Algorithm
Design and Analysis

Concrete Models and Lower Bounds

Roadmap for today

- Formal models of computation
- Finding the maximum element in an array
- A lower bound for sorting in the comparison model
- Checking whether an unknown graph is connected

Formal models of computation

- When theoretically analyzing algorithms, we don't consider their performance on a particular piece of hardware
  - E.g., how fast is this algorithm on an i9-13900K with DDR5 RAM? Who cares :)
- Instead, we define a **model of computation** which specifies:
  - Exactly what operations are permitted
  - How much each operation costs
- E.g., a *Turing Machine* is a model of computation
  - Allowed operations: Read/write/move tape
  - Cost model: all operations cost 1

What is the best model?

- No such thing… **it depends**
- It depends on the setting. Are you designing a single-threaded algorithm, a parallel algorithm, an algorithm for GPUs, an algorithm that will work on a gigantic dataset…
- It also depends on your goal. Are you trying to predict the performance of an algorithm in a particular scenario or are you trying to prove a ***lower bound***?

Examples of models

- Unit-cost (word)-RAM (next lecture!)
  - Constant-time addressable memory cells consisting of w-bit integers ($w \geq \log n$)
  - Reading/writing, arithmetic, logic, bitwise operations take constant time

- External Memory Model (not in this class)
  - Machine has M cells of "fast memory", and unlimited "slow memory"
  - Loading/storing B cells of data from/to slow memory to/from fast memory costs 1
  - All computation on data in the fast memory is free!

- Massively Parallel Computation (not in this class)
  - Input consists of N elements split across M machines each with enough space for S elements
  - The computation happens in *rounds*, in which each machine can do computation for free!
  - After each round, machines send messages to each other. <u>The cost is the number of rounds.</u>

Today's models

- ***The Comparison Model*** (as seen in Lecture 1)
  - Input to the algorithm consists of an array of n items in some order
  - The algorithm may perform comparisons (is $a_i < a_j$?) at a cost of 1
  - Copying/moving items is *free*
  - The items **can not** be assumed to be integers, or any specific type
- ***The edge-query model***
  - The input to the problem is a graph G, ***but*** the algorithm can't see it
  - The algorithm can only ask questions "does edge (u, v) exist", costing 1
  - All other computation is free. The goal is to determine whether G has some desired property (e.g., is it connected)

Today's goals

- Analyze algorithms in these concrete models, avoiding asymptotic notation, when possible, i.e., get the tightest bounds we can.
- Devise ***lower bounds*** for problems, i.e., prove that certain problems can not be solved in under a certain cost.

If we say that a specific problem on inputs of size n has a *lower bound* of $g(n)$, we mean that for **any algorithm** **A**, **there exists some input** of size n for which the cost of **A** is **at least** $g(n)$.

Select-max

Problem: Given an array of n elements, return the maximum element.

Algorithm: Scan left-to-right keeping track of the maximum so far

Cost: $n - 1$ comparisons

Question: How few comparisons could any algorithm possibly do? Is it possible to do fewer than $n - 1$?

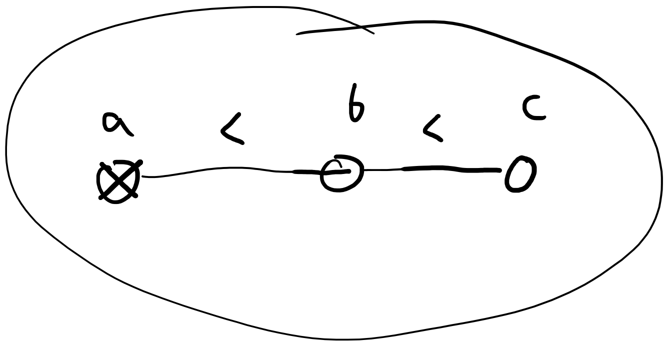
Weak lower bound

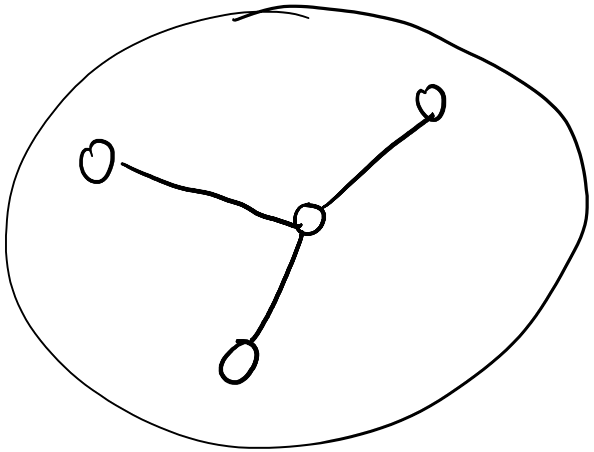
Theorem: Any deterministic algorithm for select-max costs **at least** $n/2$ comparisons

