CS 350 Algorithms and Complexity

Winter 2019

Lecture 15: Limitations of Algorithmic Power

Introduction to complexity theory

Andrew P. Black

Department of Computer Science

Portland State University

Lower Bounds

- Lower bound: an estimate of the *minimum* amount of work needed to solve a given problem
- Examples:
  - number of comparisons needed to find the largest element in a set of n numbers
  - number of comparisons needed to sort an array of size n
  - number of comparisons necessary for searching in a sorted array of size n
  - number of multiplications needed to multiply two $n \times n$ matrices

Lower Bounds (cont.)

- Lower bound can be
  - an exact count
  - an efficiency class (Ω)

- Lower bound is *tight* ≜ *there exists* an algorithm with the efficiency of the lower bound

Problem	Lower bound	Tight?
sorting	$\Omega(n \log n)$	
searching in a sorted array	$\Omega(\log n)$	
element uniqueness	$\Omega(n \log n)$	
n-digit integer multiplication	$\Omega(n)$	
multiplication of n x n matrices	$\Omega(n^2)$	

Lower Bounds (cont.)

- Lower bound can be
  - an exact count
  - an efficiency class (Ω)

- Lower bound is *tight* $\triangleq$ *there exists* an algorithm with the efficiency of the lower bound

Problem	Lower bound	Tight?
sorting	$\Omega(n \log n)$	yes
searching in a sorted array	$\Omega(\log n)$	
element uniqueness	$\Omega(n \log n)$	
n-digit integer multiplication	$\Omega(n)$	
multiplication of n x n matrices	$\Omega(n^2)$	

Lower Bounds (cont.)

- Lower bound can be
  - an exact count
  - an efficiency class (Ω)

- Lower bound is *tight* $\triangleq$ *there exists* an algorithm with the efficiency of the lower bound

Problem	Lower bound	Tight?
sorting	$\Omega(n \log n)$	yes
searching in a sorted array	$\Omega(\log n)$	yes
element uniqueness	$\Omega(n \log n)$	
n-digit integer multiplication	$\Omega(n)$	
multiplication of n x n matrices	$\Omega(n^2)$	

Lower Bounds (cont.)

- Lower bound can be
  - an exact count
  - an efficiency class (Ω)

- Lower bound is *tight* ≜ *there exists* an algorithm with the efficiency of the lower bound

Problem	Lower bound	Tight?
sorting	$\Omega(n \log n)$	yes
searching in a sorted array	$\Omega(\log n)$	yes
element uniqueness	$\Omega(n \log n)$	yes
n-digit integer multiplication	$\Omega(n)$	
multiplication of n x n matrices	$\Omega(n^2)$	

Lower Bounds (cont.)

- Lower bound can be
  - an exact count
  - an efficiency class (Ω)

- Lower bound is *tight* $\triangleq$ *there exists* an algorithm with the efficiency of the lower bound

Problem	Lower bound	Tight?
sorting	$\Omega(n \log n)$	yes
searching in a sorted array	$\Omega(\log n)$	yes
element uniqueness	$\Omega(n \log n)$	yes
n-digit integer multiplication	$\Omega(n)$	unknown
multiplication of n x n matrices	$\Omega(n^2)$	

Lower Bounds (cont.)

- Lower bound can be
  - an exact count
  - an efficiency class (Ω)

- Lower bound is *tight* $\triangleq$ *there exists* an algorithm with the efficiency of the lower bound

Problem	Lower bound	Tight?
sorting	$\Omega(n \log n)$	yes
searching in a sorted array	$\Omega(\log n)$	yes
element uniqueness	$\Omega(n \log n)$	yes
n-digit integer multiplication	$\Omega(n)$	unknown
multiplication of n x n matrices	$\Omega(n^2)$	unknown

Methods for Establishing Lower Bounds

- trivial lower bounds
  - based on data input & output
- information-theoretic arguments
  - e.g., decision trees
- adversary arguments
- problem reduction

Trivial Lower Bounds

based on counting the number of items that must be processed in input and generated as output

Examples:

- finding max element
- polynomial evaluation
- sorting
- element uniqueness
- Hamiltonian circuit existence

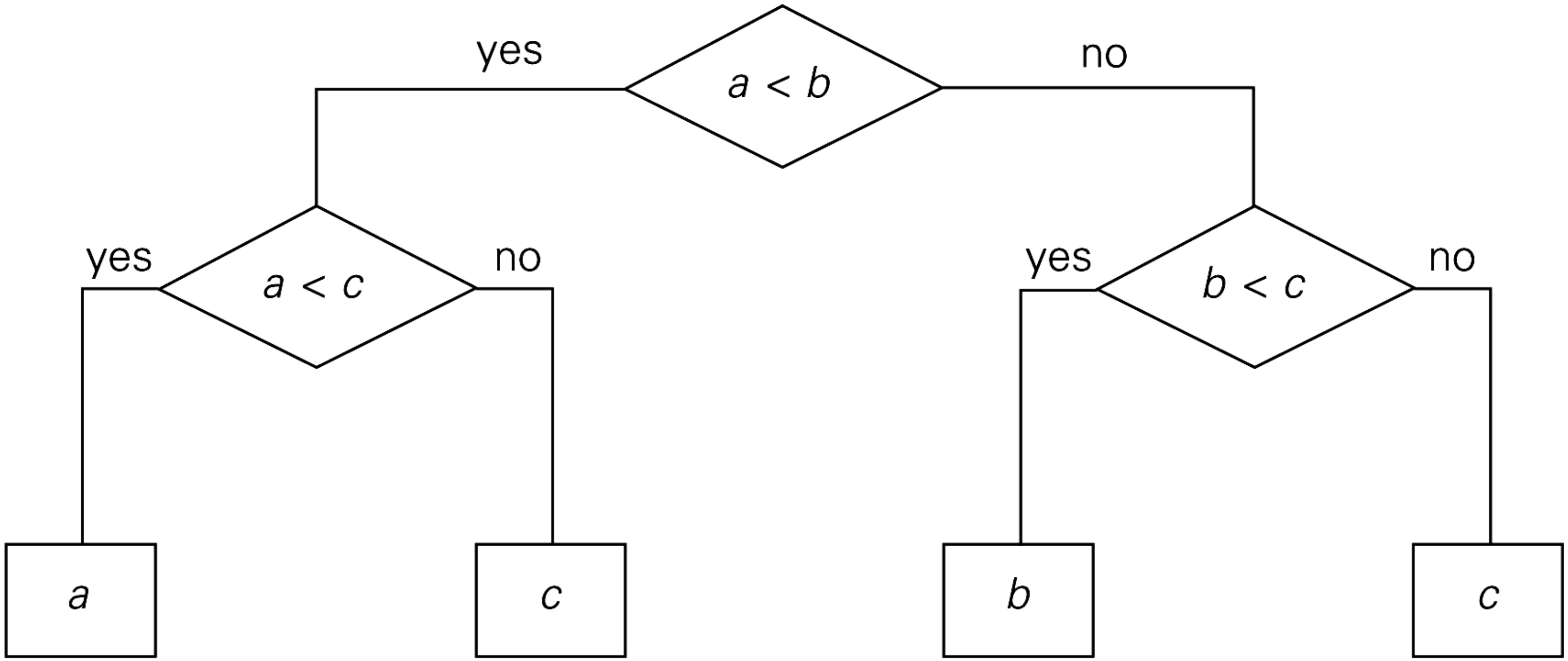
Conclusions

- may or may not be useful!
- be careful deciding how many elements *must* be processed

Decision Trees

- A model for algorithms (that involve comparisons) in which:
  - internal nodes represent comparisons
  - leaves represent outcomes

Minimum of three numbers



$a < c \wedge a < b$ $\quad$ $c \leq a < b$ $\quad$ $b \leq a \wedge b < c$ $\quad$ $c \leq b \leq a$

Median of three numbers

Median of three numbers

- Draw the decision tree (using 2-way comparisons) for finding the *median* of three numbers

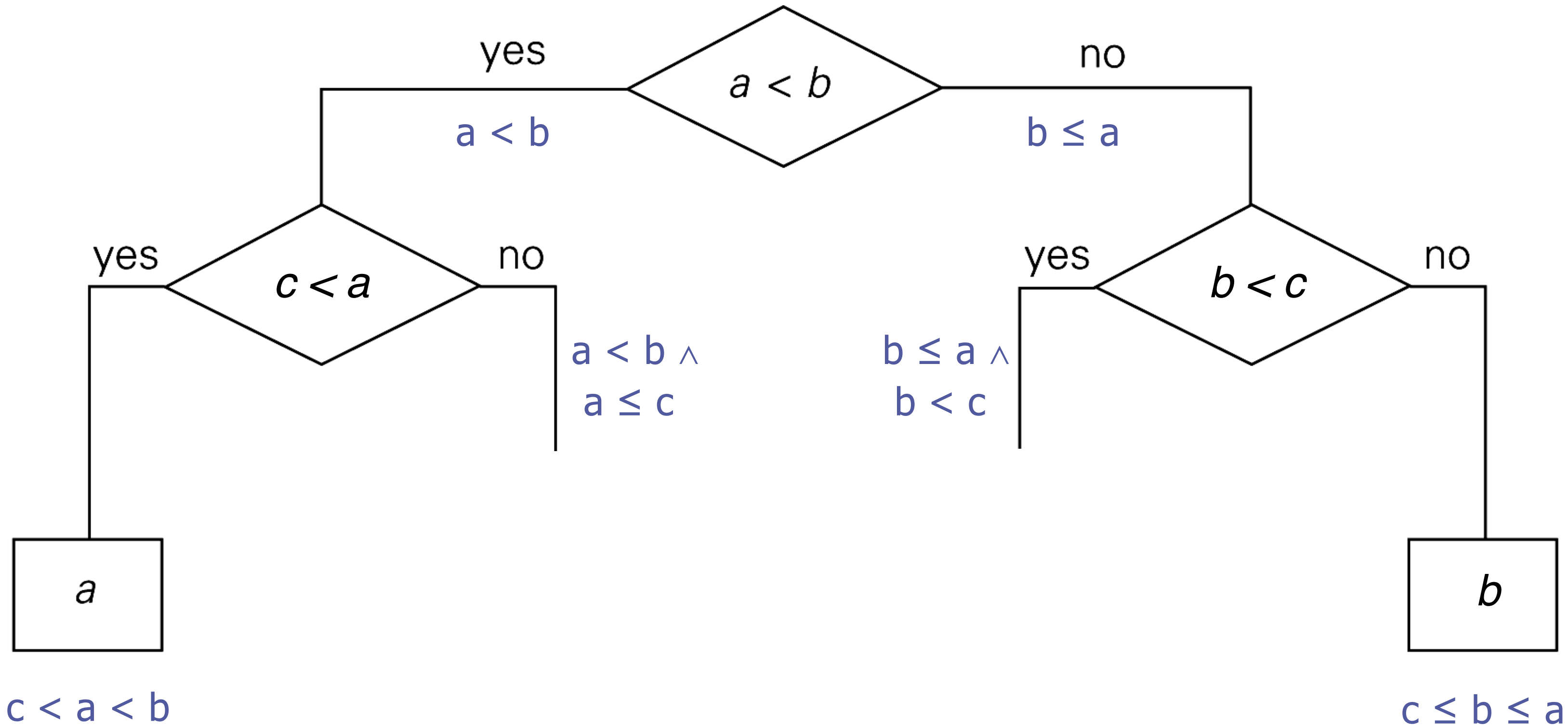
Median of three numbers

- Draw the decision tree (using 2-way comparisons) for finding the *median* of three numbers

