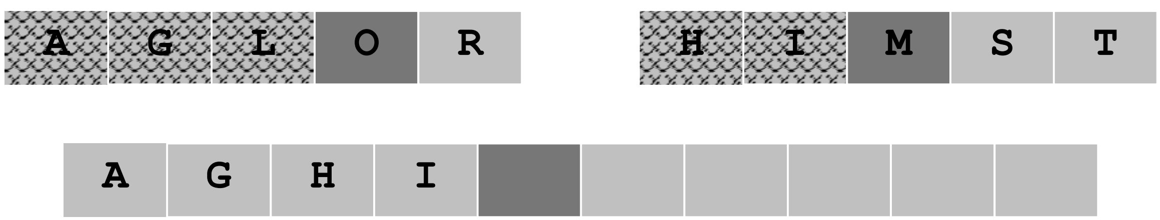
A	L	G	O	R	I	T	H	M	S			
A L G O R					I T H M S		divide	$O(1)$
A G L O R					H I M S T		sort	$2T(n/2)$
A	G	H	I	L	M	O	R	S	T		merge	$O(n)$

Merging

Merging. Combine two pre-sorted lists into a sorted whole.

How to merge efficiently?

- Linear number of comparisons.
- Use temporary array.

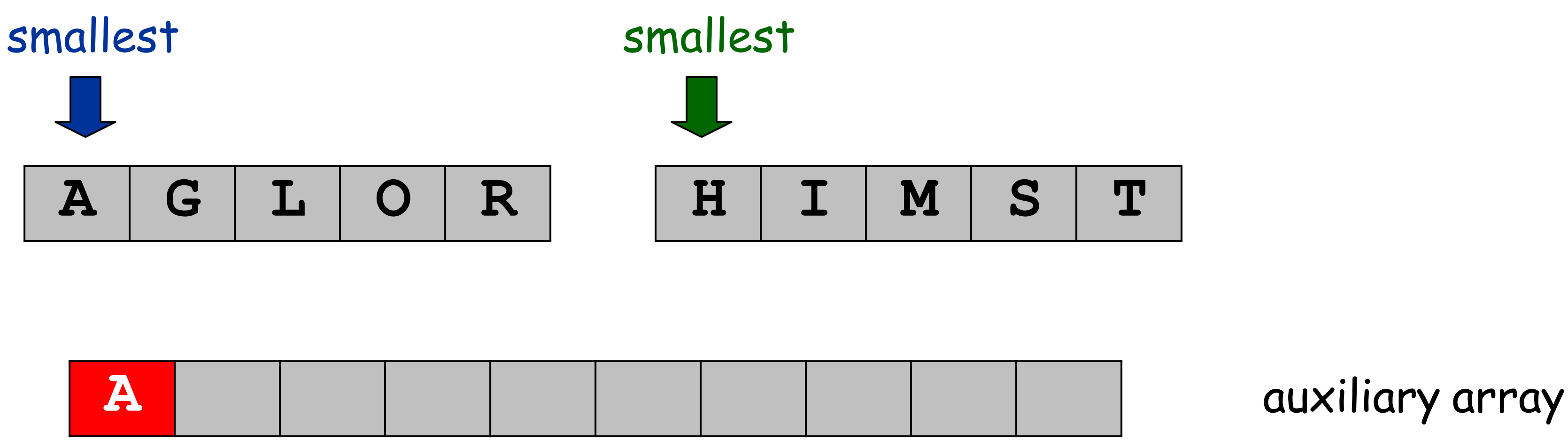


Challenge for the bored. In-place merge. [Kronrud, 1969]

using only a constant amount of extra storage

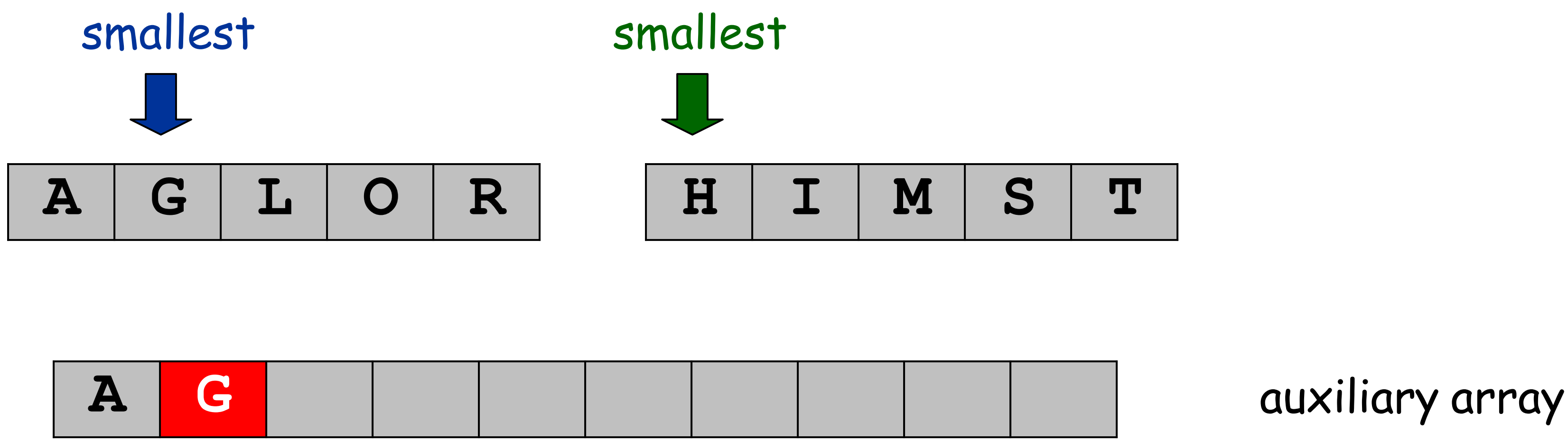
Merging

Merge.
- Keep track of smallest element in each sorted half.
- Insert smallest of two elements into auxiliary array.
- Repeat until done.



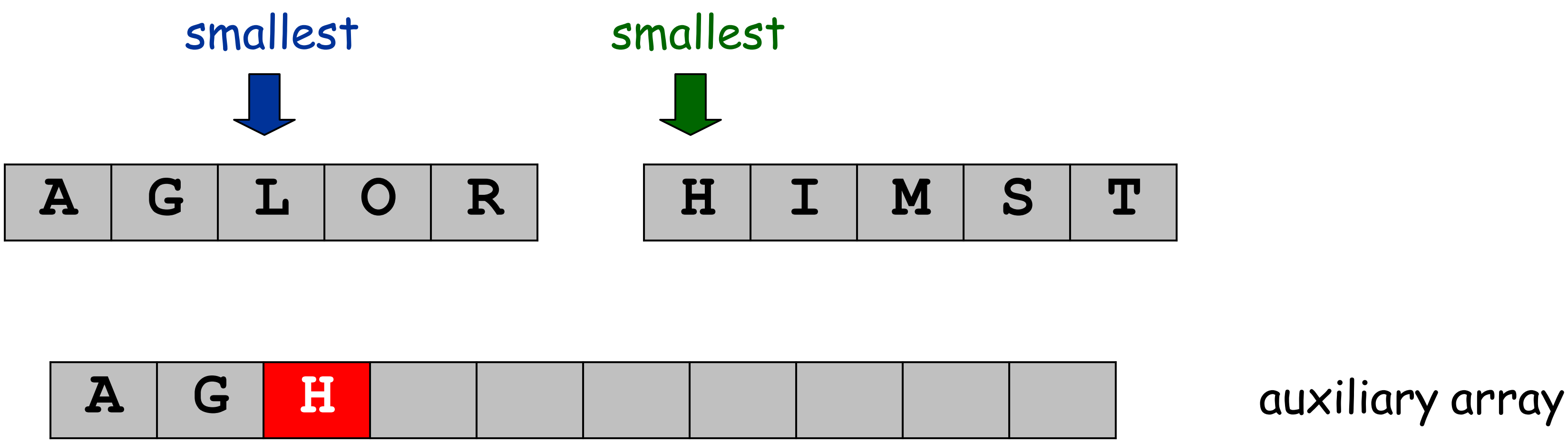
Merge.

- Keep track of smallest element in each sorted half.
- Insert smallest of two elements into auxiliary array.
- Repeat until done.



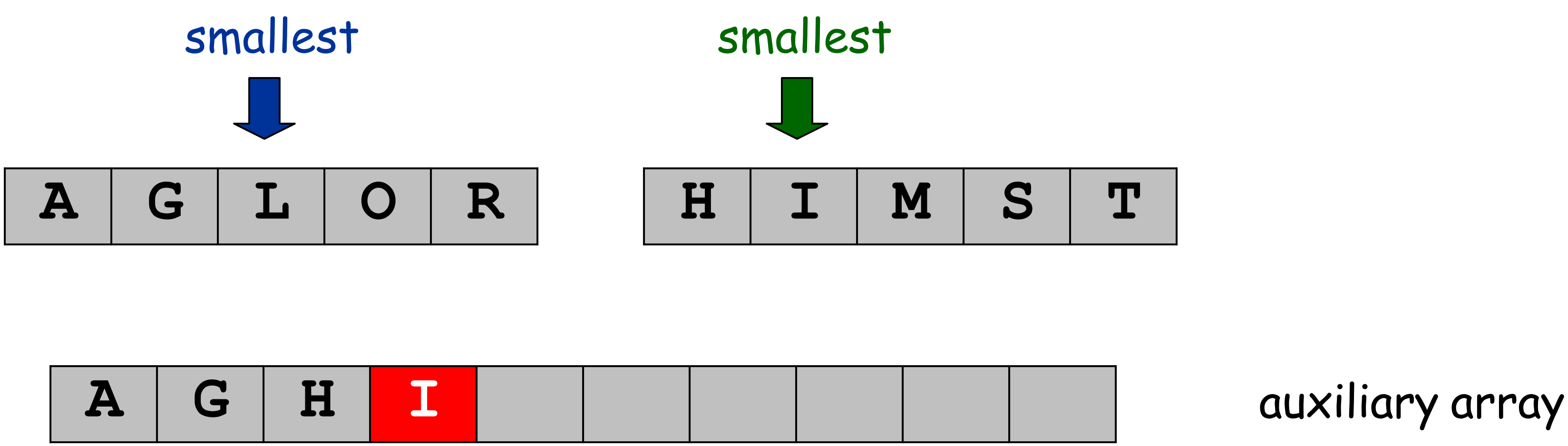
Merging

Merge.
- Keep track of smallest element in each sorted half.
- Insert smallest of two elements into auxiliary array.
- Repeat until done.