Must look at every element

Stronger lower bound

Theorem: Any deterministic algorithm for select-max costs **at least** $n - 1$ comparisons





Adversary arguments

- We proved the lower bound using an ***adversary argument***
- Given any algorithm that performs “too few” comparisons, we argued that we can always construct an input on which it must give the wrong answer.
- We are playing the role of an *adversary* trying to “break” the algorithm!
- Remember that our argument must break ***every algorithm*** that we are trying to rule out, we can not assume a specific algorithm.

Select second-max

Problem: Given an array of n elements, return the largest **and** second-largest element.

Theorem: Any deterministic algorithm for select-second-max costs **at least** $n - 1$ comparisons

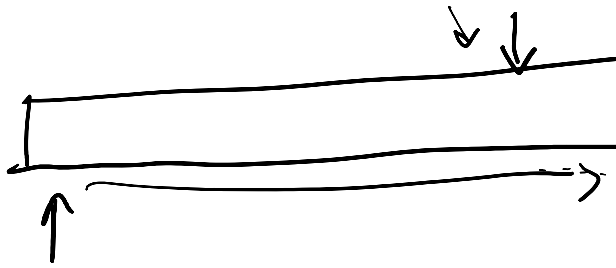
Proof. Same as select-max

Algorithm: Find the maximum, then scan again to find the second

$$2n-3 = n-1 + n-2$$

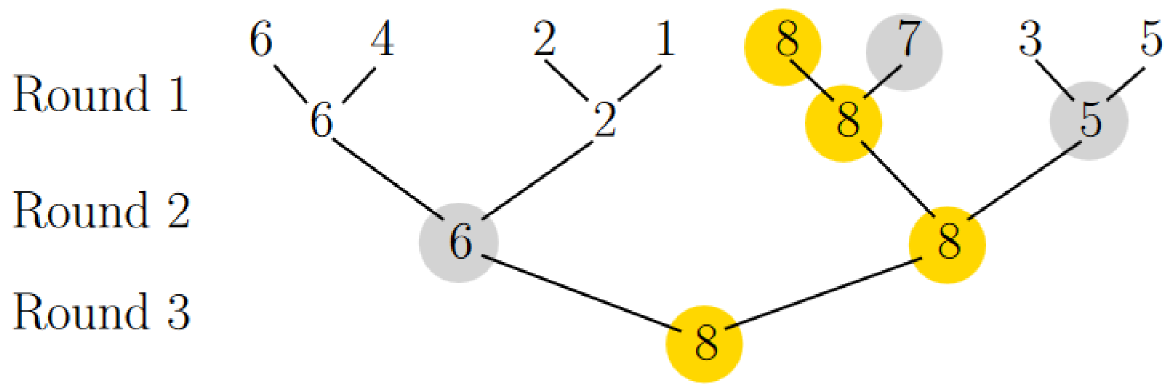
Faster algorithm

Property: In any correct algorithm for select-max, the second-largest element **must** have been compared to the largest element



A tournament algorithm

Algorithm: Compare pairs of elements, then compare the winners, and so on



Theorem: The tournament algorithm costs at most $n + \log_2 n - 2$

$n-1 \quad + \lg n - 1$

A tight lower bound

Bonus Theorem: Any deterministic algorithm for select-second-max makes at least $n + \log_2 n - 2$ comparisons

Proof too long, so we won't do it for now :)

Sorting in the comparison model

- The comparison model is widely used to analyze sorting algorithms
  - You don’t get to assume that the data are integers, or numbers, so the algorithms will be extremely general. They can sort anything!
  - The number of comparisons performed by a sorting algorithm **usually** (but not always) matches the asymptotic number of instructions performed in a more robust model like the word-RAM, so this measurement is a reasonable proxy for performance.

- We know how to achieve $O(n \log n)$ comparisons: Quicksort, Mergesort, Heapsort. Can we do better?

Setup for comparison sorting

- The input to the problem is n elements in some initial order.
- The algorithm knows nothing about the elements
- The algorithm may compare two elements (is $a_i > a_j$?) at a cost of 1
- Moving/copying/swapping items is free!
- Assume that the input contains no duplicates

Warning!: Defining the "output" of a comparison sort is extremely subtle if we want to correctly prove lower bounds. We must be very careful.