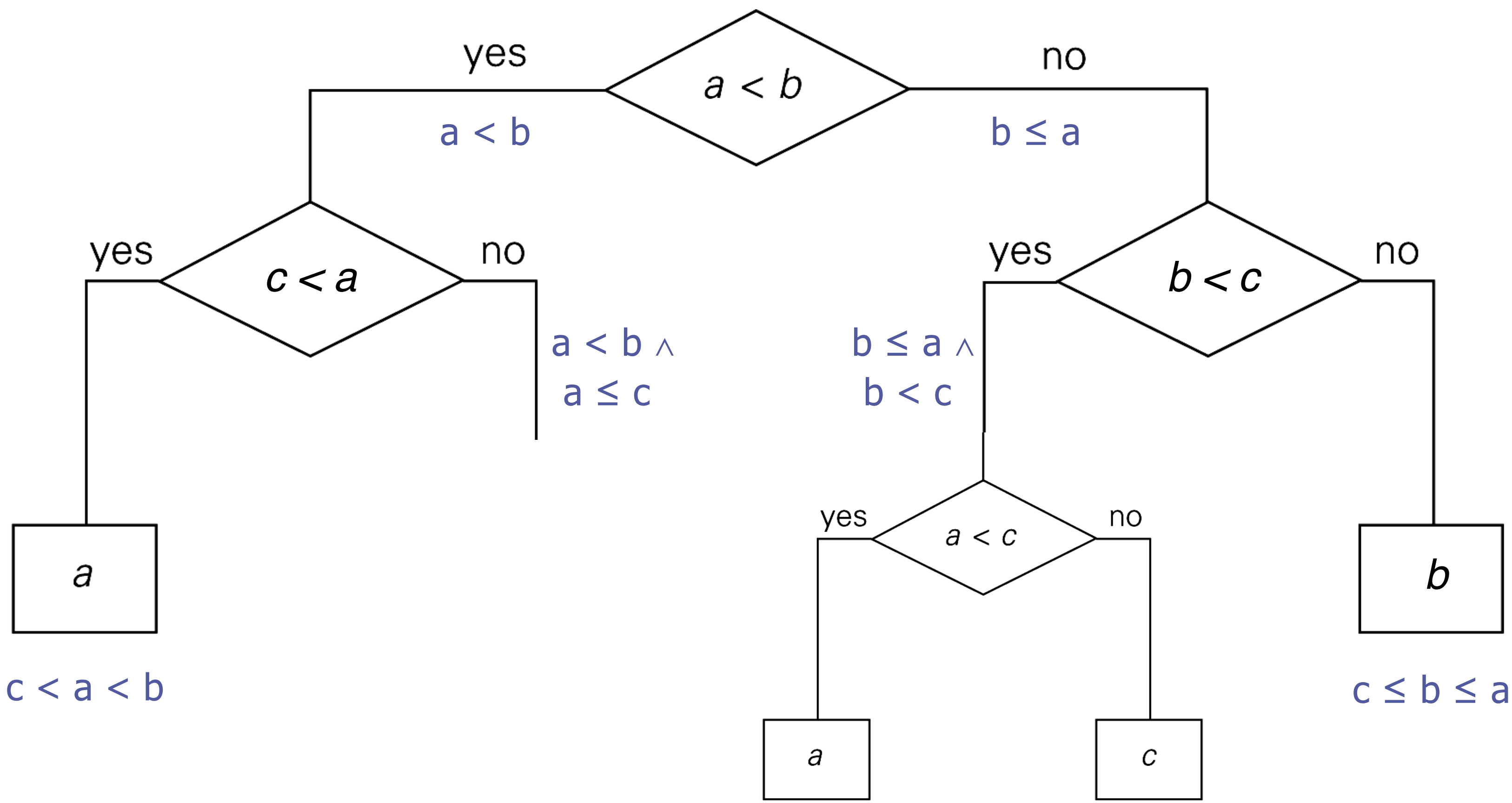
- What's the information-theoretic lower bound on the number of 2-way comparisons needed to find the *median* of three numbers?