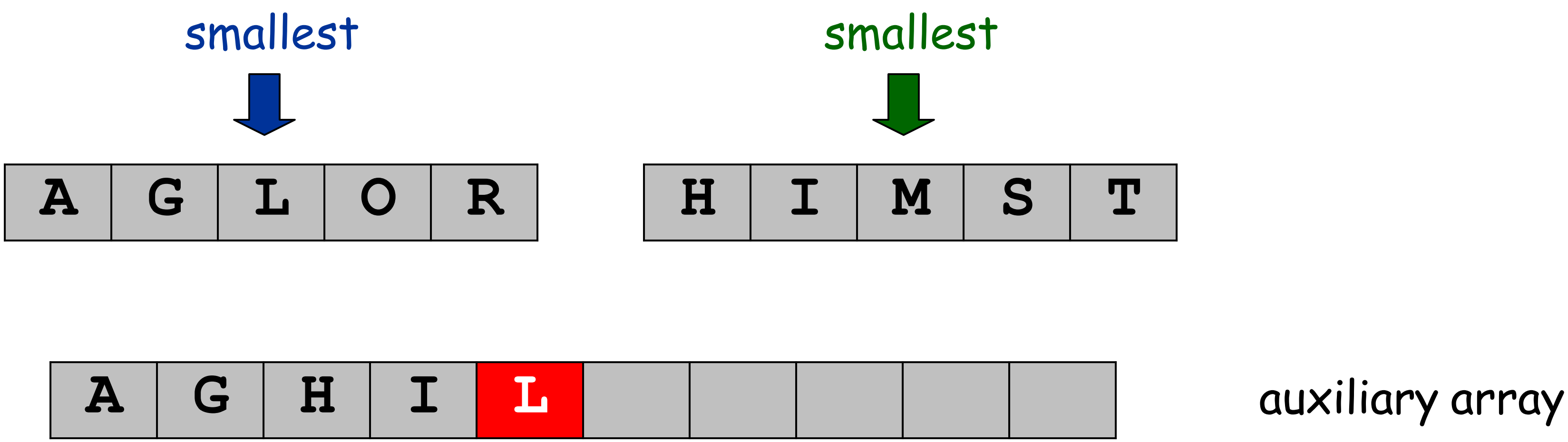
Merging

Merge.
- Keep track of smallest element in each sorted half.
- Insert smallest of two elements into auxiliary array.
- Repeat until done.



Merging

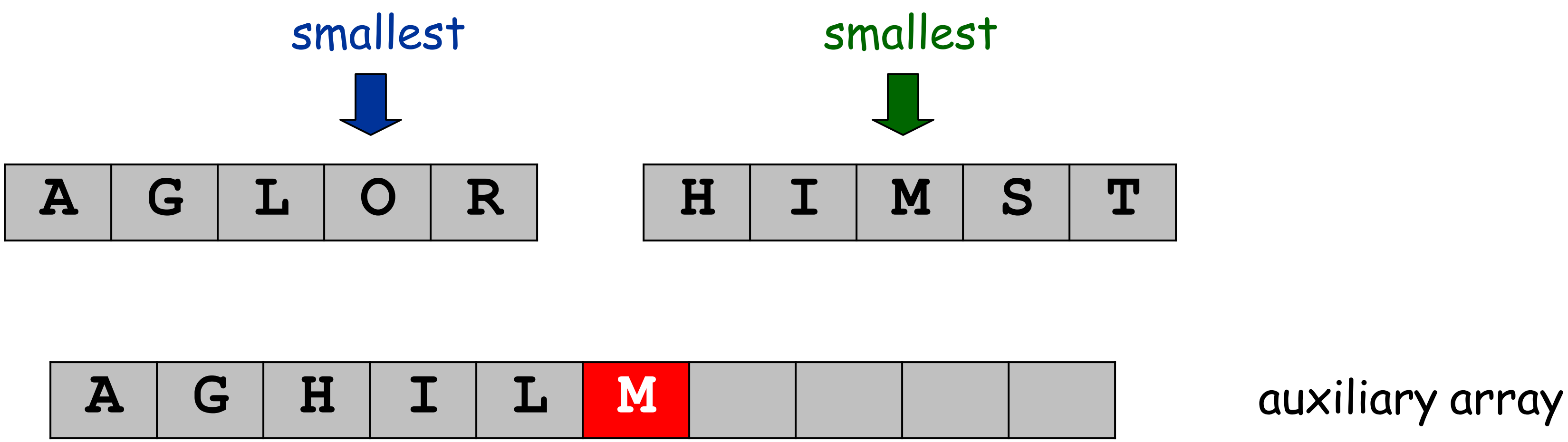
Merge.
- Keep track of smallest element in each sorted half.
- Insert smallest of two elements into auxiliary array.
- Repeat until done.



Merging

Merge.

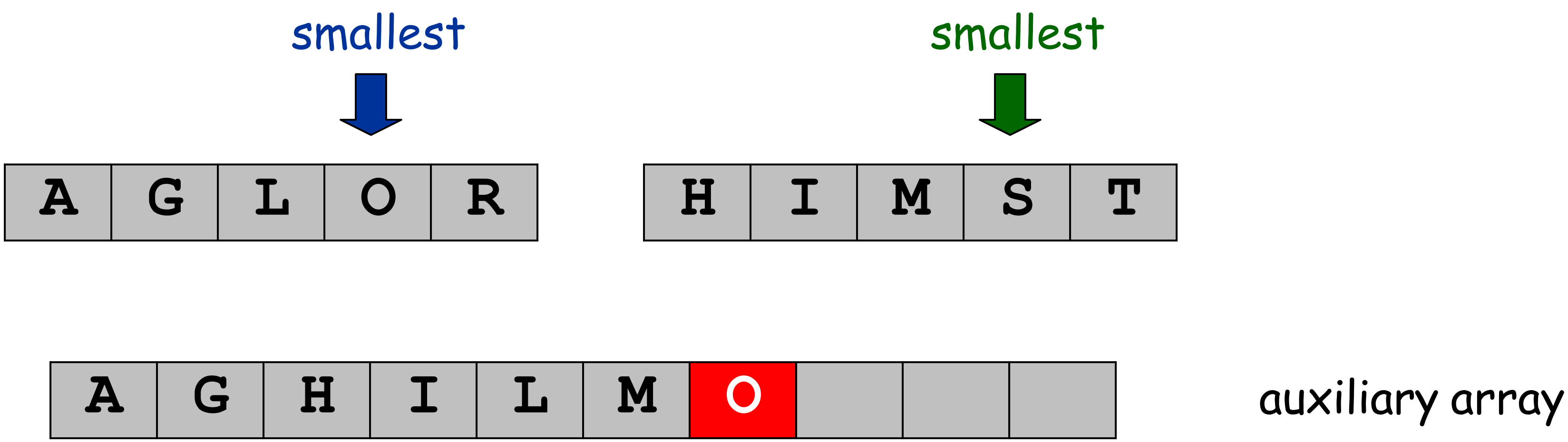
- Keep track of smallest element in each sorted half.
- Insert smallest of two elements into auxiliary array.
- Repeat until done.



Merging

Merge.

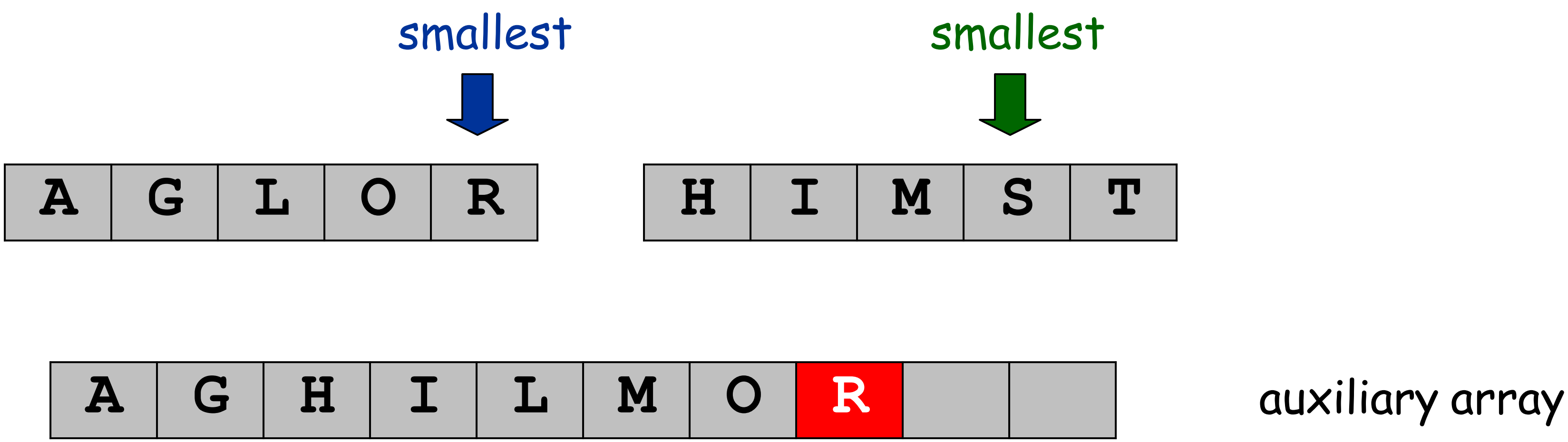
- Keep track of smallest element in each sorted half.
- Insert smallest of two elements into auxiliary array.
- Repeat until done.



Merging

Merge.

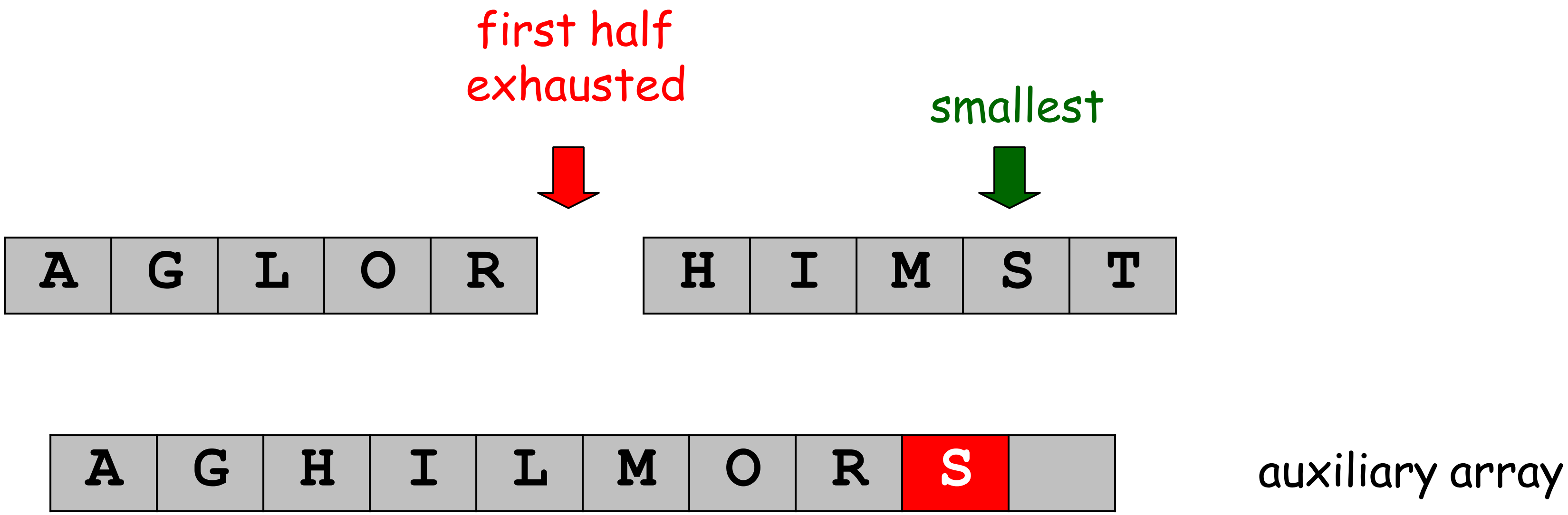
- Keep track of smallest element in each sorted half.
- Insert smallest of two elements into auxiliary array.
- Repeat until done.