Input/output of comparison sorting

- The *input* is an array of elements in some initial order

$$a_1, a_2, a_3, \dots, a_n$$

- The *output* is a permutation $\pi(1), \dots, \pi(n)$ of the input elements that sorts the input

$$a_{\pi(1)} < a_{\pi(2)} < \cdots < a_{\pi(n)}$$

Note: The "output" of a comparison sorting algorithm **is the permutation** that it applies to the input. For example, if we sort $[c, a, b, d]$ and $[b, d, a, c]$, both output $[a, b, c, d]$ but this **is not** "the same output" because the comparison algorithm can not read the elements. All it knows is that the first was sorted by the permutation [2,3,1,4] and the second was sorted by [3,1,4,2]

Sorting lower bound

Theorem: Any deterministic comparison sorting algorithm must perform at least $\log_2(n!)$ comparisons in the worst case.

- Different technique this time. Instead of an adversary, we are going to use "information theory"
- This relies vary carefully on how we define the input/output
- Remember that we must prove this fact for **every possible algorithm**, not just one.

Proof

- Consider the minimum number of distinct outputs that a sorting algorithm must be able to produce to be able to sort any possible input of length n

$$M = n!$$

- Why minimum? We don't want redundant outputs that don't allow us to solve more inputs.

- In other words, the algorithm must be capable of outputting at least M different permutations, or there would exist some input that it was not capable of sorting.

Proof continued…

- We know that any correct algorithm must be capable of outputting at least $n!$ distinct permutations, or there would exist some input that it was not capable of sorting.

- Remember, the algorithm is deterministic! Its behaviour is determined **entirely** by the results of the comparisons.

- If a deterministic algorithm makes c comparisons, how many outputs distinct outputs can it possibly produce?

2^c

Proof completing

- So, if an algorithm performs c comparisons, it can produce at most 2^c different possible outputs
- To be correct, an algorithm needs to be able to produce at least $n!$ different outputs, or there is some input that it can't solve.
- Therefore