A. 1
B. 2
C. 3
D. None of the above

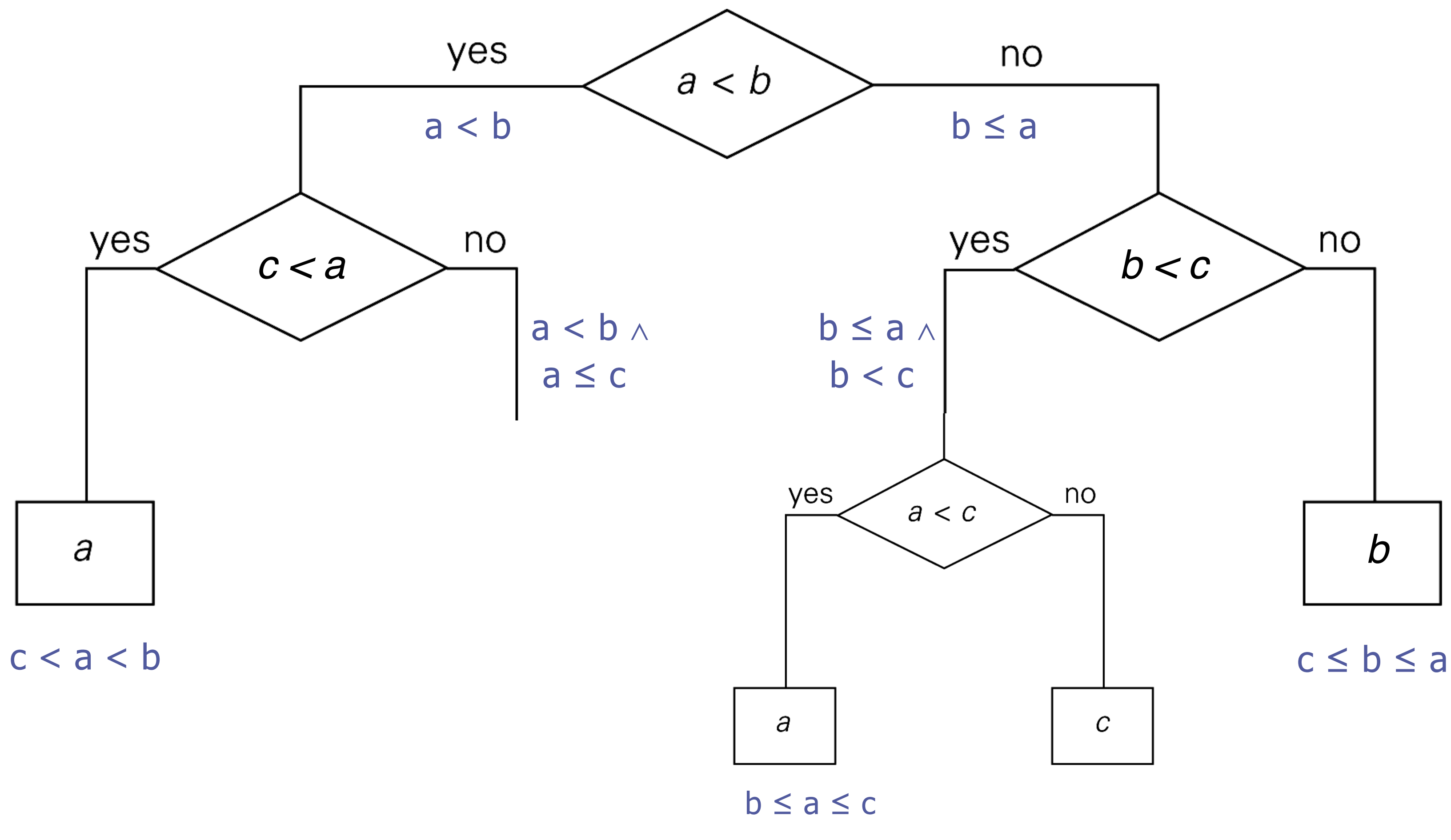
Median of three numbers



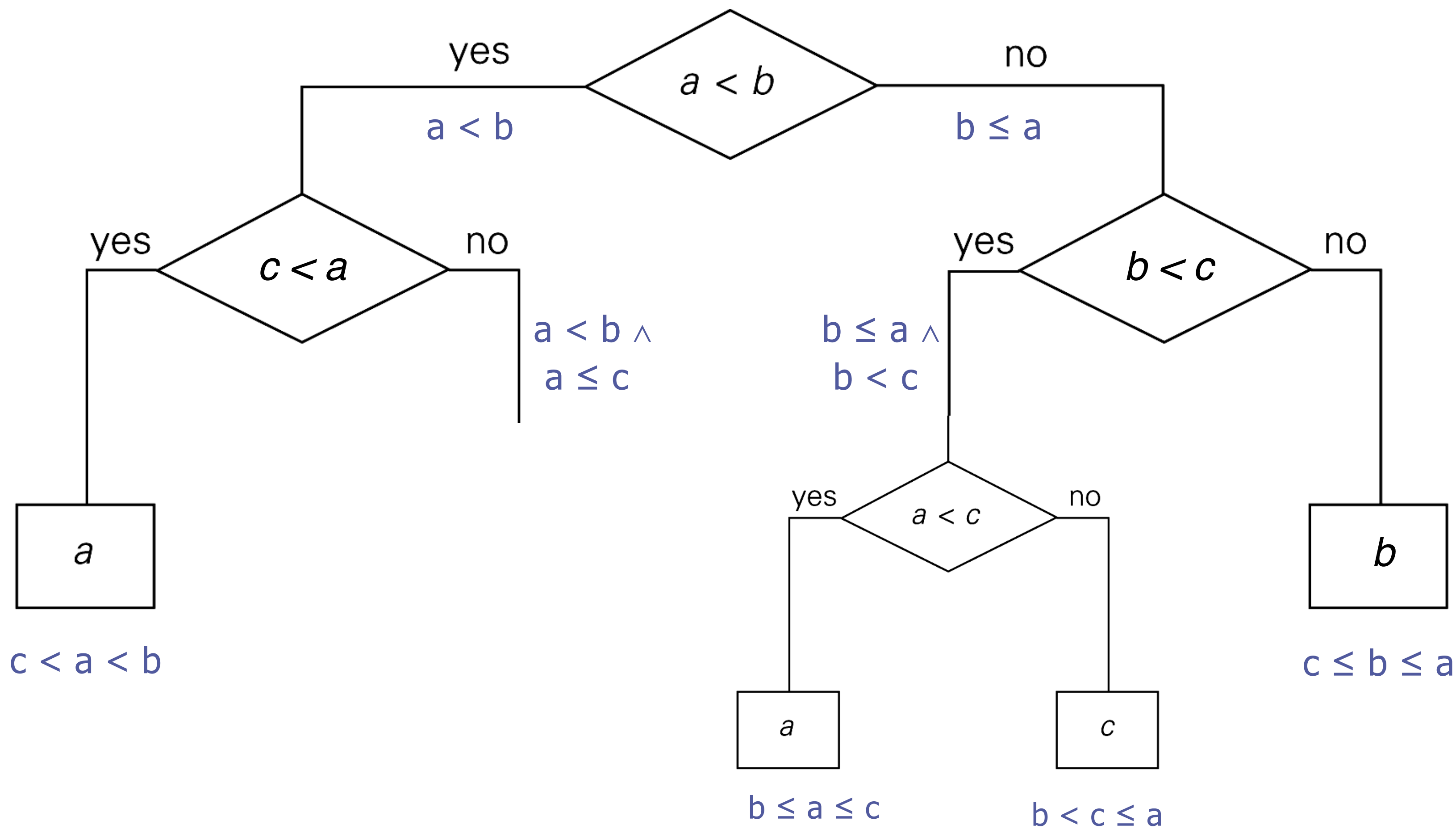
Median of three numbers



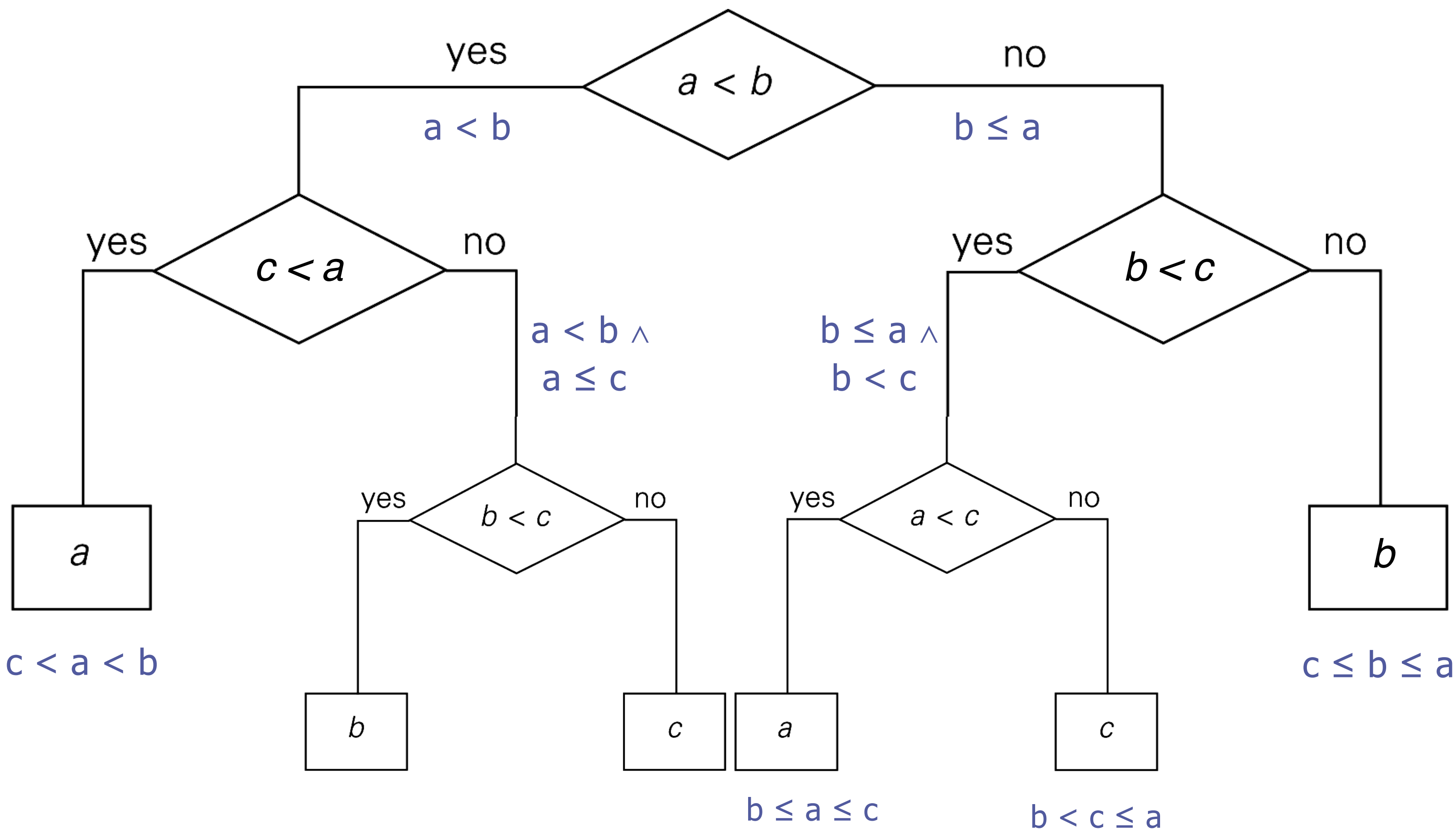
Median of three numbers



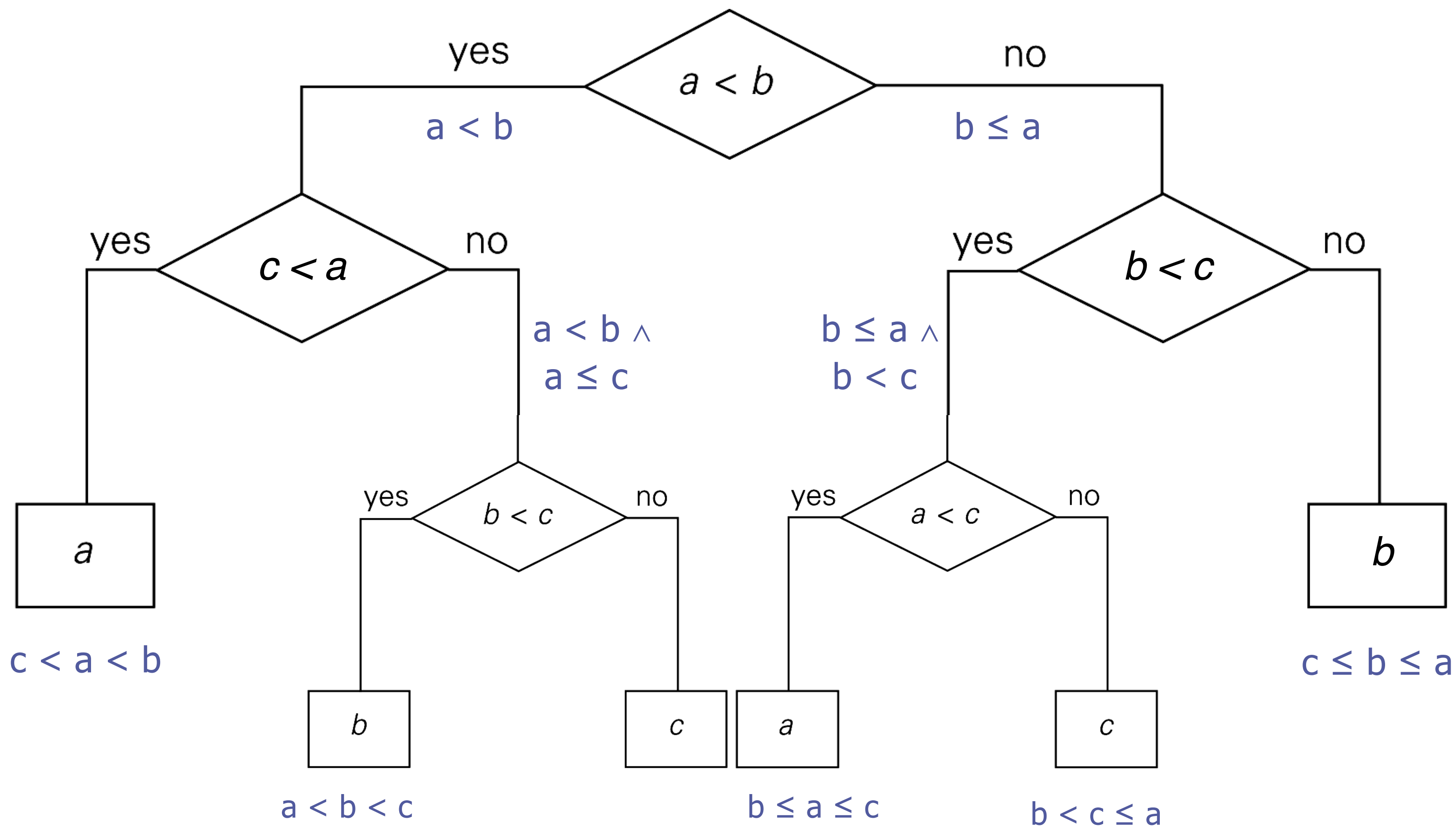
Median of three numbers



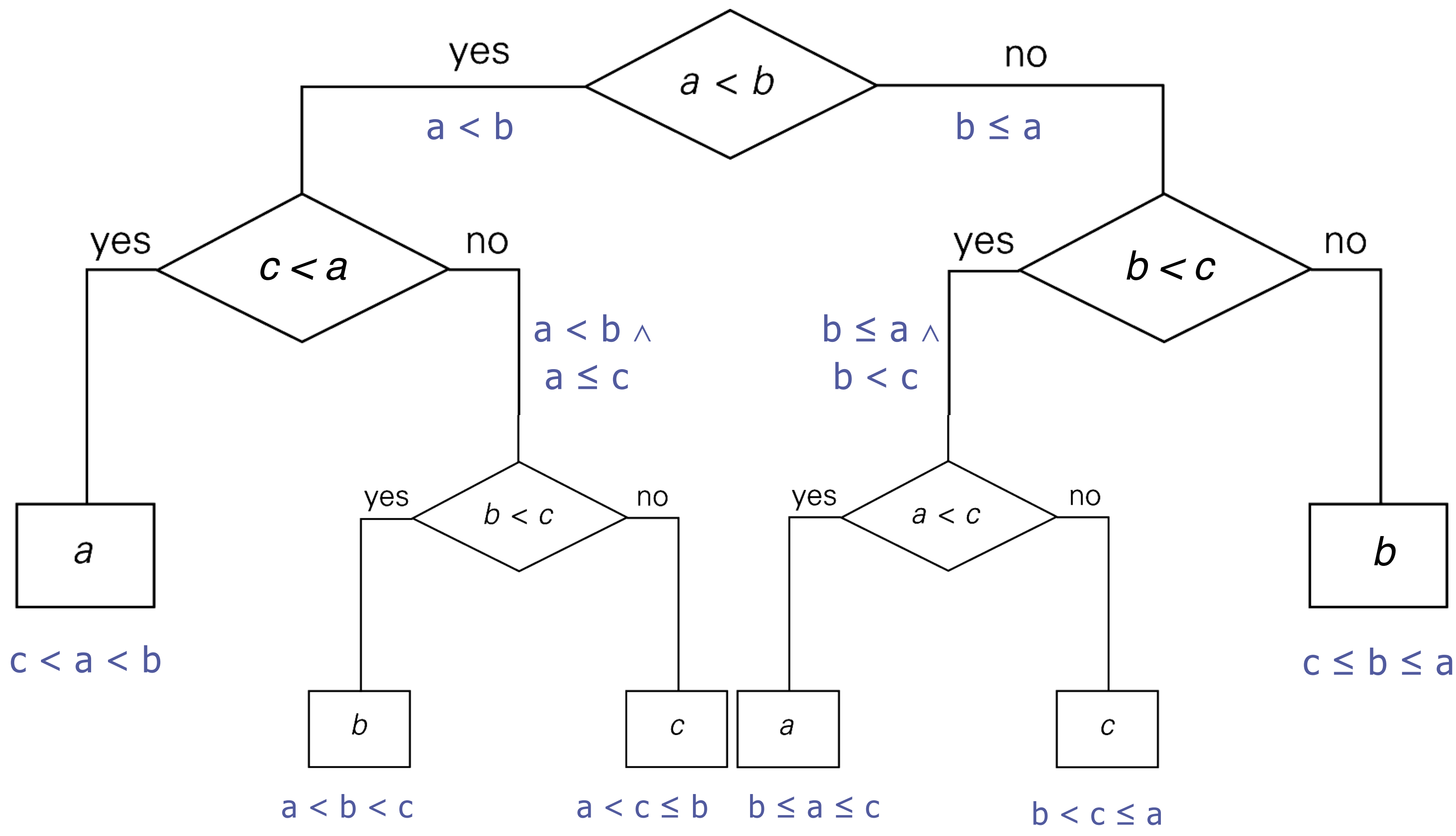
Median of three numbers



Median of three numbers



Median of three numbers



Median of three numbers?

- Is the number of comparisons (in the worst case) in your decision-tree greater than the lower bound?

A. Yes, it's greater than the lower bound
B. No, it's equal to the lower bound
C. No, it's less than the lower bound

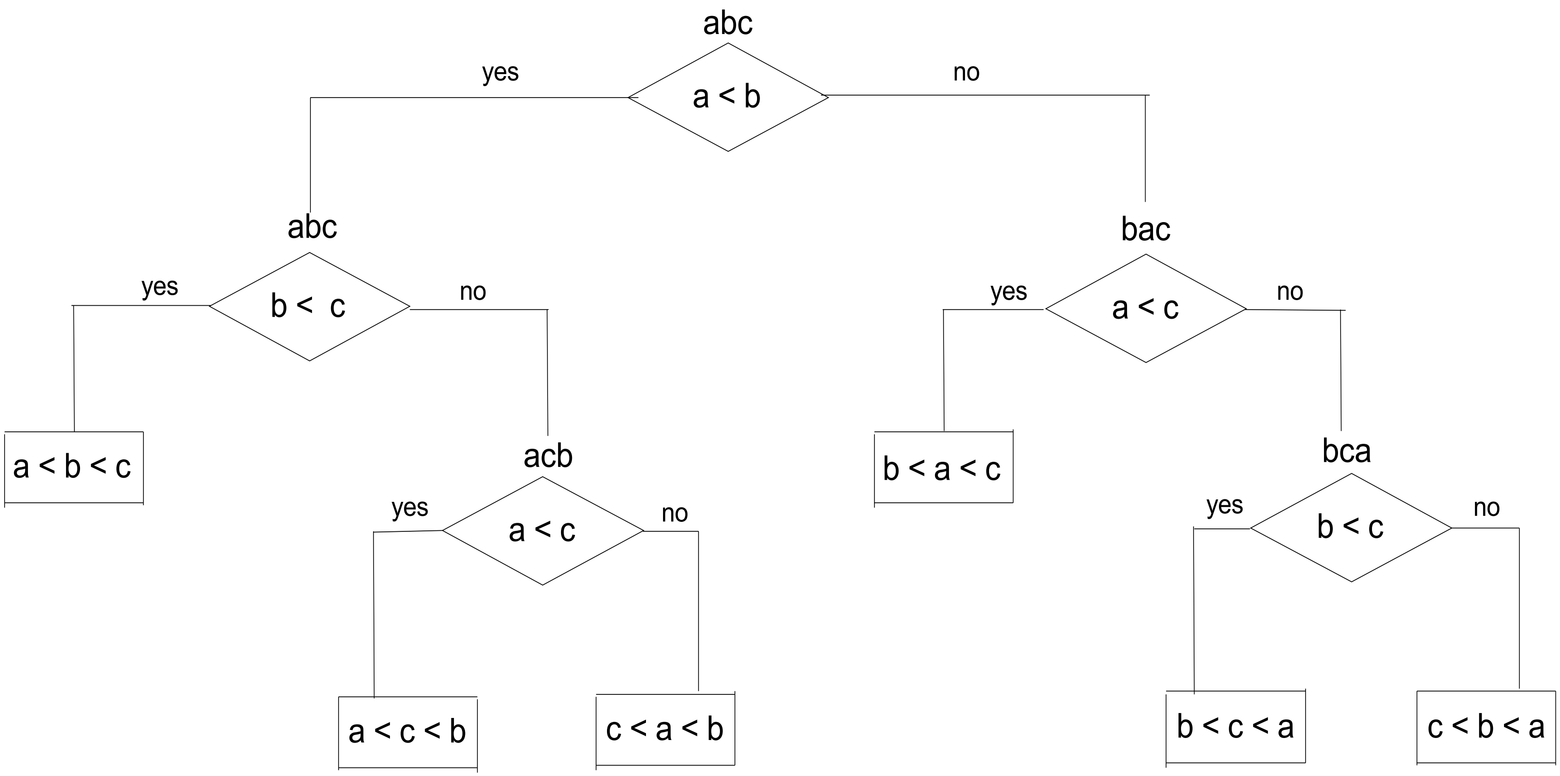
Median of three numbers

- Can you prove that no better algorithm exists?