Merging

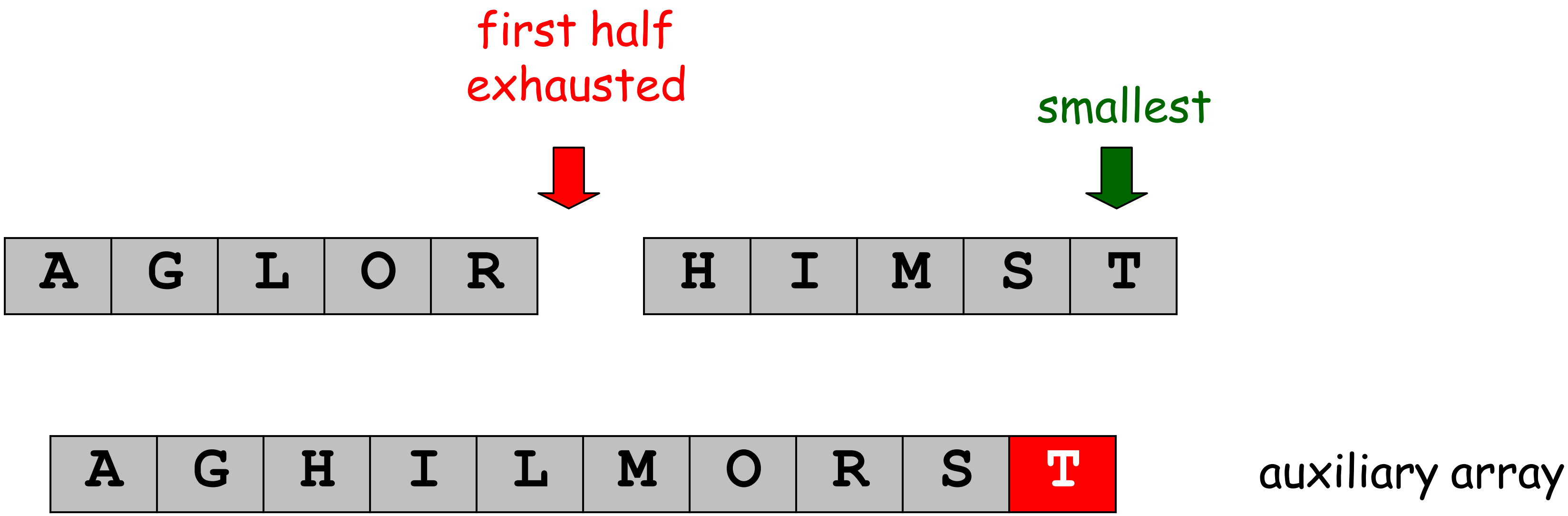
Merge.

- Keep track of smallest element in each sorted half.
- Insert smallest of two elements into auxiliary array.
- Repeat until done.



Merging

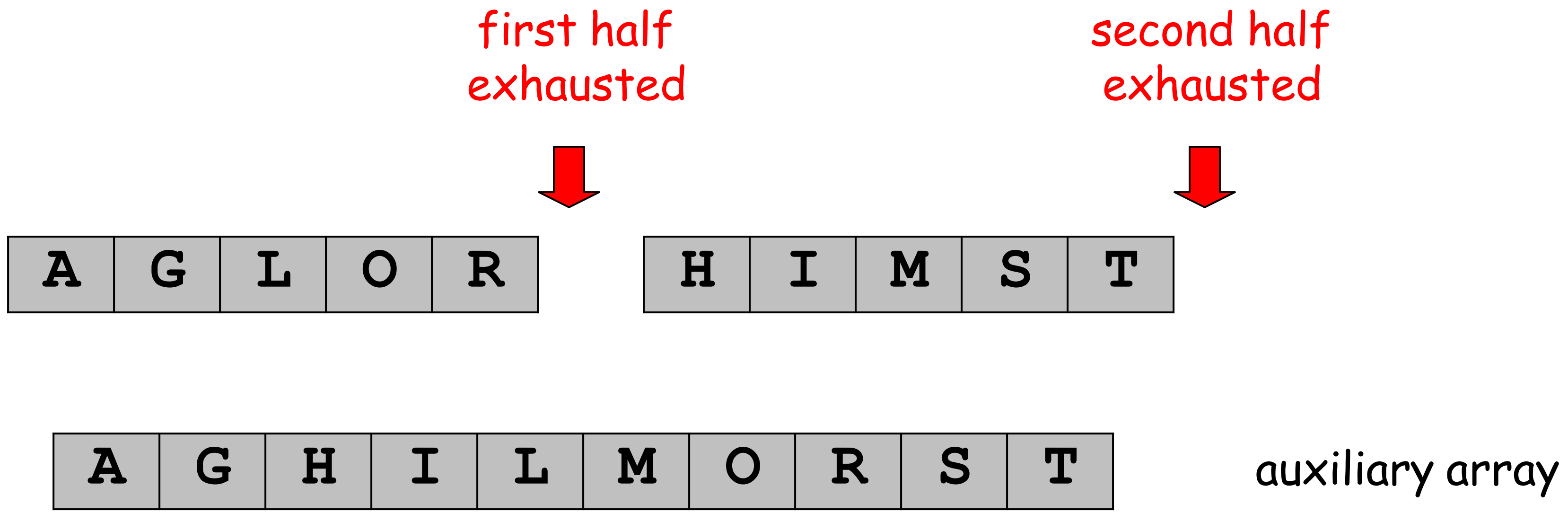
Merge.
- Keep track of smallest element in each sorted half.
- Insert smallest of two elements into auxiliary array.
- Repeat until done.



Merging

Merge.

- Keep track of smallest element in each sorted half.
- Insert smallest of two elements into auxiliary array.
- Repeat until done.



A Useful Recurrence Relation

Def. T(n) = number of comparisons to mergesort an input of size n.

Mergesort recurrence.

$$T(n) \le \begin{cases} 0 & \text{if } n = 1 \\ \underbrace{T(\lceil n/2 \rceil)}_{\text{solve left half}} + \underbrace{T(\lfloor n/2 \rfloor)}_{\text{solve right half}} + \underbrace{n}_{\text{merging}} & \text{otherwise} \end{cases}$$

Solution. $T(n) = O(n \log_2 n)$.

Assorted proofs. We describe several ways to prove this recurrence. Initially we assume n is a power of 2 and replace $\le$ with =.

Solving T(n)

Claim. If T(n) satisfies this recurrence, then $T(n) = n \log_2 n$.

assumes n is a power of 2

$$T(n) = \begin{cases} 0 & \text{if } n = 1 \\ \underbrace{2T(n/2)}_{\text{sorting both halves}} + \underbrace{n}_{\text{merging}} & \text{otherwise} \end{cases}$$

Proof: In class exercise.

Proof by Telescoping

Claim. If T(n) satisfies this recurrence, then $T(n) = n \log_2 n$.

assumes n is a power of 2

$$T(n) = \begin{cases} 0 & \text{if } n = 1 \\ \underbrace{2T(n/2)}_{\text{sorting both halves}} + \underbrace{n}_{\text{merging}} & \text{otherwise} \end{cases}$$

Pf. For n > 1:

$$\begin{aligned}
\frac{T(n)}{n} &= \frac{2T(n/2)}{n} + 1 \\
&= \frac{T(n/2)}{n/2} + 1 \\
&= \frac{T(n/4)}{n/4} + 1 + 1 \\
&\cdots \\
&= \frac{T(n/n)}{n/n} + \underbrace{1 + \cdots + 1}_{\log_2 n} \\
&= \log_2 n
\end{aligned}$$

Proof by Induction

Claim. If T(n) satisfies this recurrence, then $T(n) = n \log_2 n$.

assumes n is a power of 2

$$T(n) = \begin{cases} 0 & \text{if } n = 1 \\ \underbrace{2T(n/2)}_{\text{sorting both halves}} + \underbrace{n}_{\text{merging}} & \text{otherwise} \end{cases}$$

Pf. (by induction on n)

- Base case:
- Inductive hypothesis:
- Goal:

Proof by Induction

Claim. If T(n) satisfies this recurrence, then $T(n) = n \log_2 n$.

assumes n is a power of 2

$$T(n) = \begin{cases} 0 & \text{if } n = 1 \\ \underbrace{2T(n/2)}_{\text{sorting both halves}} + \underbrace{n}_{\text{merging}} & \text{otherwise} \end{cases}$$

Pf. (by induction on n)

- Base case: n = 1.
- Inductive hypothesis:
- Goal:

Proof by Induction

Claim. If T(n) satisfies this recurrence, then $T(n) = n \log_2 n$.

assumes n is a power of 2

$$T(n) = \begin{cases} 0 & \text{if } n = 1 \\ \underbrace{2T(n/2)}_{\text{sorting both halves}} + \underbrace{n}_{\text{merging}} & \text{otherwise} \end{cases}$$

Pf. (by induction on n)

- Base case: $n = 1$.
- Inductive hypothesis: $T(n) = n \log_2 n$.
- Goal:

Proof by Induction

Claim. If T(n) satisfies this recurrence, then $T(n) = n \log_2 n$.

assumes n is a power of 2

$$T(n) = \begin{cases} 0 & \text{if } n = 1 \\ \underbrace{2T(n/2)}_{\text{sorting both halves}} + \underbrace{n}_{\text{merging}} & \text{otherwise} \end{cases}$$