$$2^c \geq n!$$

$$c \geq \log_2 n!$$

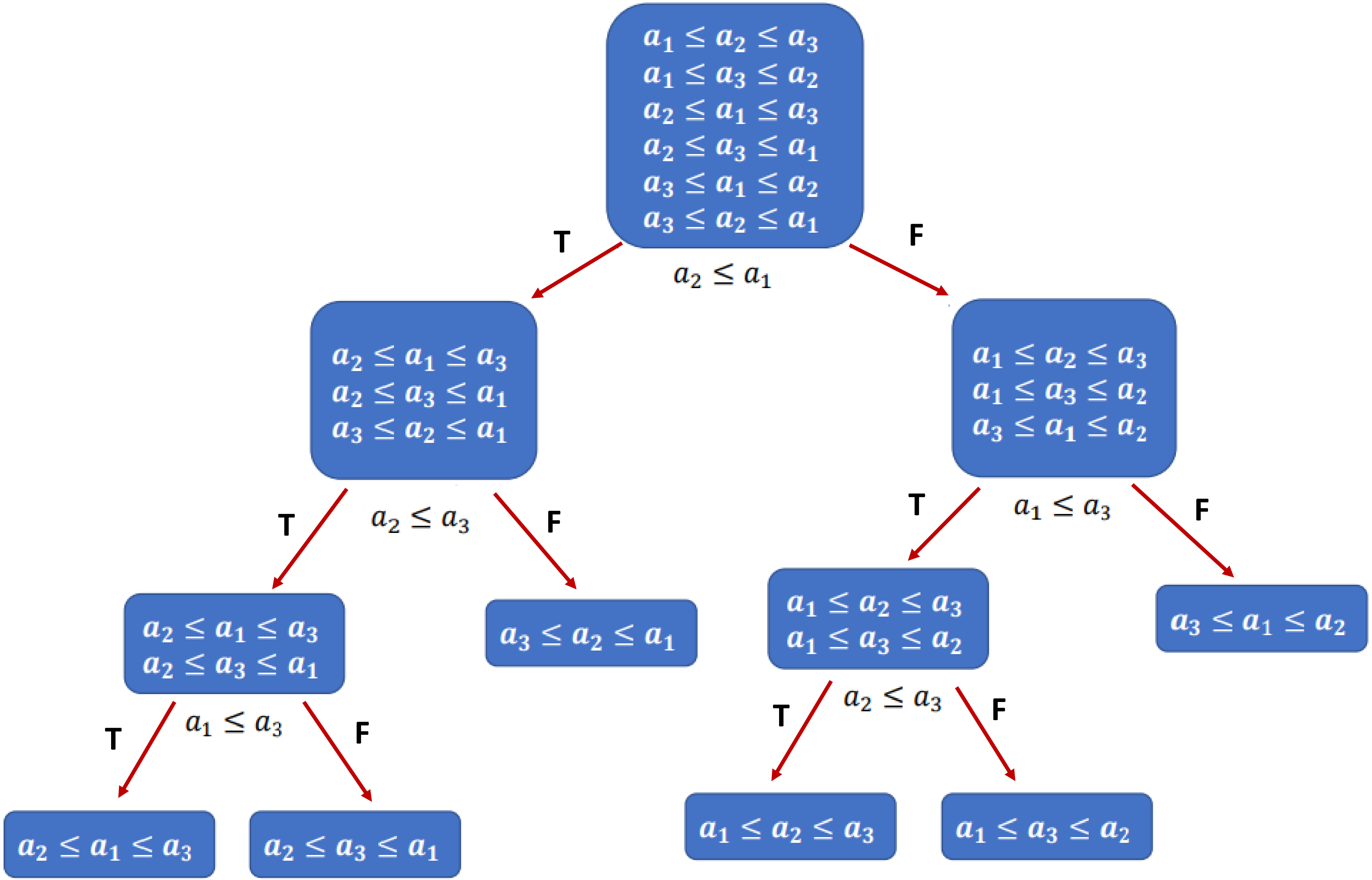
An alternative view: Decision Trees

- Consider the set of all possible outputs. Before the algorithm makes any comparisons, they all could be the answer.

- After each comparison, some of the possibilities are ruled out

$a_1 < a_2 < a_3$		$a_1 < a_2 < a_3$
$a_1 < a_3 < a_2$	$(a_1 < a_3)$ == TRUE	$a_1 < a_3 < a_2$
$a_3 < a_1 < a_2$		$\cancel{a_3 < a_1 < a_2}$

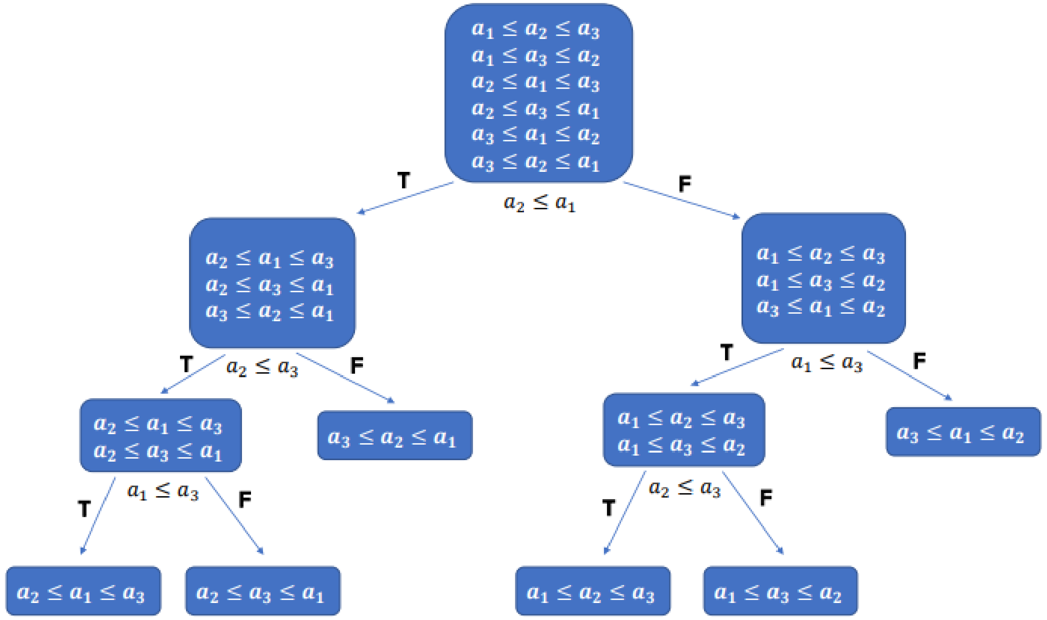
- We can represent any specific comparison-based algorithm as a ***decision tree***.