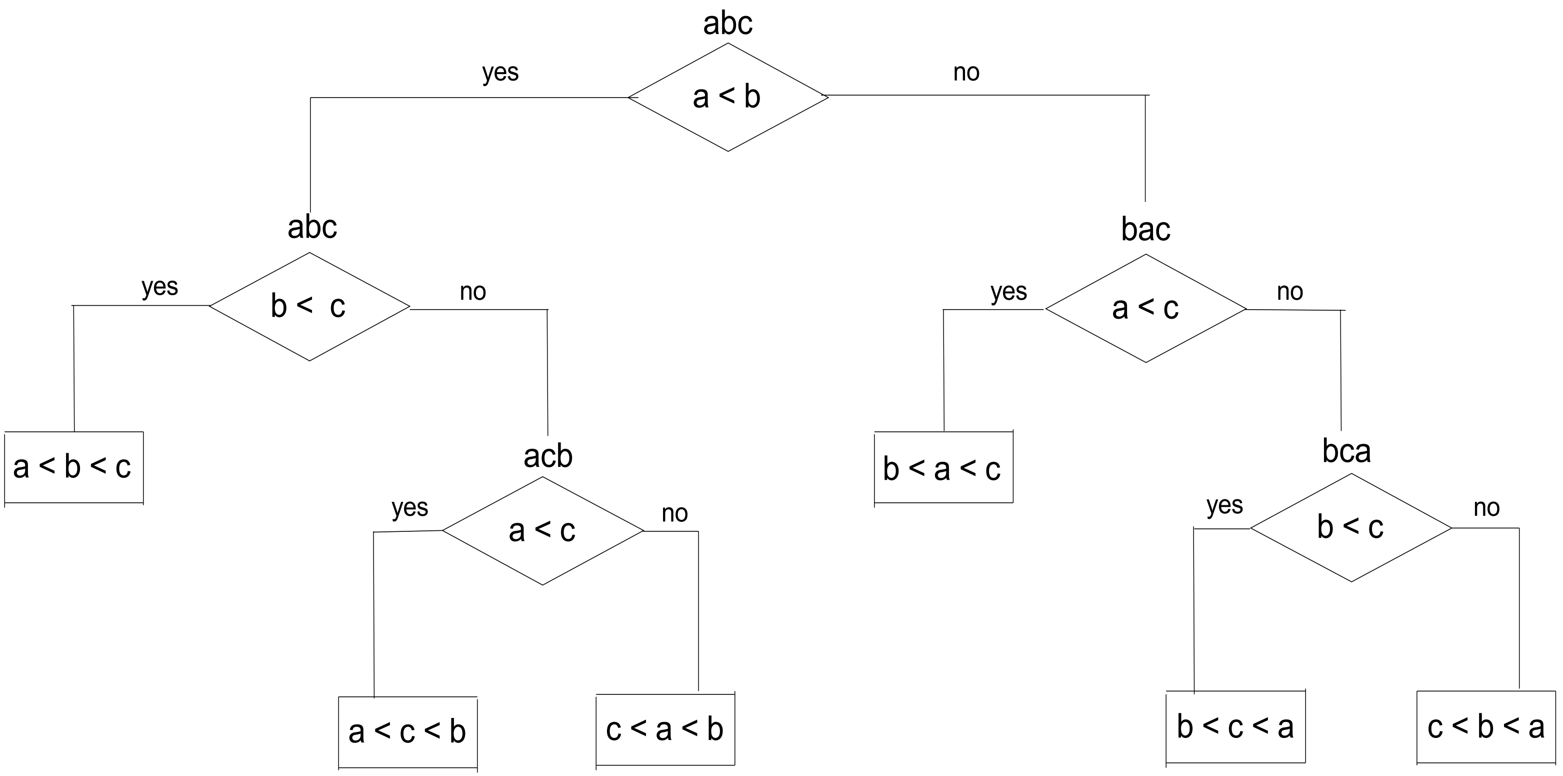
Median of three numbers

- Can you prove that no better algorithm exists?
- Issue: more leaves (6) than outcomes (3)
- Can you find a tree with lesser height (= fewer comparisons) ?

Decision tree for 3-element insertion sort:

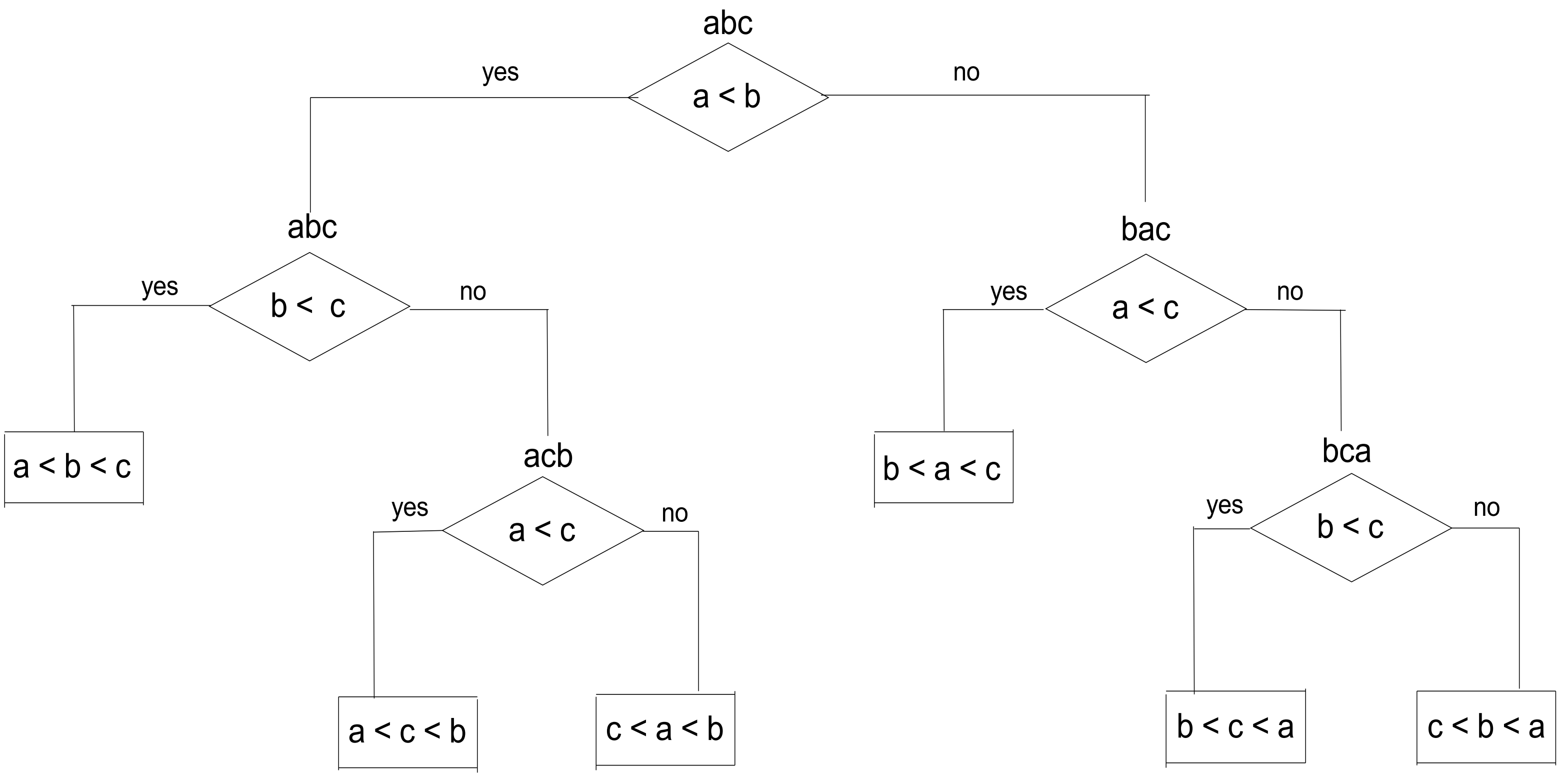


Decision tree for 3-element insertion sort:



- Average number of comparisons?

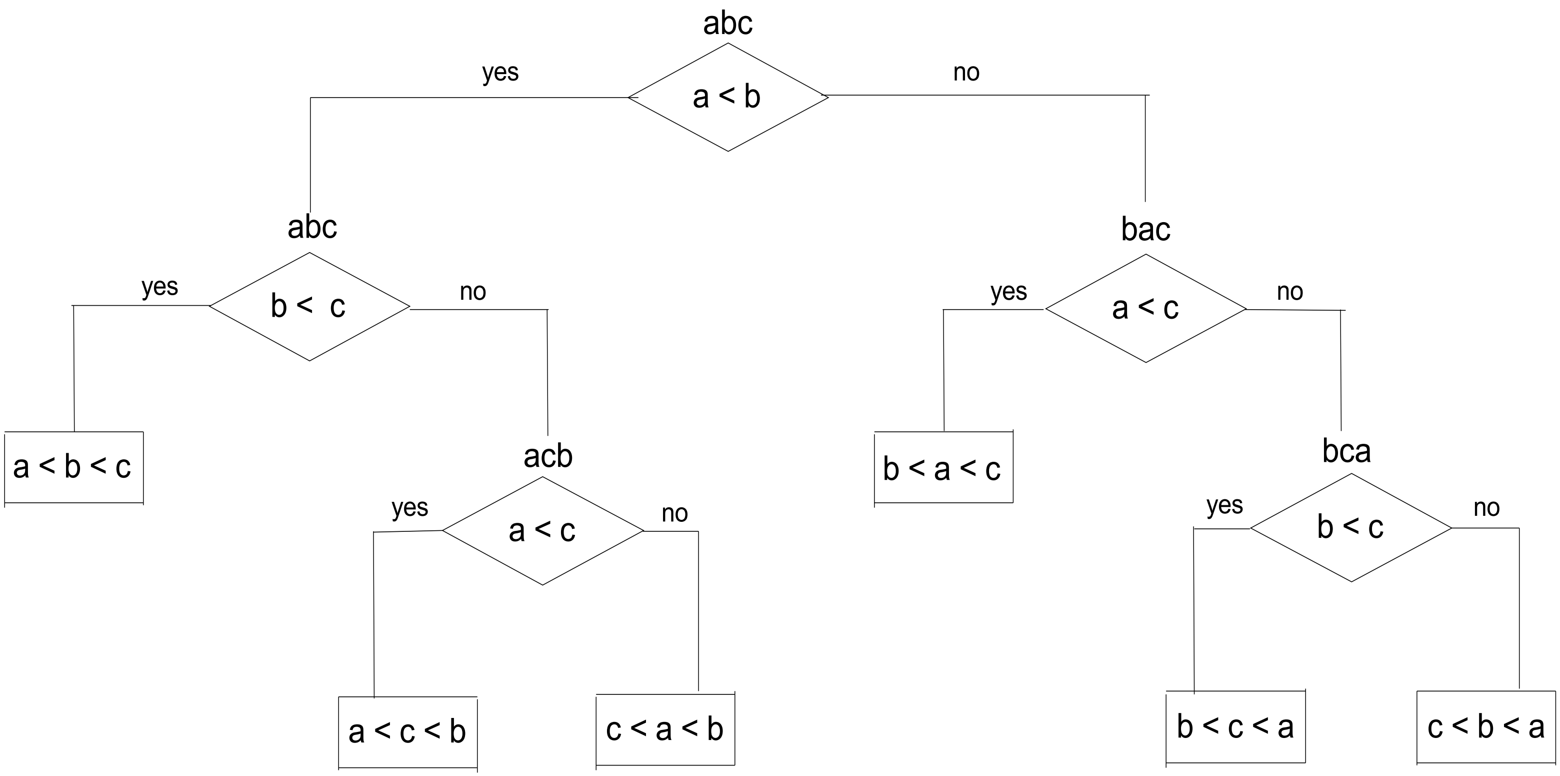
Decision tree for 3-element insertion sort:



- Average number of comparisons?

assume results are equiprobable

Decision tree for 3-element insertion sort:



- Average number of comparisons?

assume results are equiprobable

$(2 + 3 + 3 + 2 + 3 + 3) / 6 = {}^{16}/_{6} = {}^{8}/_{3} = 2\ {}^{2}/_{3}$

Decision Trees and Sorting Algorithms

Decision Trees and Sorting Algorithms

- *Any* comparison-based sorting algorithm can be represented by a decision tree

Decision Trees and Sorting Algorithms

- *Any* comparison-based sorting algorithm can be represented by a decision tree
- Number of leaves (outcomes) = $n!$

Decision Trees and Sorting Algorithms

- *Any* comparison-based sorting algorithm can be represented by a decision tree
- Number of leaves (outcomes) = $n!$
- Height of binary tree with $n!$ leaves $\geq \lceil \lg n! \rceil$

Decision Trees and Sorting Algorithms

- *Any* comparison-based sorting algorithm can be represented by a decision tree
- Number of leaves (outcomes) $= \; n!$
- Height of binary tree with $n!$ leaves $\geq \lceil \lg n! \rceil$
- Minimum number of comparisons in the worst case $\geq \lceil \lg n! \rceil$ for *any* comparison-based algorithm

Decision Trees and Sorting Algorithms

- *Any* comparison-based sorting algorithm can be represented by a decision tree
- Number of leaves (outcomes) = $n!$
- Height of binary tree with $n!$ leaves $\geq \lceil \lg n! \rceil$
- Minimum number of comparisons in the worst case $\geq \lceil \lg n! \rceil$ for *any* comparison-based algorithm
- $\lceil \lg n! \rceil \in \Omega(n \lg n)$ *(Why?)*

Decision Trees and Sorting Algorithms

- *Any* comparison-based sorting algorithm can be represented by a decision tree
- Number of leaves (outcomes) = $n!$
- Height of binary tree with $n!$ leaves $\geq \lceil \lg n! \rceil$
- Minimum number of comparisons in the worst case $\geq \lceil \lg n! \rceil$ for *any* comparison-based algorithm
- $\lceil \lg n! \rceil \in \Omega(n \lg n)$ *(Why?)*
- Is this lower bound tight? A: Yes B: No

Jigsaw puzzle

Jigsaw puzzle

- A jigsaw puzzle contains n pieces. A "section" of the puzzle is a set of one or more pieces that have been connected to each other. A "move" consists of connecting two sections. What algorithm will minimize the number of moves required to complete the puzzle?