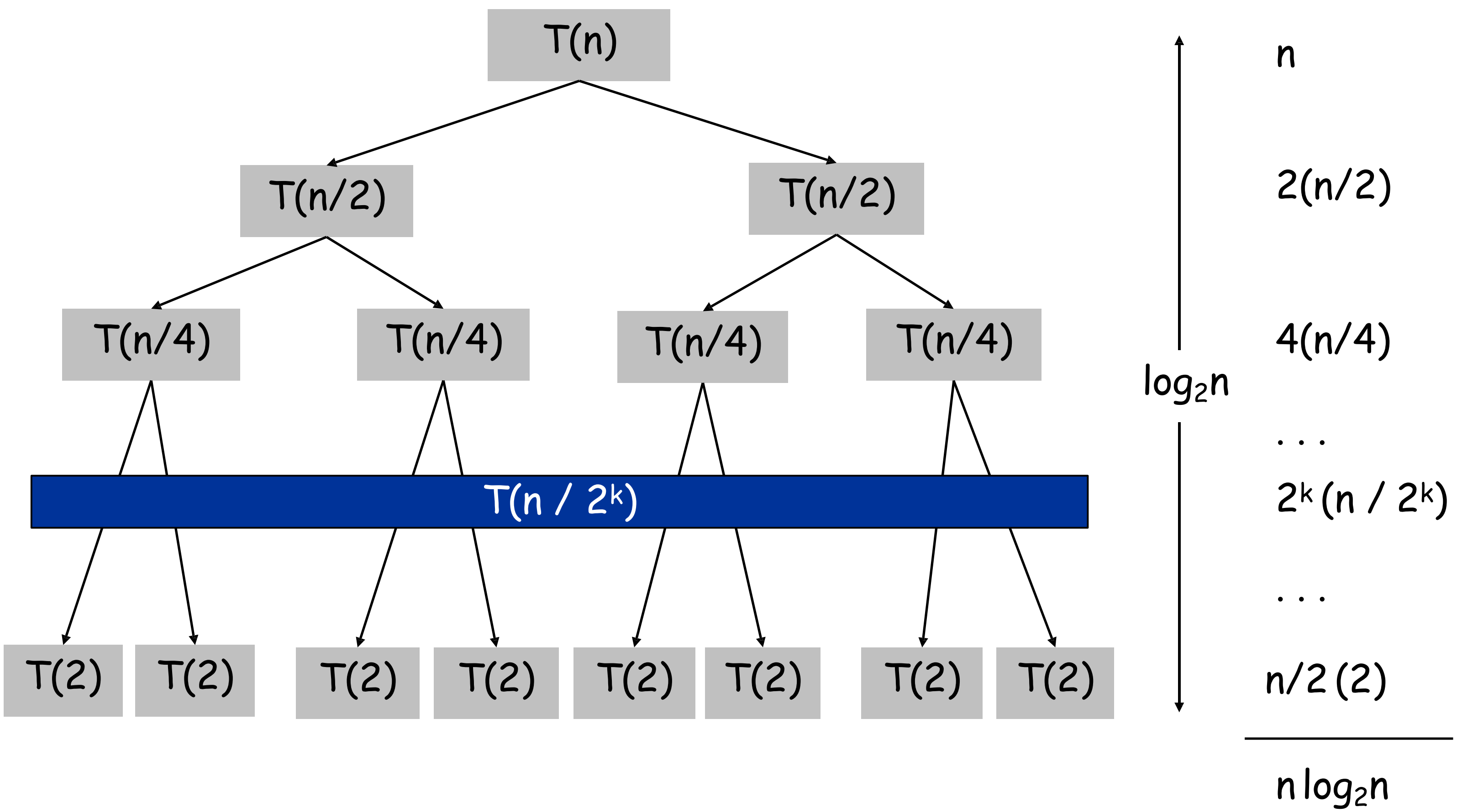
Pf. (by induction on n)

- Base case: n = 1.
- Inductive hypothesis: $T(n) = n \log_2 n$.
- Goal: show that $T(2n) = 2n \log_2 (2n)$.

$$\begin{aligned} T(2n) &= 2T(n) + 2n \\ &= 2n \log_2 n + 2n \\ &= 2n\left(\log_2 (2n) - 1\right) + 2n \\ &= 2n \log_2 (2n) \end{aligned}$$

Proof Visualized

$$T(n) = \begin{cases} 0 & \text{if } n = 1 \\ \underbrace{2T(n/2)}_{\text{sorting both halves}} + \underbrace{n}_{\text{merging}} & \text{otherwise} \end{cases}$$



Analysis of Mergesort Recurrence

Claim. If T(n) satisfies the following recurrence, then $T(n) \le n \lceil \lg n \rceil$.

$\uparrow$ $\log_2 n$

$$T(n) \le \begin{cases} 0 & \text{if } n = 1 \\ \underbrace{T(\lceil n/2 \rceil)}_{\text{solve left half}} + \underbrace{T(\lfloor n/2 \rfloor)}_{\text{solve right half}} + \underbrace{n}_{\text{merging}} & \text{otherwise} \end{cases}$$

Pf. (by induction on n)

- Base case: n = 1.
- Define $n_1 = \lfloor n / 2 \rfloor$, $n_2 = \lceil n / 2 \rceil$.
- Induction step: assume true for 1, 2, ... , n–1.

$$\begin{aligned} T(n) &\le T(n_1) + T(n_2) + n \\ &\le n_1 \lceil \lg n_1 \rceil + n_2 \lceil \lg n_2 \rceil + n \\ &\le n_1 \lceil \lg n_2 \rceil + n_2 \lceil \lg n_2 \rceil + n \\ &= n \lceil \lg n_2 \rceil + n \\ &\le n(\lceil \lg n \rceil - 1) + n \\ &= n \lceil \lg n \rceil \end{aligned}$$

$$\begin{aligned} n_2 &= \lceil n/2 \rceil \\ &\le \left\lceil 2^{\lceil \lg n \rceil} / 2 \right\rceil \\ &= 2^{\lceil \lg n \rceil} / 2 \\ \Rightarrow \lg n_2 &\le \lceil \lg n \rceil - 1 \end{aligned}$$

Lower Bounds for Sorting Algorithms

Merge Sort: O(n * log(n))

Q: Can we do better?

Answer: It depends on the model of computation.

Comparisons counted as the expensive operation

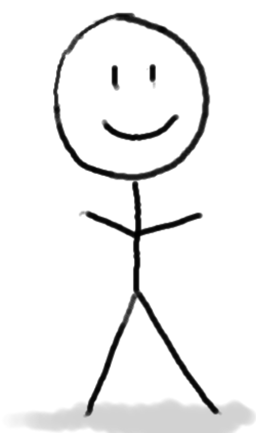
Who is bigger?

Credit to Mary Wooters for lower bound slides

Comparison-based sorting

Want to sort these items.
There's some ordering on them, but we don't know what it is.

Is  bigger than  ?



Algorithm

YES

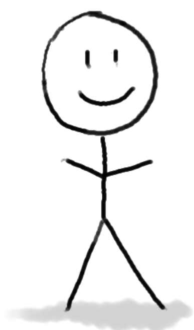
There is a genie who knows what the right order is.

The algorithm's job is to output a correctly sorted list of all the objects.

The genie can answer YES/NO questions of the form:
is [this] bigger than [that]?

Merge Sort and many other sorting algorithms work like this.

7	6	3	5	1	4	2



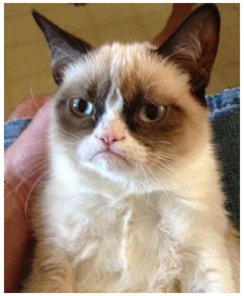
Is 7 bigger than 5 ? **YES**

Is 6 bigger than 5 ? **YES**

Is 3 bigger than 5 ? **NO**

5

etc.



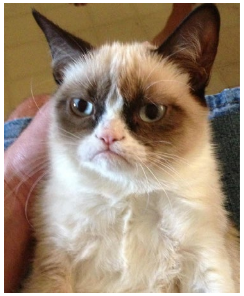
Lower bound of $\Omega(n \log(n))$.

- Theorem:
  - Any deterministic comparison-based sorting algorithm must take $\Omega(n \log(n))$ steps.

This covers all the sorting algorithms we know!!!

- How might we prove this?

  1. Consider all comparison-based algorithms, one-by-one, and analyze them.



Lower bound of $\Omega(n \log(n))$.

- Theorem:
  - Any deterministic comparison-based sorting algorithm must take $\Omega(n \log(n))$ steps.

This covers all the sorting algorithms we know!!!

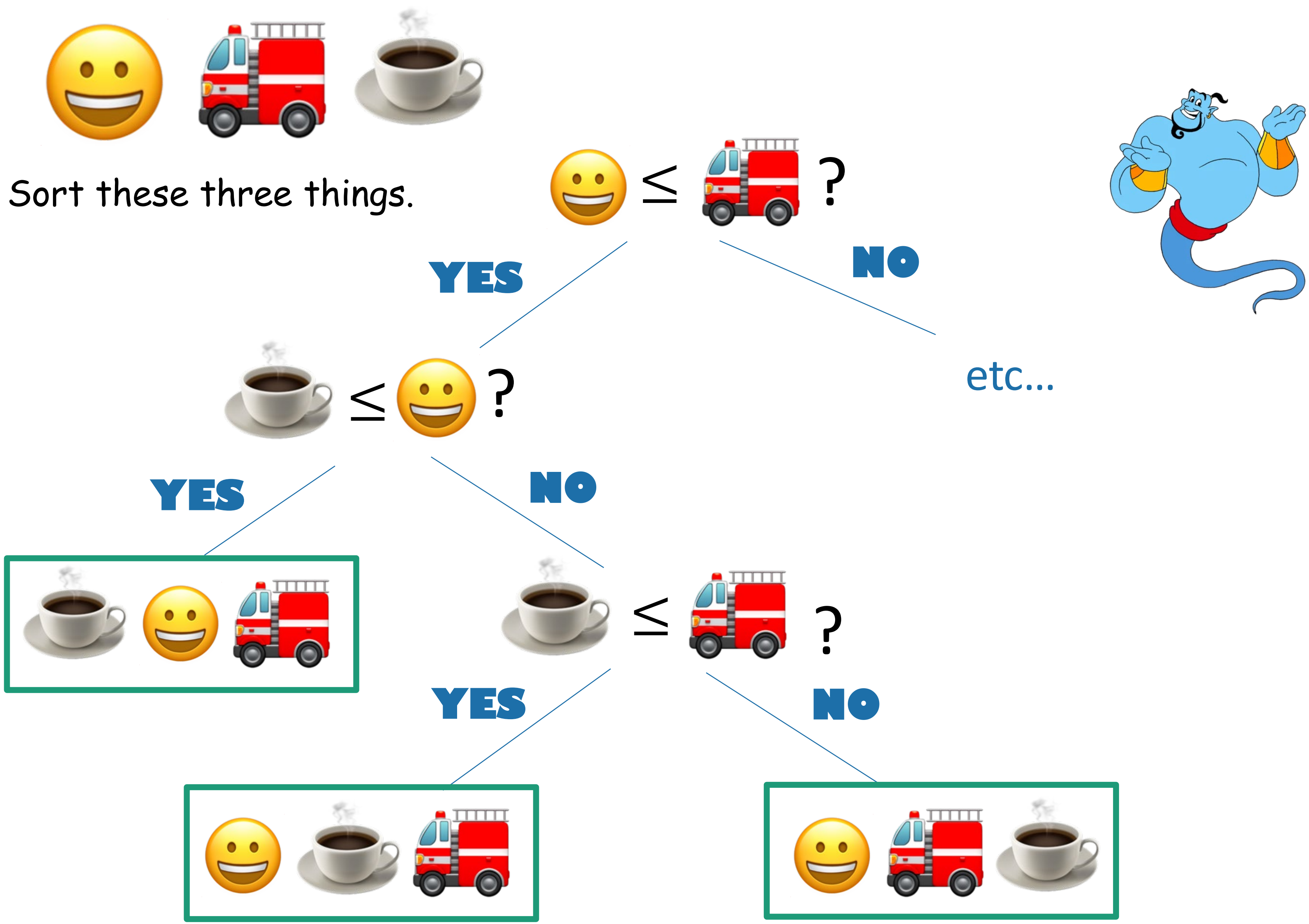
- How might we prove this?