$a_1 \le a_2 \le a_3$
$a_1 \le a_3 \le a_2$
$a_2 \le a_1 \le a_3$
$a_2 \le a_3 \le a_1$
$a_3 \le a_1 \le a_2$
$a_3 \le a_2 \le a_1$
$a_2 \le a_1$
T
F
$a_2 \le a_1 \le a_3$
$a_2 \le a_3 \le a_1$
$a_3 \le a_2 \le a_1$
$a_2 \le a_3$
T
F
$a_1 \le a_2 \le a_3$
$a_1 \le a_3 \le a_2$
$a_3 \le a_1 \le a_2$
$a_1 \le a_3$
T
F
$a_2 \le a_1 \le a_3$
$a_2 \le a_3 \le a_1$
$a_1 \le a_3$
T
F
$a_3 \le a_2 \le a_1$
$a_1 \le a_2 \le a_3$
$a_1 \le a_3 \le a_2$
$a_2 \le a_3$
T
F
$a_3 \le a_1 \le a_2$
$a_2 \le a_1 \le a_3$
$a_2 \le a_3 \le a_1$
$a_1 \le a_2 \le a_3$
$a_1 \le a_3 \le a_2$

Properties of this tree

- The root node contains $n!$ elements, one for each possible output
- Each internal node corresponds to a comparison
- Leaf nodes contain a unique output
- There are $n!$ leaf nodes



worst-case #comparisons = height

Alternative proof of the lower bound

Consider a decision tree for **any comparison-based sorting algorithm**

> **Remember**: A decision tree corresponds to a **specific algorithm**, and we want to prove a lower bound for **all algorithms**, so our proof must consider **any valid possible decision tree** for the problem

$n!$ leaves height h

$$2^h \geq n!$$

$$h \geq \log_2 n!$$

What does $\log_2 n!$ look like?

$$\log_2 n! = \log_2 n + \log_2(n-1) + \cdots + \log_2(1)$$

What does $\log_2 n!$ look like?

Stirling's approximation gives an even tighter bound!

$$\log_2(n!) = n\log_2 n - n\log_2 e + O(\log_2 n)$$

The fact that $\left(\frac{n}{e}\right)^n \leq n! \leq n^n$ will often be useful (see recitation!)

Lower bound techniques so far

- ***Adversary***: Show that you can construct an input to "break" the algorithm if it performs too few comparisons

- ***Information-theoretic***: Count the minimum number of necessary distinct outputs that the algorithm must be able to produce

- ***Decision Tree***: Model any algorithm for the problem as a binary tree of possible outputs and lower bound the height of the tree

Query models and evasiveness

- Let G be an undirected n vertex graph
- We want to test whether this graph has a certain property
- A property can be any true/false question about the graph:
  - Is it connected?
  - Is it bipartite?
  - Does it contain any cycles?
- However, we do not know the graph! We can pay a cost of 1 to ask a query: Does the edge (u, v) exist?
- The algorithm of course wants to minimize the cost

Checking connectivity

Theorem: Checking whether a graph is connected takes $\binom{n}{2}$ queries.

Proof. Query every possible edge then check if G is connected.

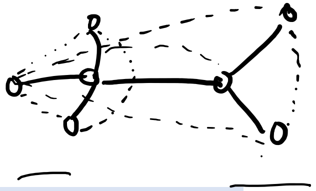
Definition: A property is called ***evasive*** if every algorithm to check it requires $\binom{n}{2}$ queries in the worst case.

Theorem: Connectivity is an evasive property.

Proof strategy. **Adversary.** We will describe an “evader”, an adversary that answers the queries in such a way that the querier can not figure out the answer until the very last edge.

Proof

- The goal of the evader is to force the querier to ask $\binom{n}{2}$ queries
- **Key Idea**: Only ever answer yes if saying no would confirm that the graph is disconnected.



Invariant:

- At any point, the edges that have been revealed to exist form a forest of trees.
  - For each such tree T, the querier has queried **every pair** of vertices in T
- For each pair of trees T and T', there exists $x \in T, y \in T'$ such that (x, y) has not been queried.

Proof continued…

- When the querier asks about edge (u, v), where u and v are in different trees T_u and T_v:
  - If for every pair of vertices $x \in T_u, y \in T_v (x, y)$, every edge $(x, y) \neq (u, v)$ has already been queried: answer YES
  - Otherwise, answer NO

Summary of today

- Formal models of computation allow us to prove ***lower bounds*** on the complexity of algorithms
- E.g., sorting can not be done in fewer than $\log_2 n! = \Theta(n \log n)$ comparisons.
- Techniques include:
  - ***Adversaries***
  - ***Information theoretic***
  - ***Decision tree***