Jigsaw puzzle

- A jigsaw puzzle contains n pieces. A "section" of the puzzle is a set of one or more pieces that have been connected to each other. A "move" consists of connecting two sections. What algorithm will minimize the number of moves required to complete the puzzle?

- *Hint:* use a lower bound argument

Adversary Arguments

Adversary argument: a method of proving a lower bound by playing a "game" in which your opponent (the adversary) makes the algorithm work as hard as possible by adjusting the input

Example 1: "Guessing" a number between 1 and n with yes/no questions

Adversary: Puts the number in the larger of the two subsets generated by last question

Simulates the *worst case*

Example 2: Merging two sorted lists of size n

$a_1 < a_2 < \ldots < a_n$ and $b_1 < b_2 < \ldots < b_n$

Adversary: $a_i < b_j$ iff $i < j$

Output $b_1 < a_1 < b_2 < a_2 < \ldots < b_n < a_n$ requires $2n$-1 comparisons of adjacent elements

Example 2: Merging two sorted lists of size n

$a_1 < a_2 < \ldots < a_n$ and $b_1 < b_2 < \ldots < b_n$

Adversary: $a_i < b_j$ iff $i < j$

Output $b_1 < a_1 < b_2 < a_2 < \ldots < b_n < a_n$ requires $2n$-1 comparisons of adjacent elements

b_1 to a_1, a_1 to b_2, b_2 to a_2, etc.

Suppose that one of these comparisons is not made …

Lower Bounds by Problem Reduction

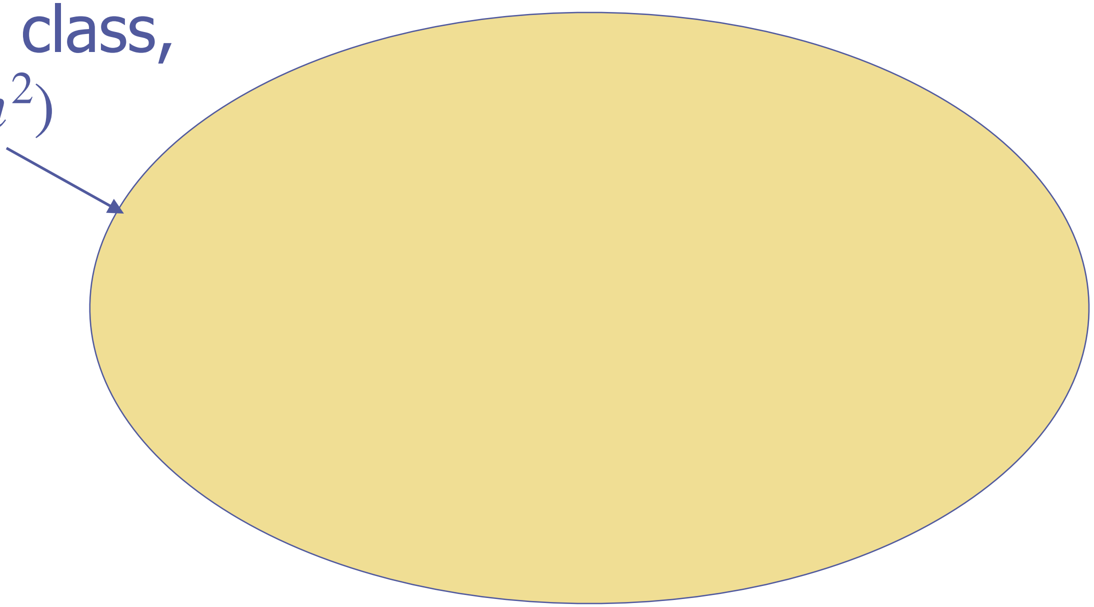
Idea: If problem P is "at least as hard" as problem Q, then a lower bound for Q is also a lower bound for P.

Hence: find problem Q with a known lower bound *that can be reduced to* problem P.

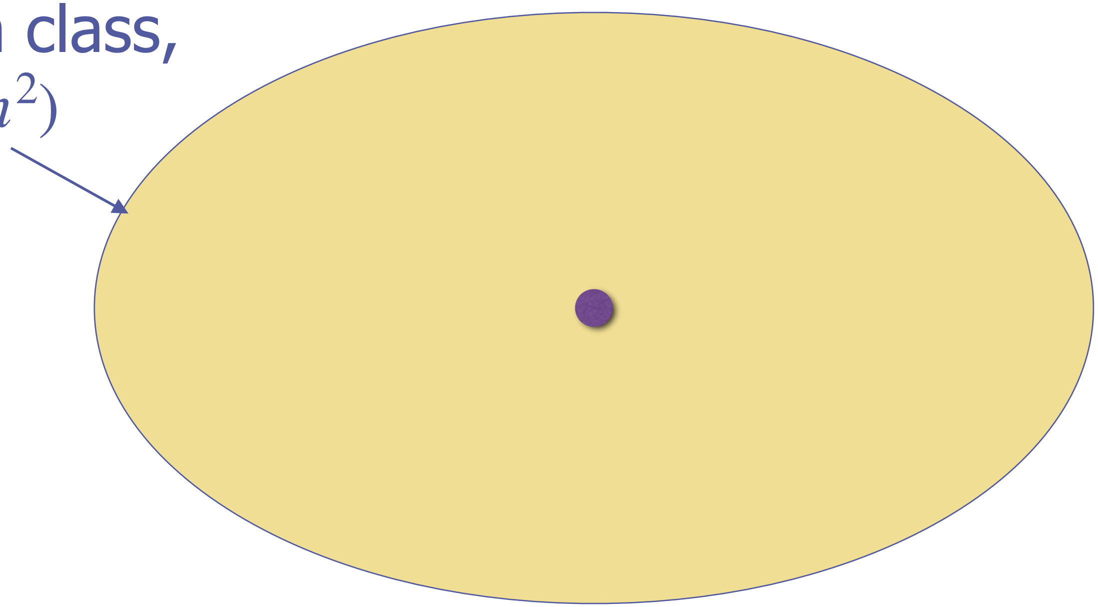
Example:

- You need a lower bound for P: finding minimum spanning tree for n points in Cartesian plane
- Q is element uniqueness problem — known to be in $\Omega(n \log n)$.
- Reduce Q to P (note direction: known → unknown)