1. Consider all comparison-based algorithms, one-by-one, and analyze them.
2. Don't do that.

Instead, argue that all comparison-based sorting algorithms give rise to a **decision tree**.

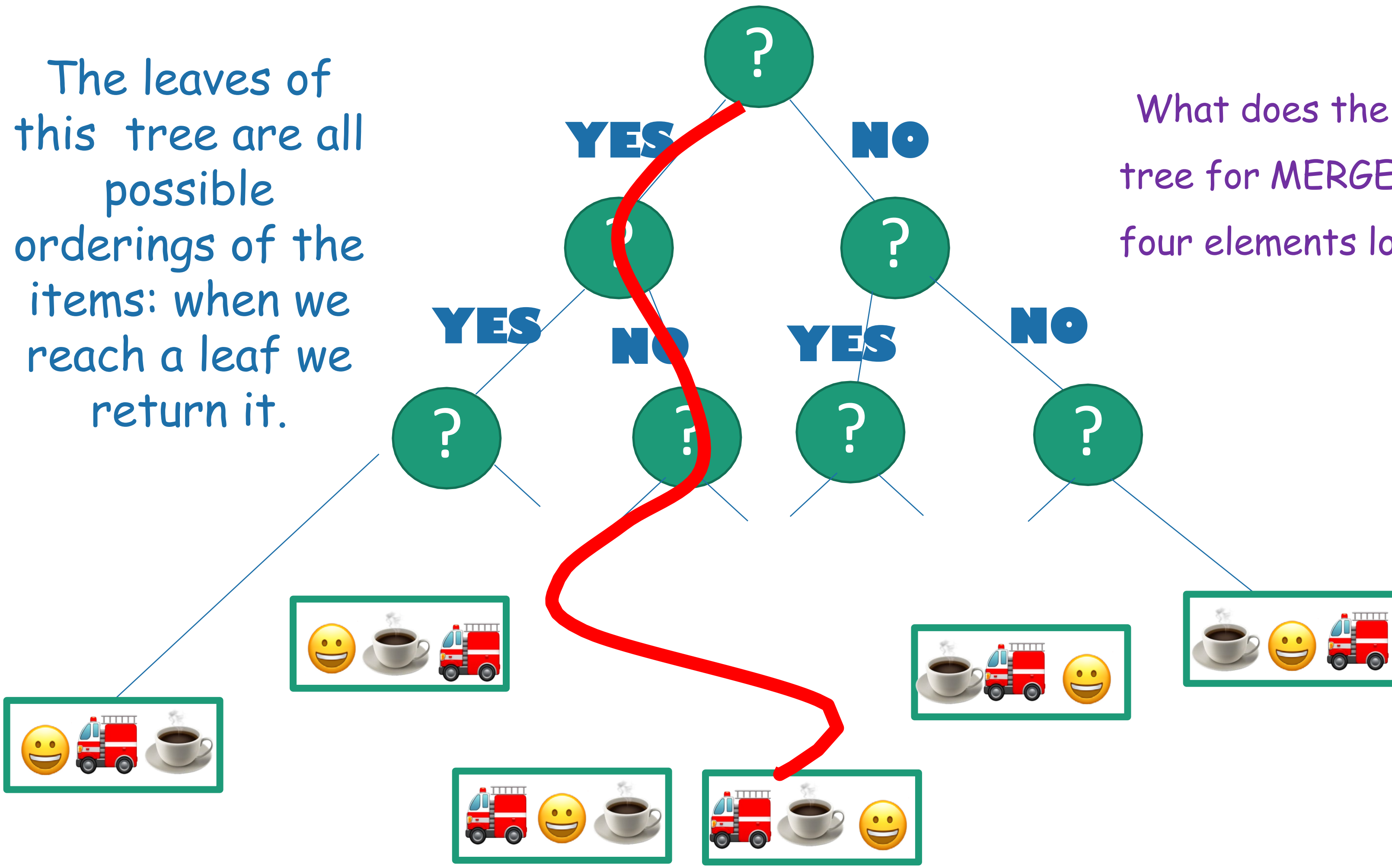
Then analyze decision trees.

Decision trees



All comparison-based algorithms have an associated decision tree.

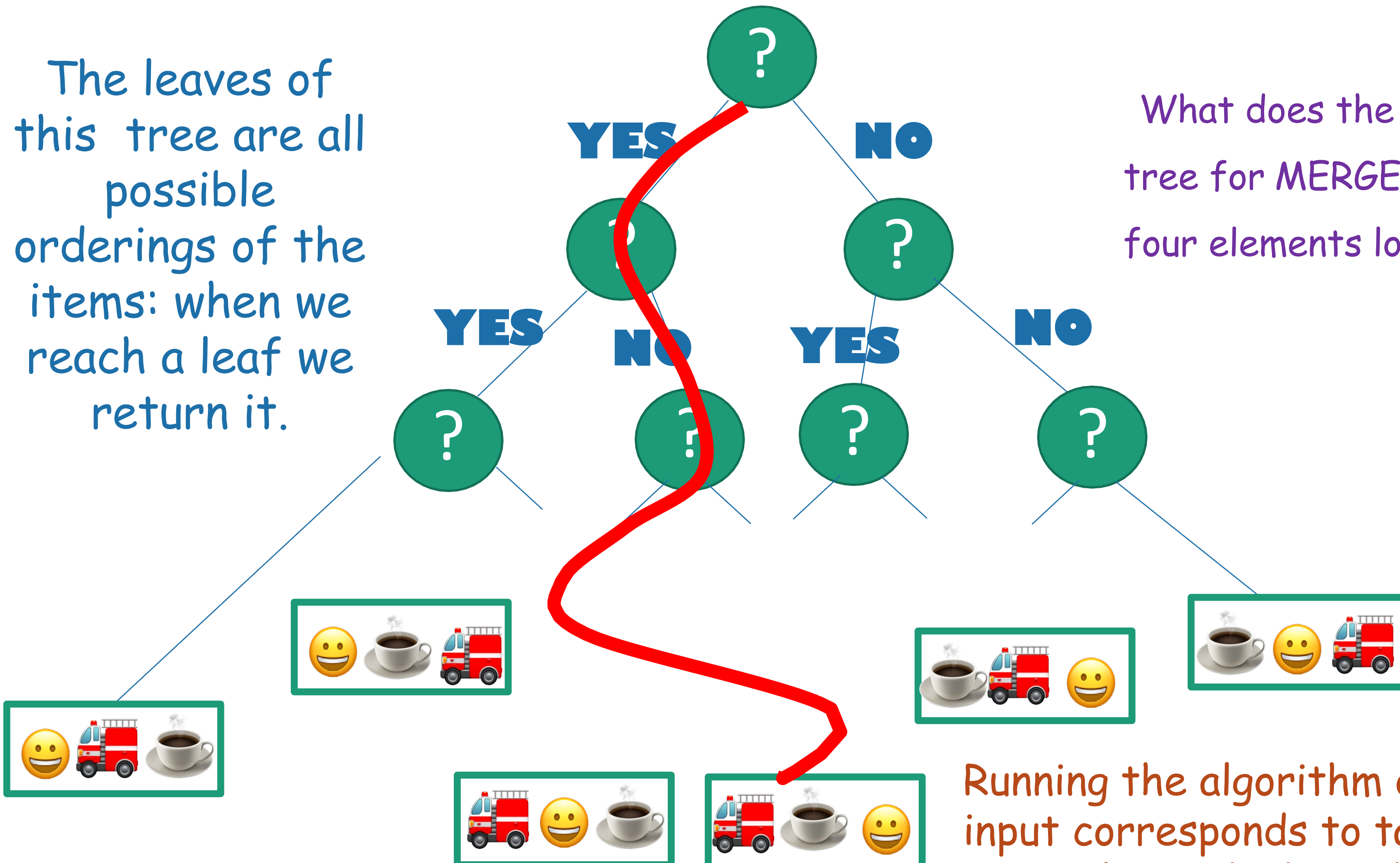
The leaves of this tree are all possible orderings of the items: when we reach a leaf we return it.



What does the decision tree for MERGE-SORT four elements look like?

All comparison-based algorithms have an associated decision tree.

The leaves of this tree are all possible orderings of the items: when we reach a leaf we return it.



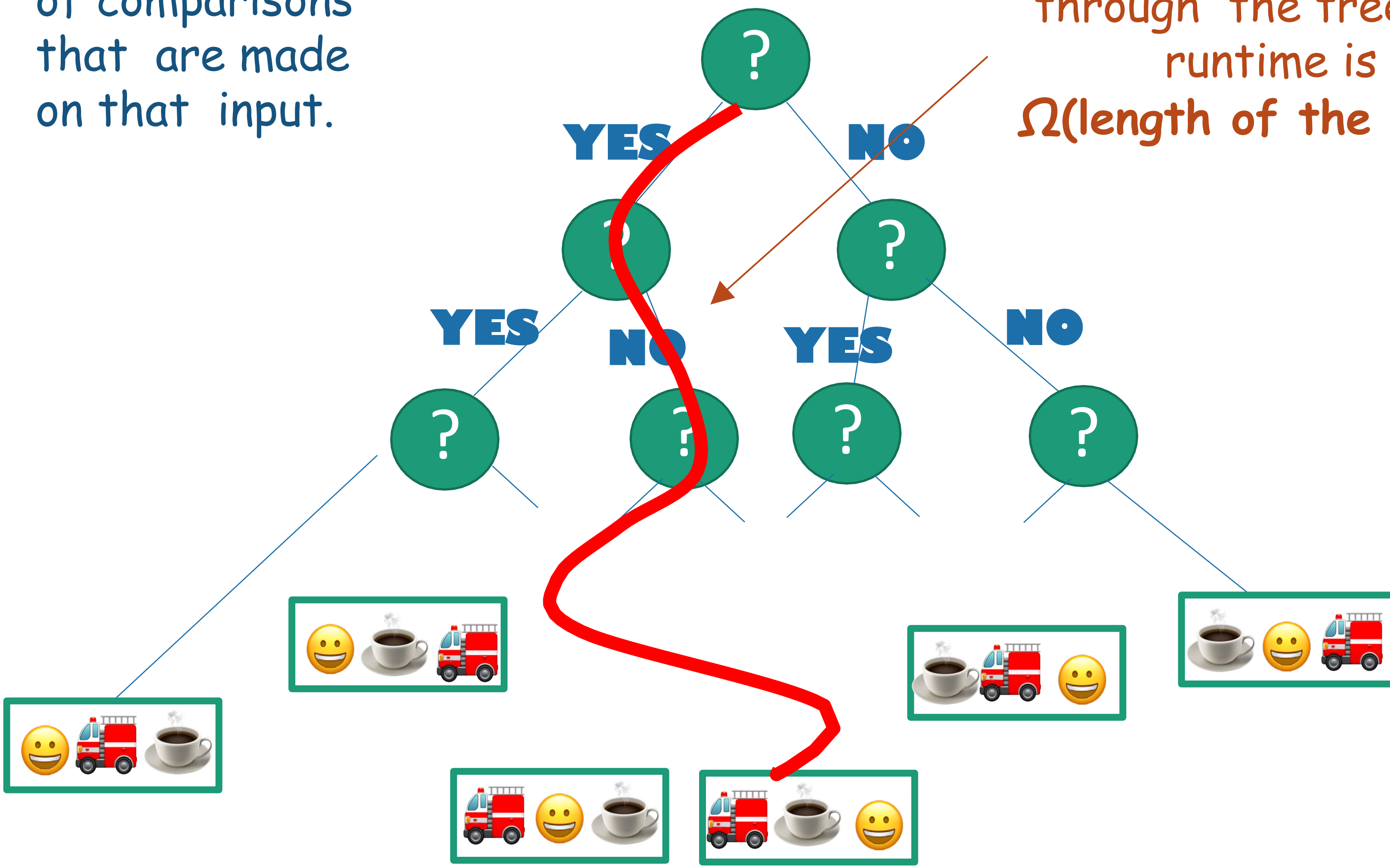
What does the decision tree for MERGE-SORT four elements look like?