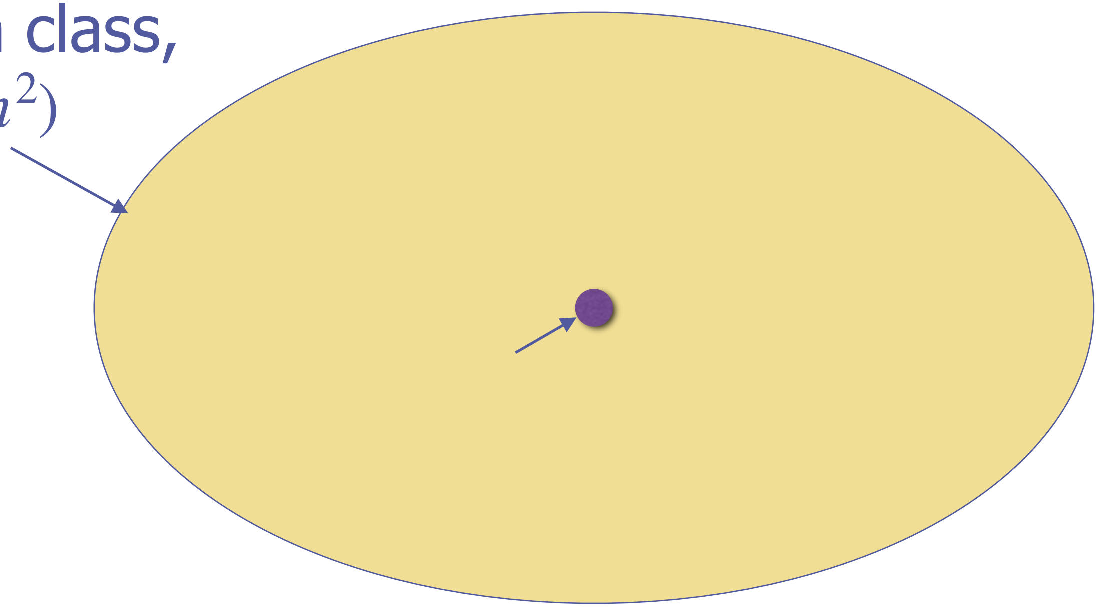
problem class,
e.g., $\Omega(n^2)$

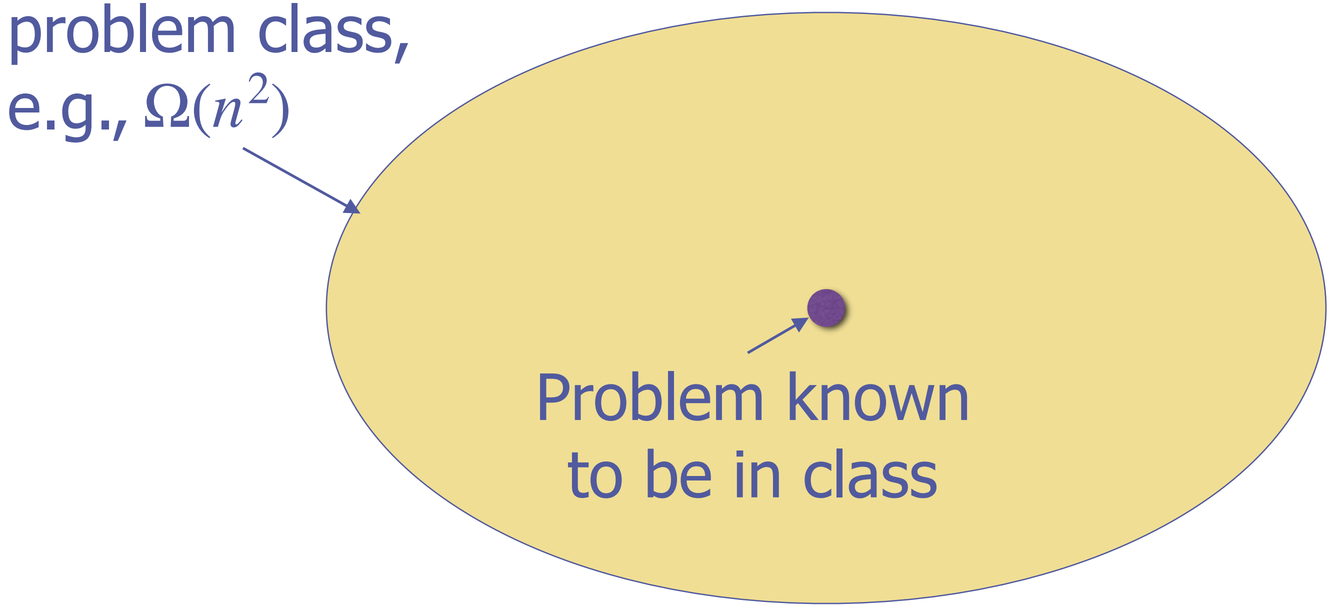


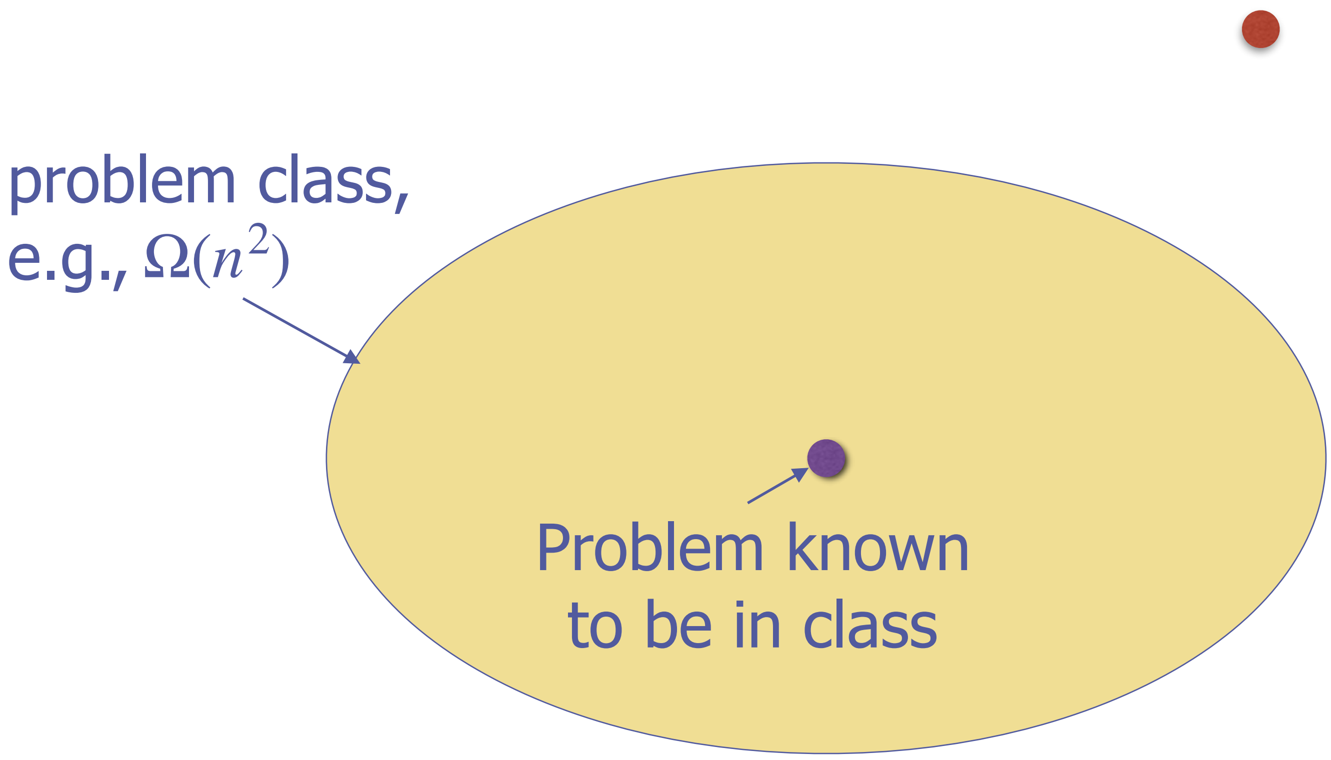
problem class,
e.g., $\Omega(n^2)$

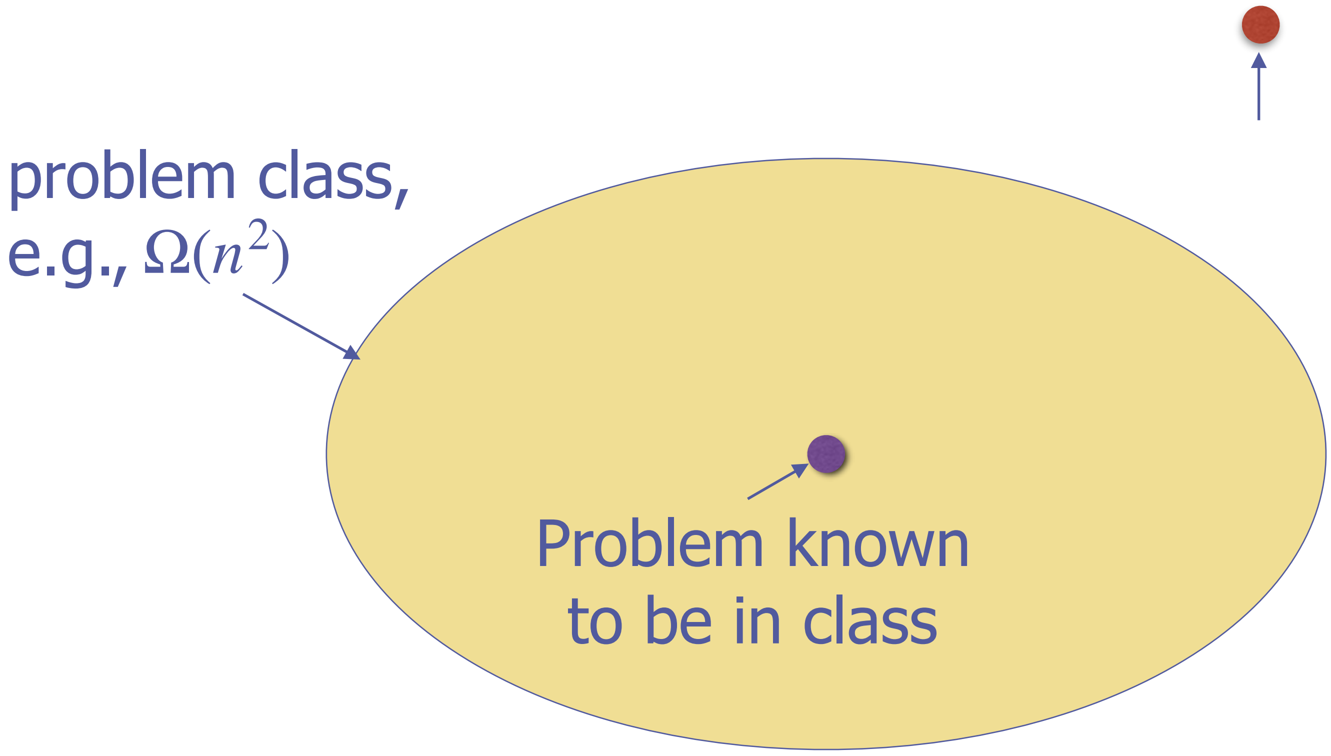


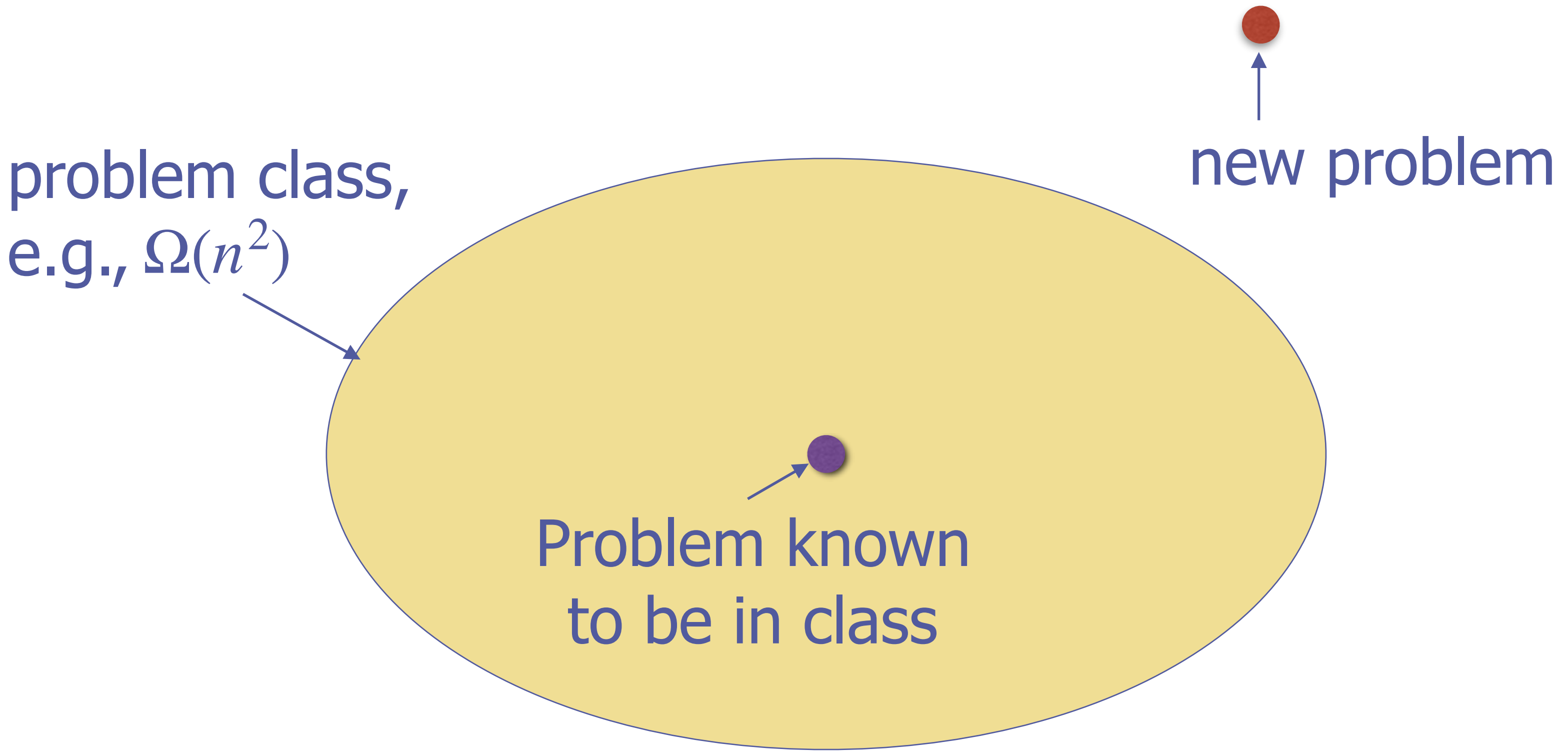
problem class,
e.g., $\Omega(n^2)$

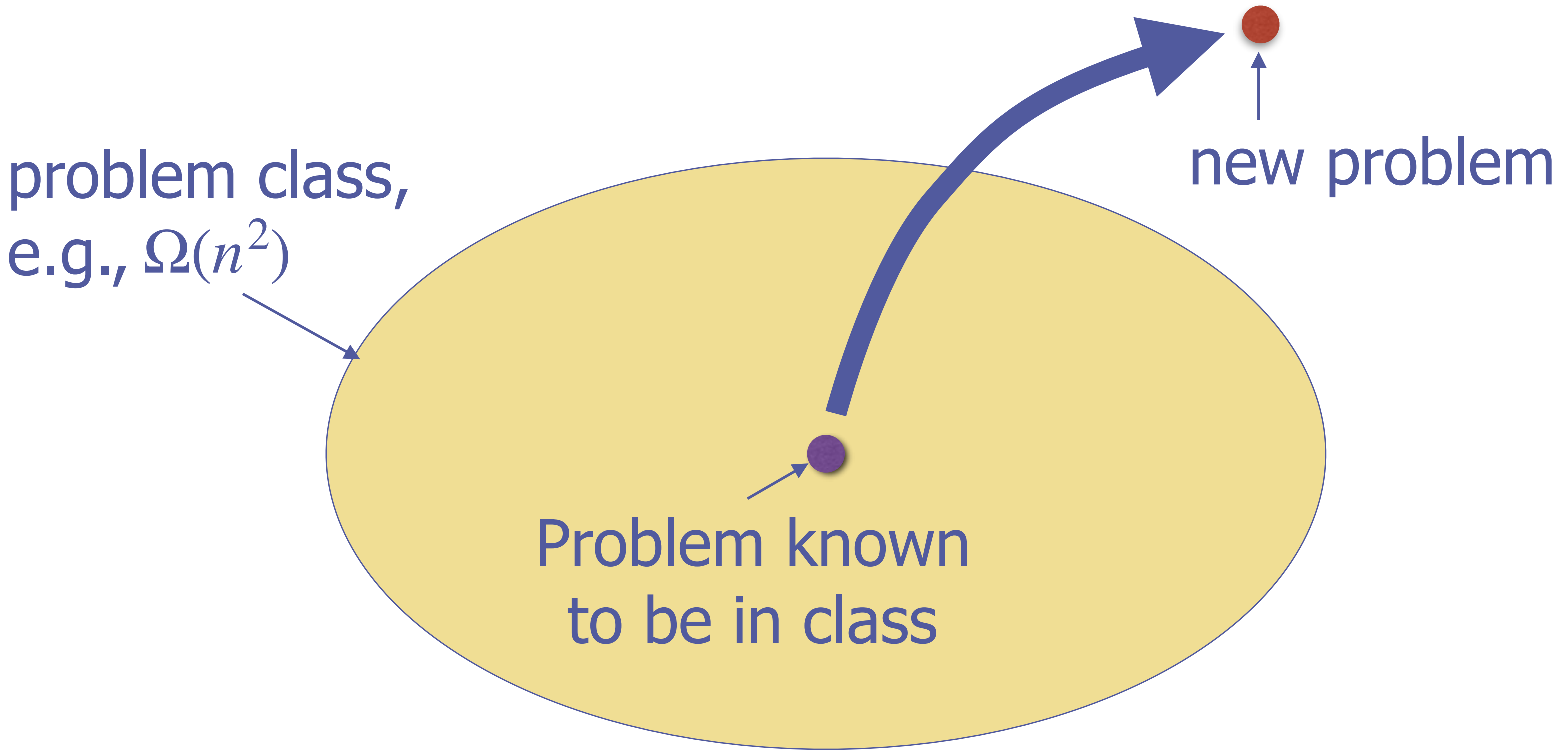


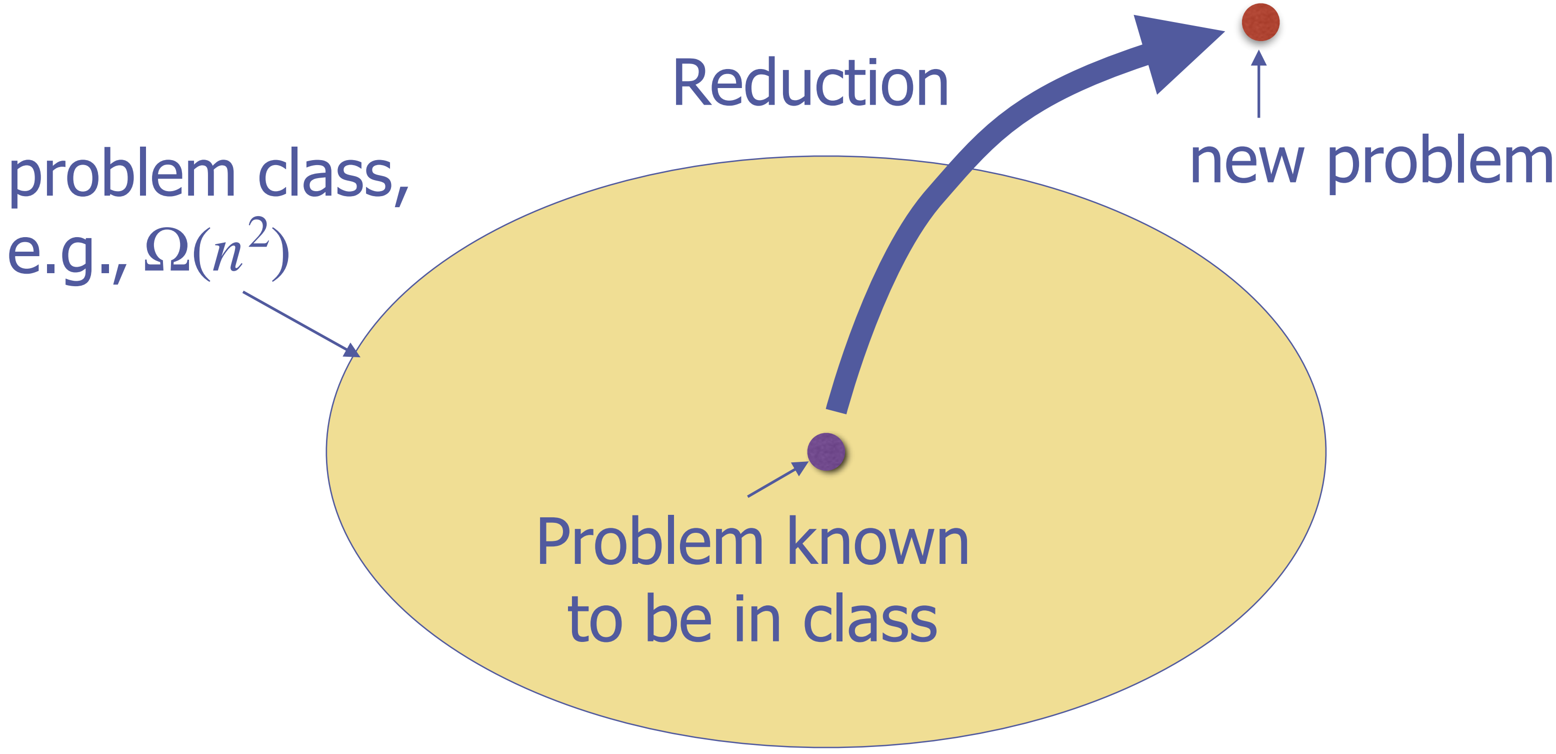

problem class,
e.g., $\Omega(n^2)$
Problem known
to be in class

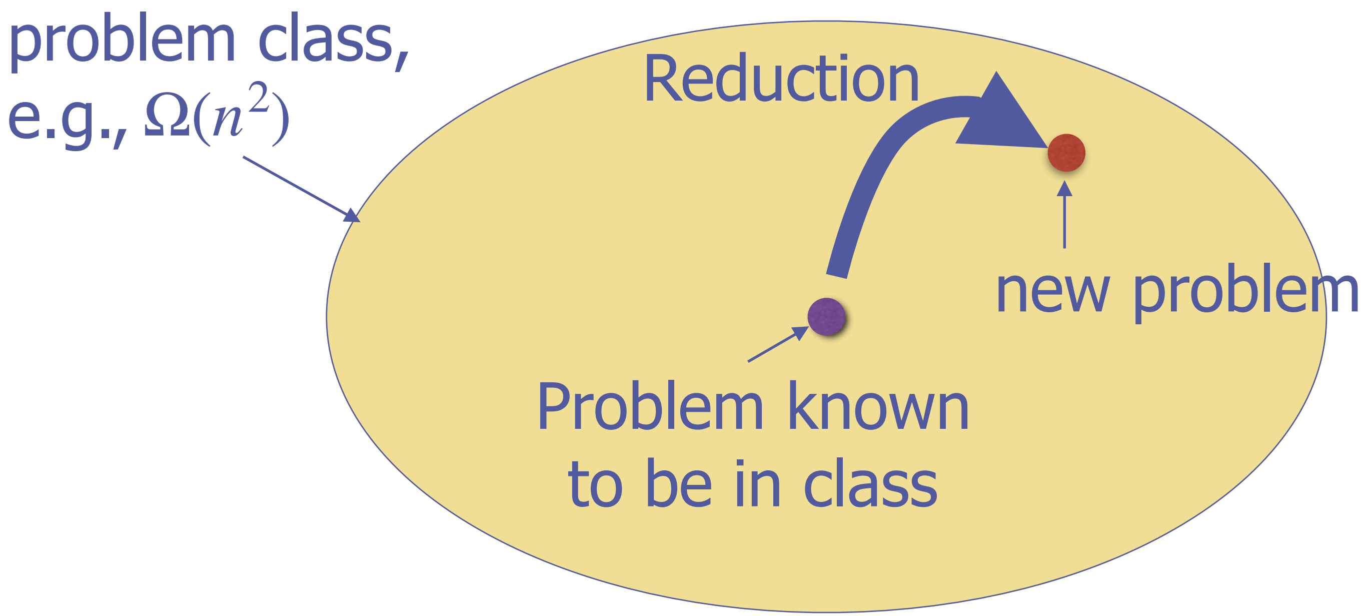

problem class,
e.g., $\Omega(n^2)$
Problem known
to be in class


problem class,
e.g., $\Omega(n^2)$
Problem known
to be in class


new problem
problem class,
e.g., $\Omega(n^2)$
Problem known
to be in class

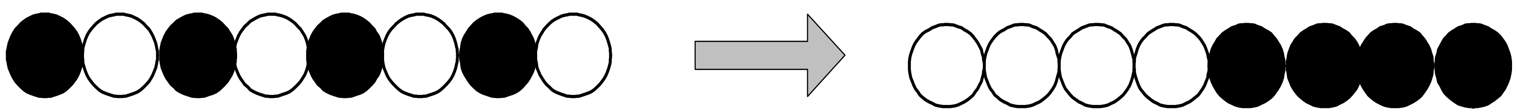

problem class,
e.g., $\Omega(n^2)$
new problem
Problem known
to be in class


Reduction
new problem
problem class,
e.g., $\Omega(n^2)$
Problem known
to be in class


problem class,
e.g., $\Omega(n^2)$
Reduction
new problem
Problem known
to be in class

Problem

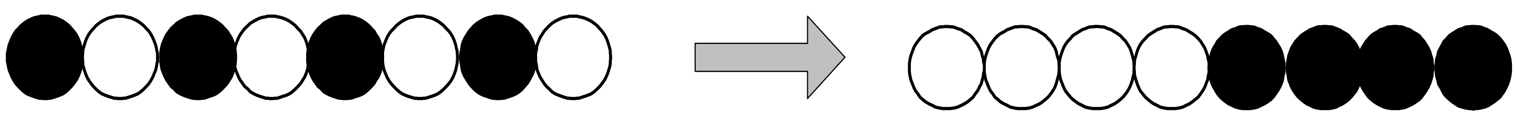
Alternating disks You have a row of $2n$ disks of two colors, n dark and n light. They alternate: dark, light, dark, light, and so on. You want to get all the dark disks to the right-hand end, and all the light disks to the left-hand end. The only moves you are allowed to make are those which interchange the positions of two neighboring disks.



Problem

- Prove that *any* algorithm solving the alternating disk puzzle must make at least $n(n+1)/2$ moves to solve it

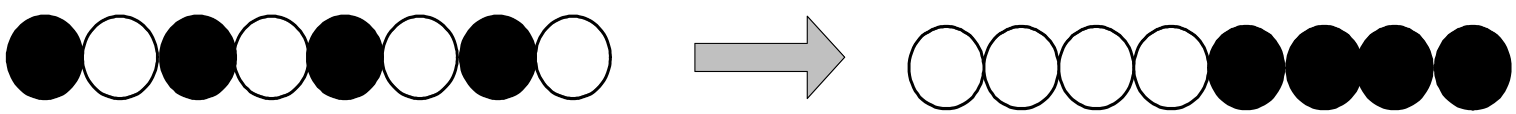
Alternating disks You have a row of $2n$ disks of two colors, n dark and n light. They alternate: dark, light, dark, light, and so on. You want to get all the dark disks to the right-hand end, and all the light disks to the left-hand end. The only moves you are allowed to make are those which interchange the positions of two neighboring disks.



Problem

- Prove that *any* algorithm solving the alternating disk puzzle must make at least $n(n+1)/2$ moves to solve it
- Is this lower bound tight? A: Yes B: No

Alternating disks You have a row of $2n$ disks of two colors, n dark and n light. They alternate: dark, light, dark, light, and so on. You want to get all the dark disks to the right-hand end, and all the light disks to the left-hand end. The only moves you are allowed to make are those which interchange the positions of two neighboring disks.



Problem

- Find a trivial lower-bound for the following problem. Is this bound tight?
  - find the largest element in an n-element array

A. $\Omega(1)$

B. $\Omega(n)$

C. $\Omega(n \lg n)$

D. None of the above

Problem

- Find a trivial lower-bound for the following problem. Is this bound tight?
  - is a graph with n vertices (represented by an $n \times n$ adjacency matrix) complete?

A. $\Omega(n^2)$

B. $\Omega(n^3)$

C. $\Omega(n \lg n)$

D. None of the above

Problem

- Find a <u>trivial</u> lower-bound for the following problem. Is this bound tight?
  - generate all subsets of an n-element set

A. $\Omega(n^2)$

B. $\Omega(n^3)$

C. $\Omega(n^n)$

D. $\Omega(2^n)$

E. None of the above

Problem

- Find a trivial lower-bound for the following problem. Is this bound tight?
  - are all the members of a set of n real numbers distinct?

A. $\Omega(n)$
B. $\Omega(n^2)$
C. $\Omega(n \lg n)$
D. None of the above

Fake-coin Problem

You have $n > 2$ identical-looking coins and a two-pan balance with no weights. One of the coins is a fake, but you do not know whether it is lighter or heavier than the genuine coins, which all weigh the same.

What is the *information-theoretic lower bound* on the number of 3-way weighings required to determine if the fake coin is *light* or *heavy*?

A. $\Omega(1)$ B. $\Omega(n)$ C. $\Omega(\lg n)$ D. something else

Problem

Problem

- What do information-theoretic arguments tell us about Fake-coins problem?

Problem

- What do information-theoretic arguments tell us about Fake-coins problem?
- Suppose we wish to discover *which* coin is fake

Problem

- What do information-theoretic arguments tell us about Fake-coins problem?
- Suppose we wish to discover *which* coin is fake
- Can we deduce that we will need at least $\lceil \log_2 n \rceil$ weighings?

Problem

- What do information-theoretic arguments tell us about Fake-coins problem?
- Suppose we wish to discover *which* coin is fake
- Can we deduce that we will need at least $\lceil \log_2 n \rceil$ weighings?
- If $n=12$, what is $\lceil \log_2 n \rceil$?

Problem

- What do information-theoretic arguments tell us about Fake-coins problem?
- Suppose we wish to discover *which* coin is fake
- Can we deduce that we will need at least $\lceil \log_2 n \rceil$ weighings?
- If $n=12$, what is $\lceil \log_2 n \rceil$?
- Can you solve the problem with < 4 weighings?

12 Coins

- If one out of 12 coins is too light:
  - 12 possible outcomes
- If we don't know whether the fake is heavy or light:
  - 24 possible outcomes

- 1 weighing: 3 outcomes
- 3 weighings: $3^3 = 27$ outcomes
- Therefore: there is enough information in three weighings to distinguish between the 24 possibilities

Lower-bounds by reduction

TABLE 11.1 Problems often used for establishing lower bounds by problem reduction

Problem	Lower bound	Tightness
sorting	$\Omega(n \log n)$	yes
searching in a sorted array	$\Omega(\log n)$	yes
element uniqueness problem	$\Omega(n \log n)$	yes
multiplication of n-digit integers	$\Omega(n)$	unknown
multiplication of $n \times n$ matrices	$\Omega(n^2)$	unknown

Lower-bounds by reduction

- Find a tight lower bound for the problem of finding the two closest numbers in a set of n real numbers.