Running the algorithm on a given input corresponds to taking a particular path through the tree.

What's the runtime on a particular input?

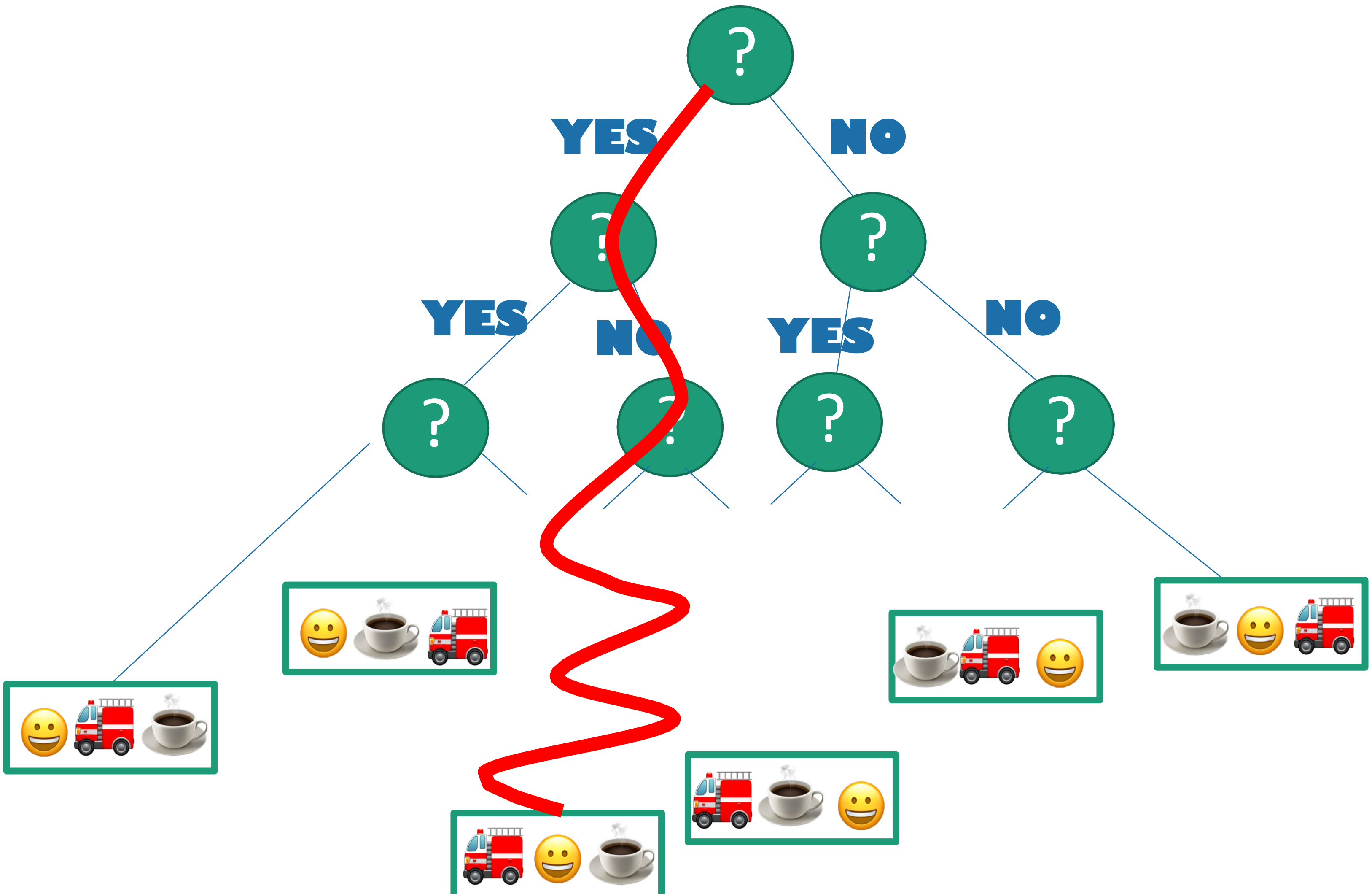
At least the number of comparisons that are made on that input.

If we take this path through the tree, the runtime is Ω(length of the path).



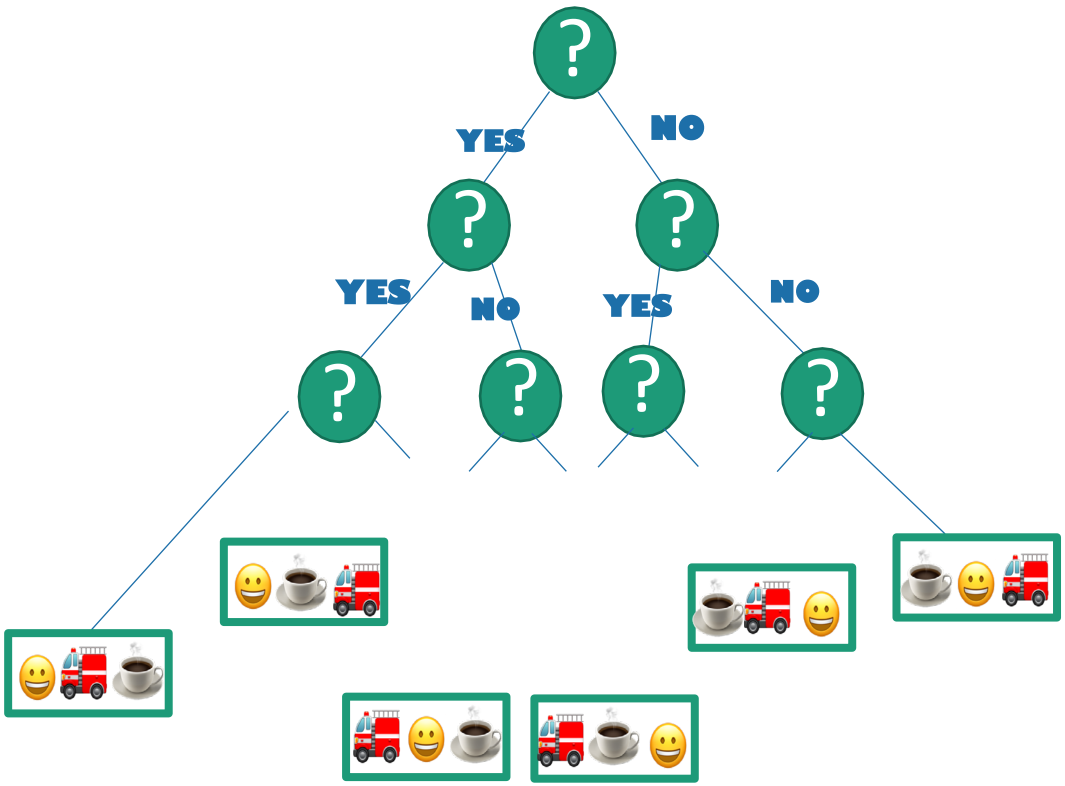
What's the worst-case runtime?

At least Ω(length of the longest path).



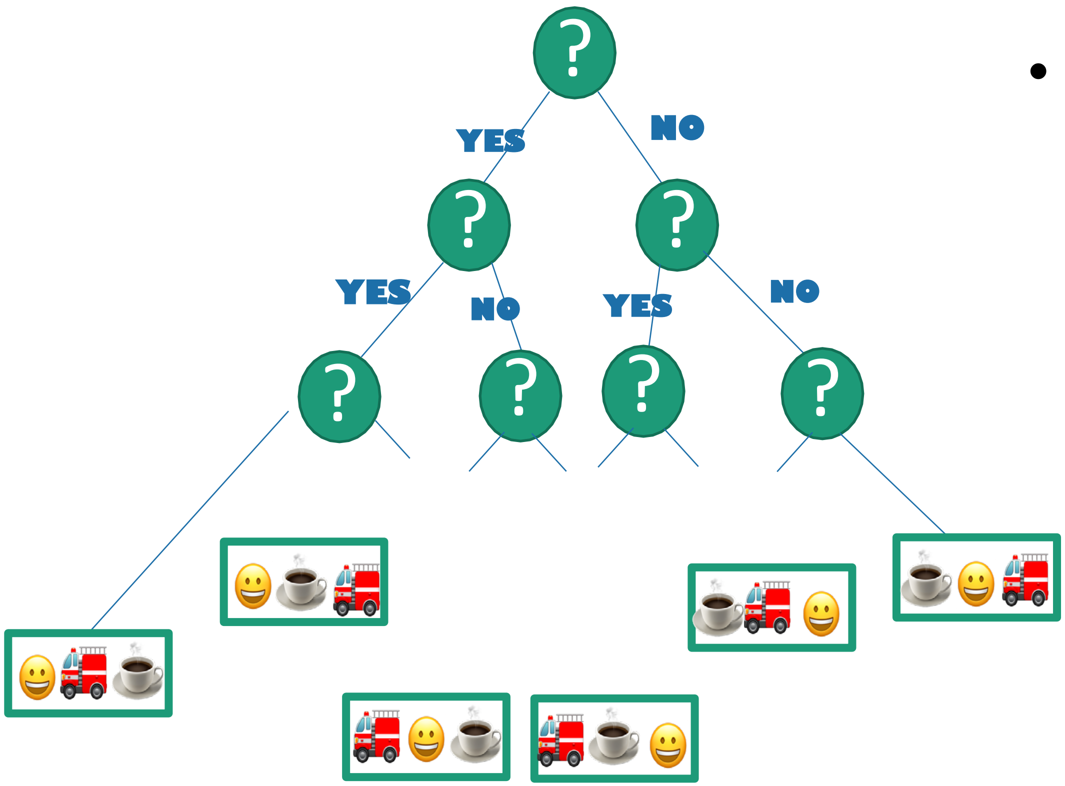
How long is the longest path?

We want a statement: in all such trees, the longest path is at least ______



How long is the longest path?

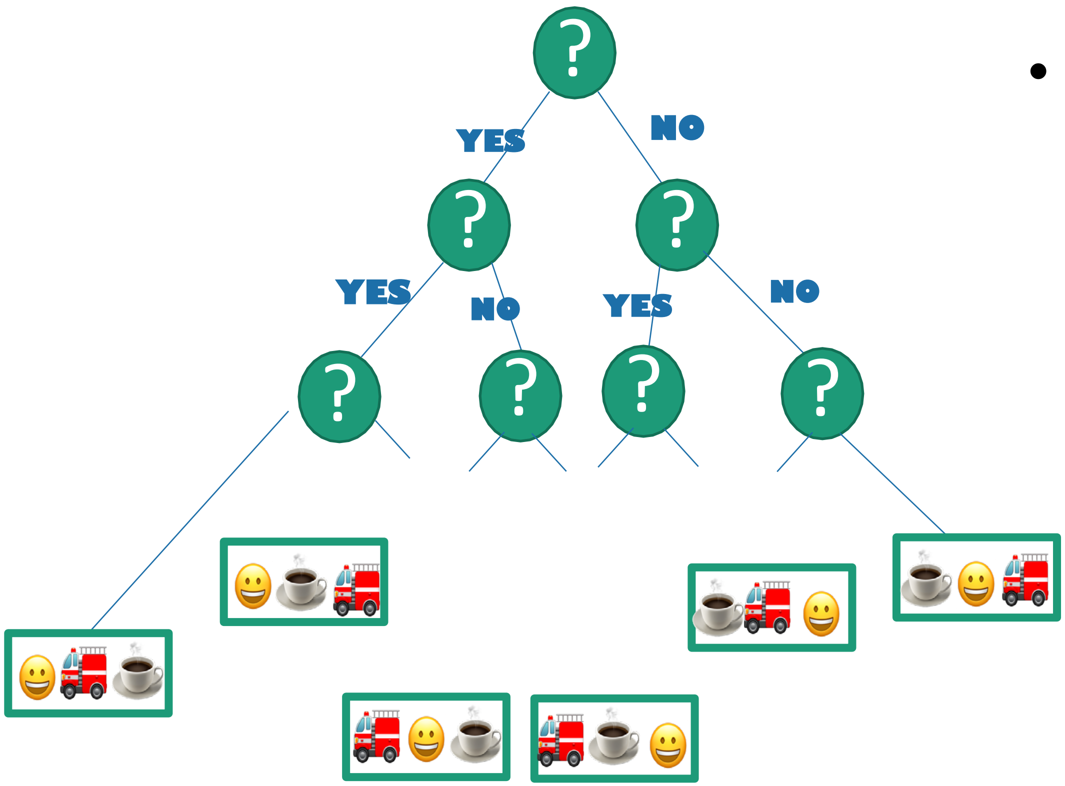
We want a statement: in all such trees, the longest path is at least ______



- This is a binary tree with at least______leaves.

How long is the longest path?

We want a statement: in all such trees, the longest path is at least ______

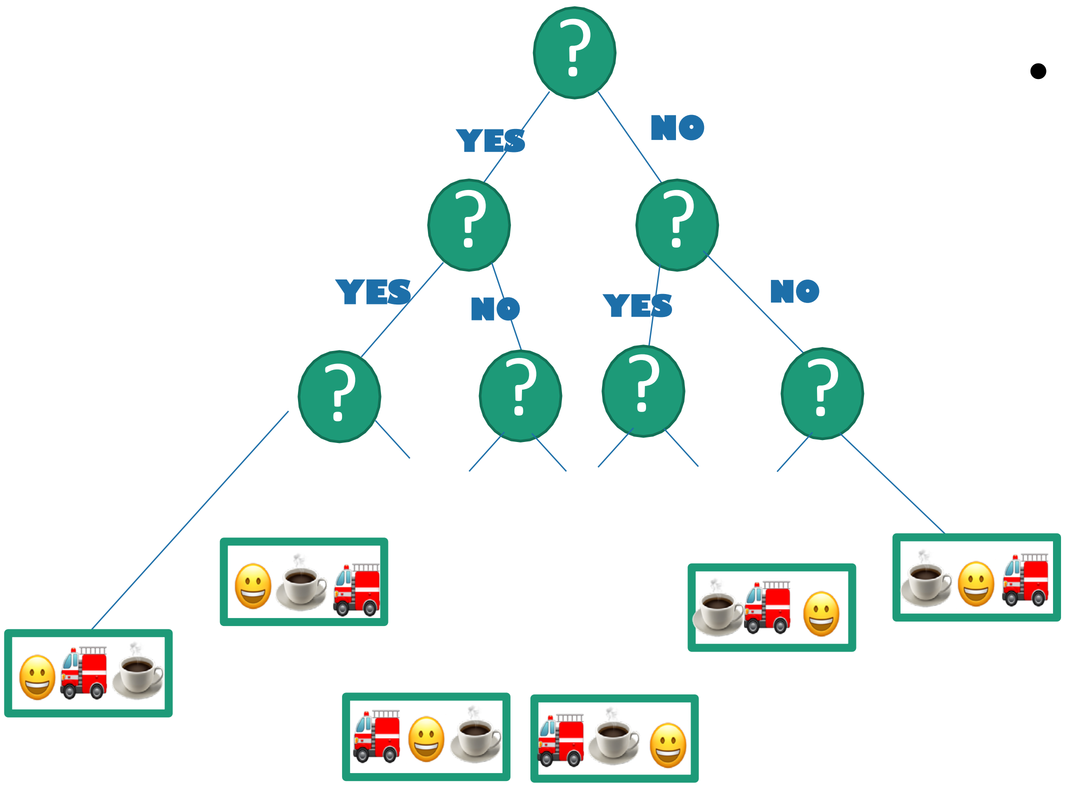


- This is a binary tree with at least__n!__leaves.

- n! is about $(n/e)^n$ (Stirling's formula).
- log(n!) can be approximated as … ?

How long is the longest path?

We want a statement: in all such trees, the longest path is at least ______

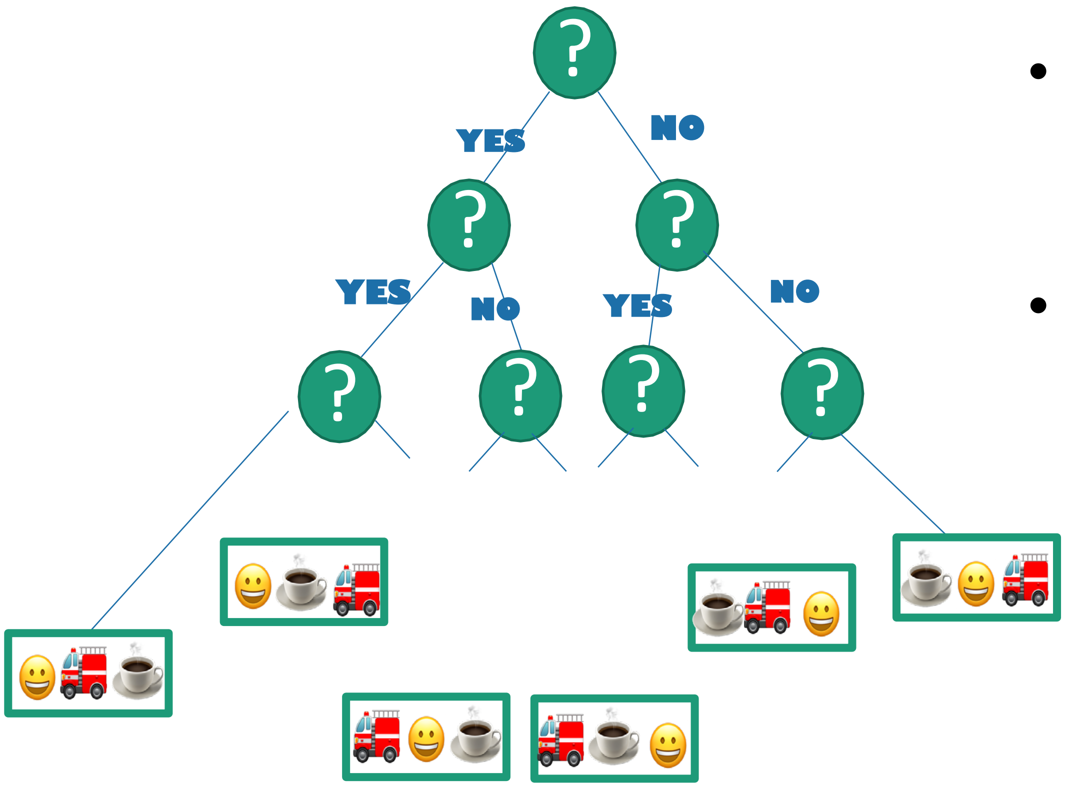


- This is a binary tree with at least ___n!___leaves.

- n! is about $(n/e)^n$ (Stirling's formula).
- log(n!) is about n log(n/e) = Ω(n log(n)).

How long is the longest path?

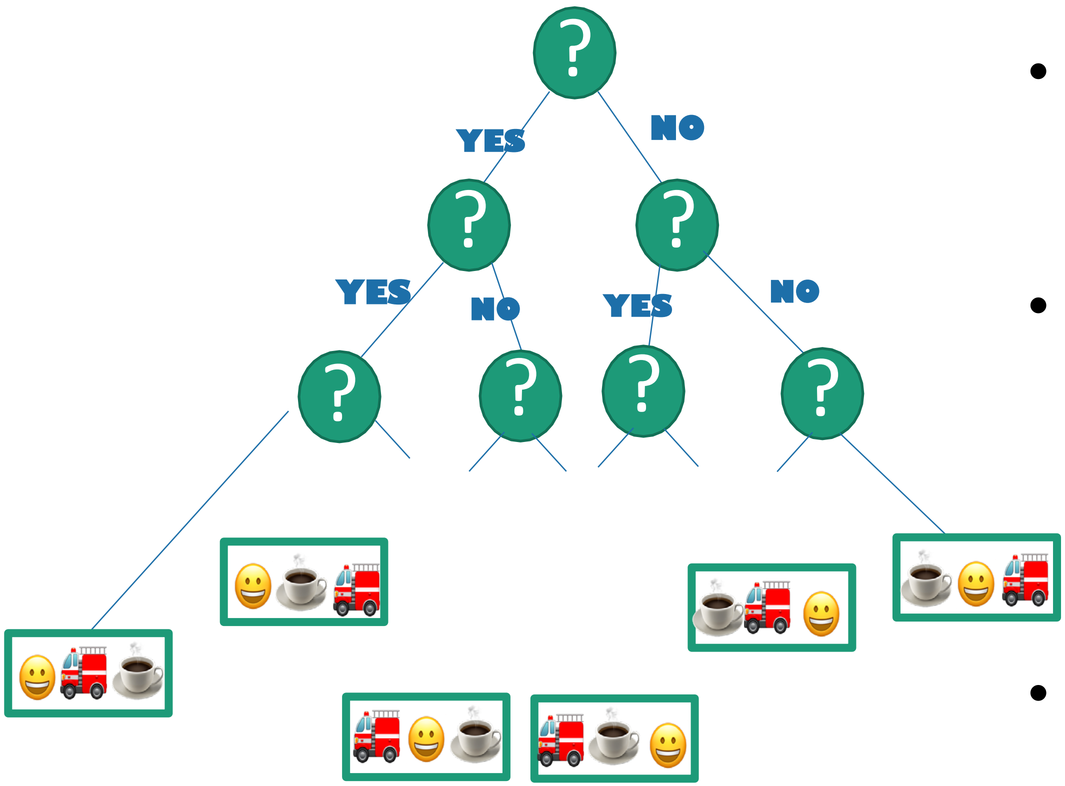
We want a statement: in all such trees, the longest path is at least ______



- This is a binary tree with at least___n!___leaves.
- The shallowest tree with n! leaves is the completely balanced one, which has depth ______

- n! is about $(n/e)^n$ (Stirling's formula).
- log(n!) is about n log(n/e) = Ω(n log(n)).

How long is the longest path?



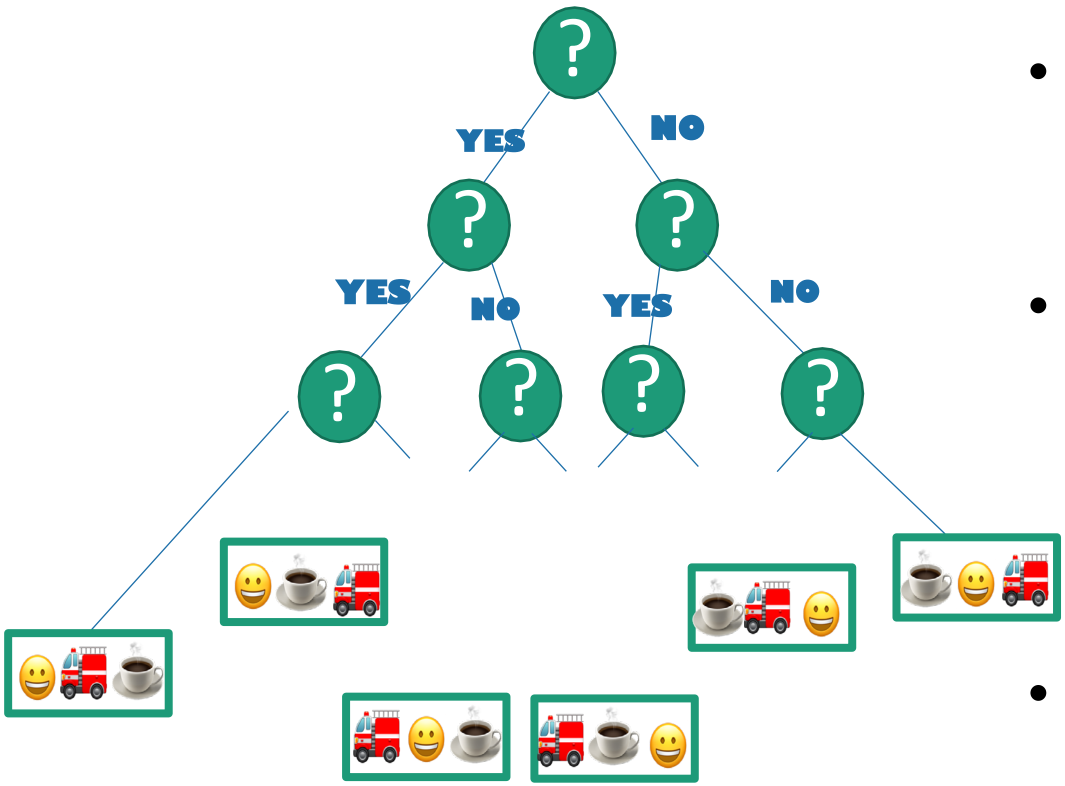
We want a statement: in all such trees, the longest path is at least ______

- This is a binary tree with at least __n!__leaves.
- The shallowest tree with n! leaves is the completely balanced one, which has depth _log(n!)_.
- So in all such trees, the longest path is at least ---------.

- n! is about $(n/e)^n$ (Stirling's formula).
- log(n!) is about n log(n/e) = Ω(n log(n)).

How long is the longest path?

We want a statement: in all such trees,
the longest path is at least ______

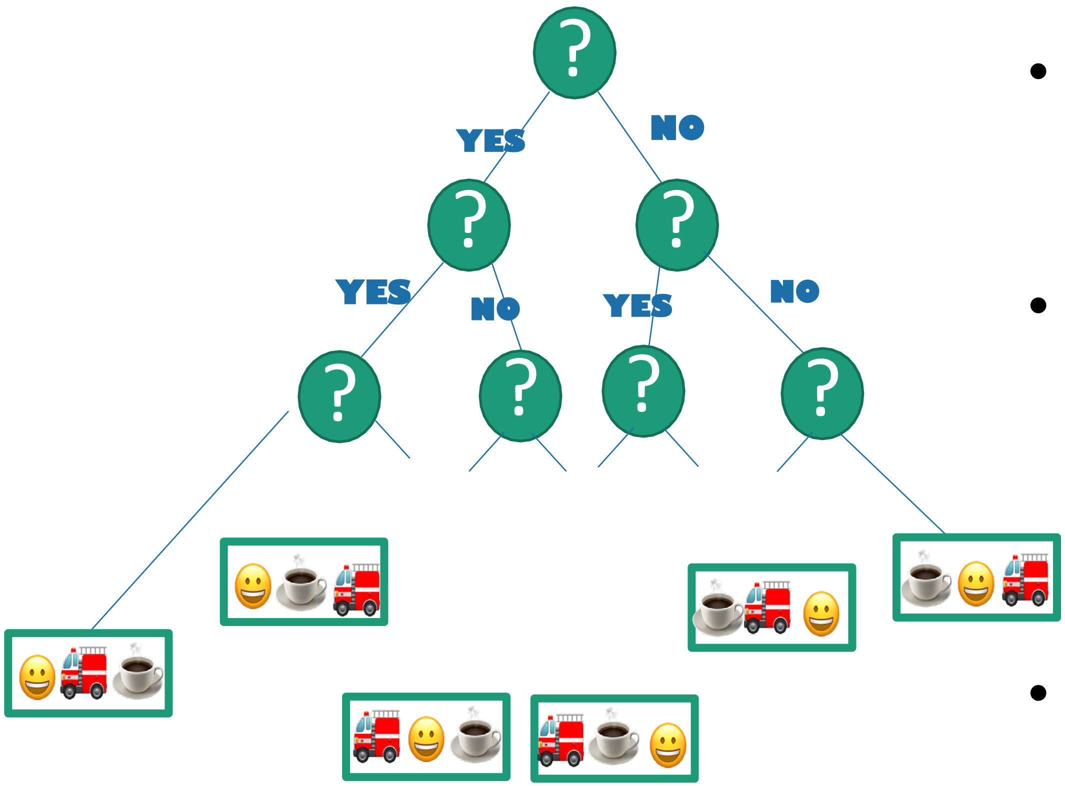


- This is a binary tree with at least__n!__leaves.
- The shallowest tree with n! leaves is the completely balanced one, which has depth _log(n!)_.
- So in all such trees, the longest path is at least log(n!).

- n! is about $(n/e)^n$ (Stirling's formula).
- log(n!) is about $n \log(n/e) = \Omega(n \log(n))$.

How long is the longest path?

We want a statement: in all such trees, the longest path is at least ______

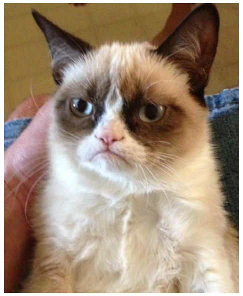


- This is a binary tree with at least ___n!___ leaves.
- The shallowest tree with n! leaves is the completely balanced one, which has depth __log(n!)__.
- So in all such trees, the longest path is at least log(n!).

- n! is about $(n/e)^n$ (Stirling's formula).
- log(n!) is about n log(n/e) = Ω(n log(n)).

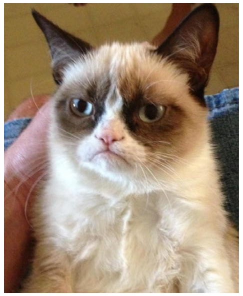
Conclusion: the longest path has length at least Ω(n log(n)).

Lower bound of $\Omega(n \log(n))$.



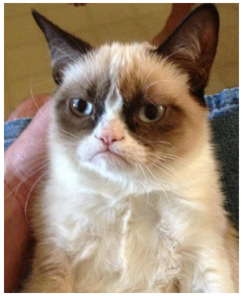
- Theorem:
  - Any deterministic comparison-based sorting algorithm must make $\Omega(n \log(n))$ comparisons.
- Proof?

Lower bound of $\Omega(n \log(n))$.



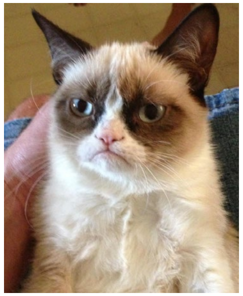
- Theorem:
  - Any deterministic comparison-based sorting algorithm must make $\Omega(n \log(n))$ comparisons.
- Proof:
  - Any deterministic comparison-based algorithm can be represented as a decision tree with n! leaves.

Lower bound of $\Omega(n \log(n))$.



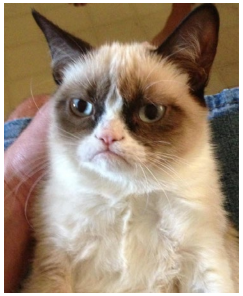
- Theorem:
  - Any deterministic comparison-based sorting algorithm must make $\Omega(n \log(n))$ comparisons.
- Proof:
  - Any deterministic comparison-based algorithm can be represented as a decision tree with $n!$ leaves.
  - The worst-case running time is at least the depth of the decision tree.

Lower bound of $\Omega(n \log(n))$.



- Theorem:
  - Any deterministic comparison-based sorting algorithm must make $\Omega(n \log(n))$ comparisons.
- Proof:
  - Any deterministic comparison-based algorithm can be represented as a decision tree with n! leaves.
  - The worst-case running time is at least the depth of the decision tree.
  - All decision trees with n! leaves have depth $\Omega(n \log(n))$.

Lower bound of $\Omega(n \log(n))$.



- Theorem:
  - Any deterministic comparison-based sorting algorithm must make $\Omega(n \log(n))$ comparisons.

- Proof:
  - Any deterministic comparison-based algorithm can be represented as a decision tree with n! leaves.
  - The worst-case running time is at least the depth of the decision tree.
  - All decision trees with n! leaves have depth $\Omega(n \log(n))$.
  - So any comparison-based sorting algorithm must have worst-case running time at least $\Omega(n \log(n))$.