TABLE 11.1 Problems often used for establishing lower bounds by problem reduction

Problem	**Lower bound**	**Tightness**
sorting	$\Omega(n \log n)$	yes
searching in a sorted array	$\Omega(\log n)$	yes
element uniqueness problem	$\Omega(n \log n)$	yes
multiplication of n-digit integers	$\Omega(n)$	unknown
multiplication of $n \times n$ matrices	$\Omega(n^2)$	unknown

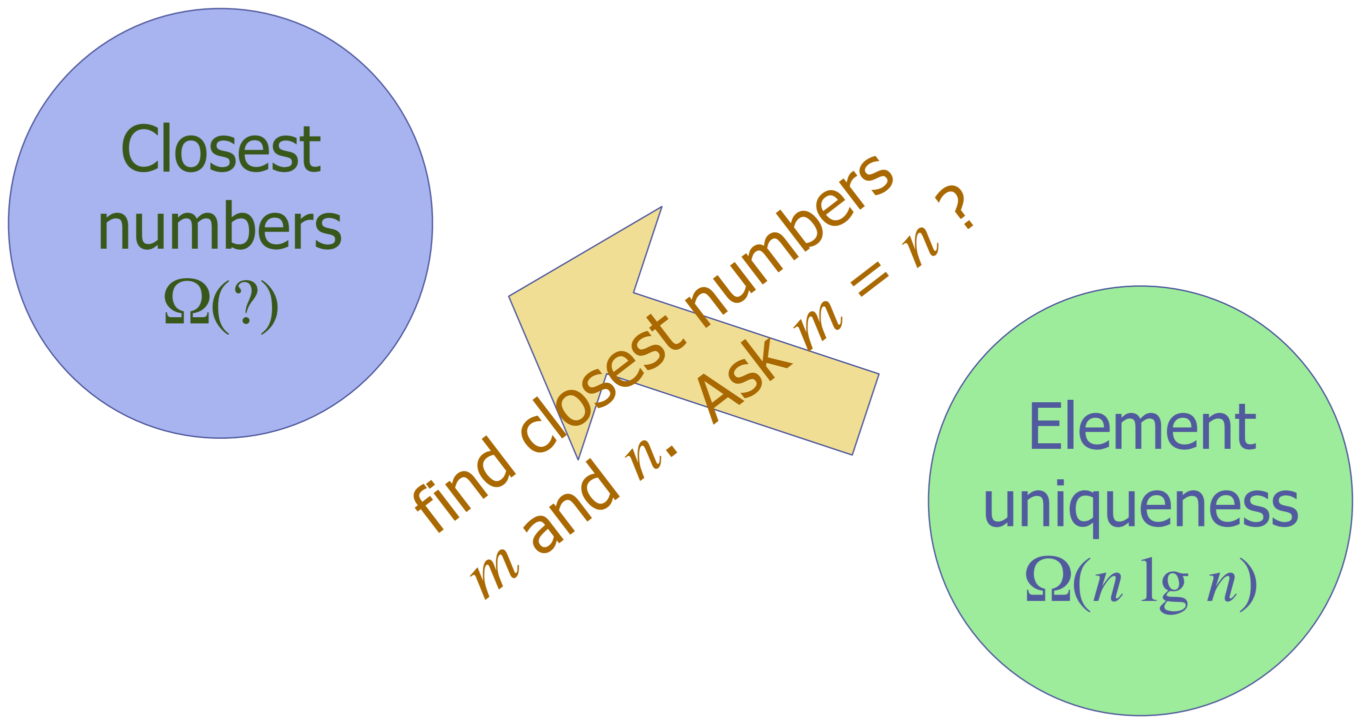
Lower-bounds by reduction

- Find a tight lower bound for the problem of finding the two closest numbers in a set of n real numbers.
- Hint: use a reduction from the element uniqueness problem.

TABLE 11.1 Problems often used for establishing lower bounds by problem reduction

Problem	Lower bound	Tightness
sorting	$\Omega(n \log n)$	yes
searching in a sorted array	$\Omega(\log n)$	yes
element uniqueness problem	$\Omega(n \log n)$	yes
multiplication of n-digit integers	$\Omega(n)$	unknown
multiplication of $n \times n$ matrices	$\Omega(n^2)$	unknown

Reduction



Closest numbers
$\Omega(?)$
find closest numbers m and n. Ask $m = n$?
Element uniqueness
$\Omega(n \lg n)$

Classifying Problem Complexity

Tractability

Tractability

- Is the problem *tractable*, *i.e.*, is there a polynomial-time ($O(p(n))$) algorithm that solves it?

Tractability

- Is the problem *tractable*, *i.e.*, is there a polynomial-time ($O(p(n))$) algorithm that solves it?
- Possible answers:
  - yes (give examples)
  - no
    - because it's been proved that no algorithm exists at all
    - because it's been proved that any algorithm takes exponential time (or worse)
  - unknown

Problem Types

- *Optimization problem*: find a solution that maximizes or minimizes some objective function
- *Decision problem*: answer yes/no to a question
- Many problems have decision and optimization versions.
  - e.g.: traveling salesman problem

    optimization: find Hamiltonian cycle of minimum length

    decision: find Hamiltonian cycle of length $\leq m$
- Decision problems are more convenient for formal investigation of their complexity.

Class P

- The class of decision problems that are solvable in $O(p(n))$ time, where $p(n)$ is a polynomial in problem's input size n
- Examples:
  - searching
  - element uniqueness
  - graph connectivity
  - graph acyclicity
  - primality testing (AKS Primality test, 2002)

Class NP

- NP (nondeterministic polynomial): class of decision problems whose proposed solutions can be *verified* in polynomial time = solvable by a nondeterministic polynomial algorithm
- A nondeterministic polynomial algorithm is an abstract two-stage procedure that:
  1. generates a random string purported to solve the problem
  2. checks whether this solution is correct in polynomial time

  By definition, it solves the problem if it's capable of generating and verifying a solution on one of its tries
- Why this definition?
  - led to development of the rich theory called "computational complexity"

Example: CNF satisfiability

Problem: is a boolean expression in its conjunctive normal form (CNF) satisfiable, *i.e.*, are there values of its variables that makes it true?

This problem is in *NP*. Nondeterministic algorithm:

1. Guess truth assignment
2. Substitute the values into the CNF formula to see if it evaluates to true

Example: $(A \vee \neg B \vee \neg C) \wedge (A \vee B) \wedge (\neg B \vee \neg D \vee E) \wedge (\neg D \vee \neg E)$

A	B	C	D	E
0	0	0	0	0
		…		
1	1	1	1	1

Checking phase: $O(n)$

What problems are in NP?

- Hamiltonian circuit existence
- Partition problem: is it possible to partition a set of n integers into two disjoint subsets with the same sum?
- Decision versions of TSP, knapsack problem, graph coloring, and many other combinatorial optimization problems. (Few exceptions including MST, shortest paths)

- All the problems in P can also be solved in this manner (but no guessing is necessary), so we have:

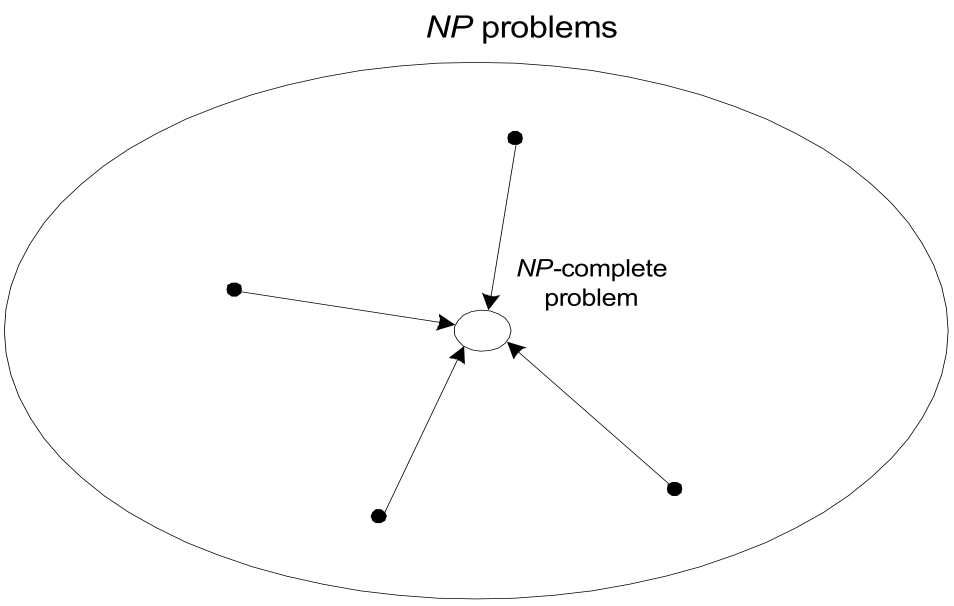
$$P \subseteq NP$$

- Big question: P = NP ?

NP-Complete Problems

- A decision problem D is NP-complete if it is as hard as any problem in NP, *i.e.*,

1. D is in NP
2. every problem in NP is polynomial-time reducible to D

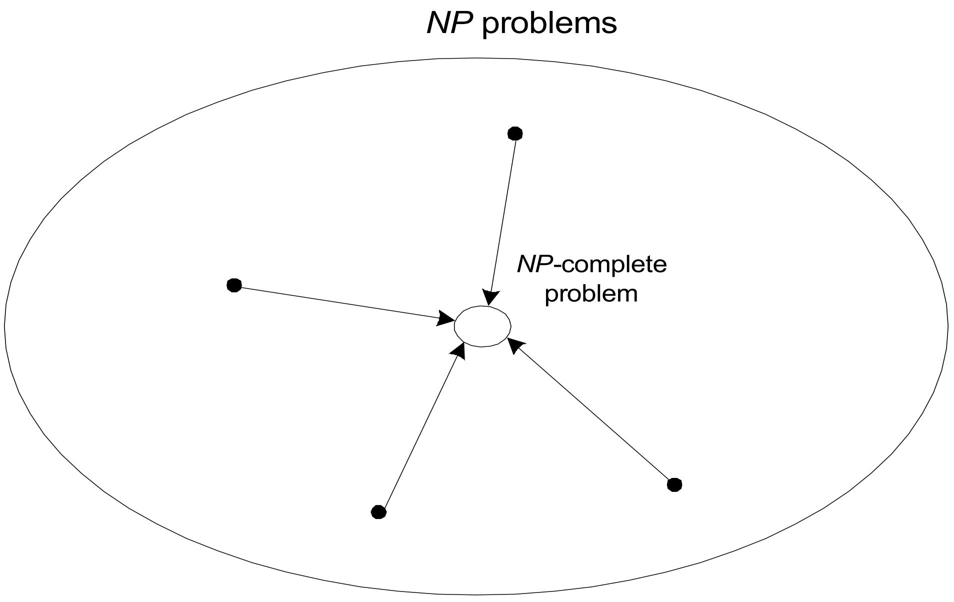


- Cook's theorem (1971): CNF-sat is NP-complete

NP-Complete Problems

- A decision problem D is NP-complete if it is as hard as any problem in NP, *i.e.*,

1. D is in NP
2. every problem in NP is polynomial-time reducible to D

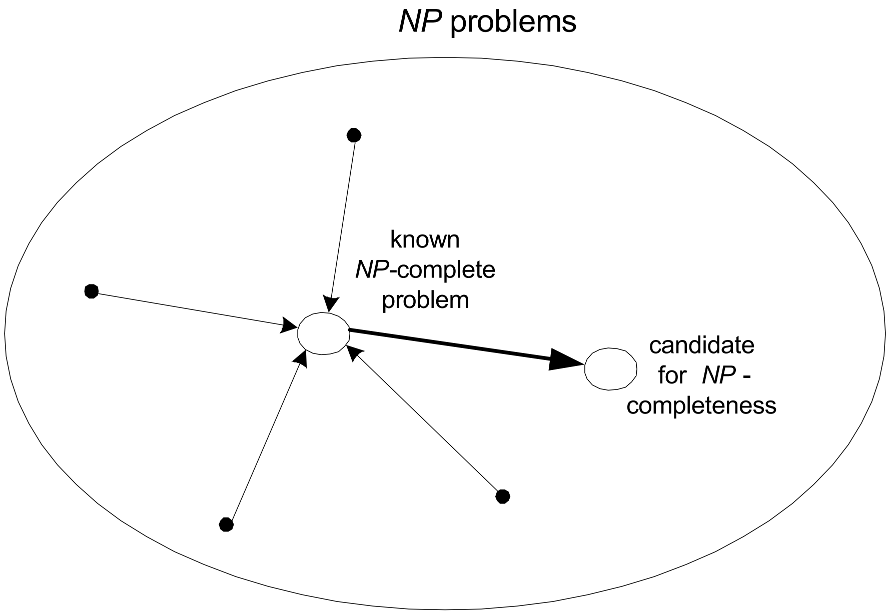


- Cook's theorem (1971): CNF-sat is NP-complete

  (Also know as Cook-Levin Theorem)

NP-Complete Problems (cont.)

- Other NP-complete problems obtained through polynomial-time reductions from a known NP-complete problem



- Examples: TSP, knapsack, partition, graph-coloring and hundreds of other problems of combinatorial nature

Knapsack?

- Didn't we solve this by Dynamic Programming?
- For a knapsack of capacity W, and n items, how big is the table?
- What's the efficiency of the Dynamic Programming Algorithm?

A. $O(n)$
B. $O(W)$
C. $O(nW)$
D. $O(W^n)$
E. None of the above

Knapsack?

- Didn't we solve this by Dynamic Programming?
- For a knapsack of capacity W, and n items, how big is the table? $n \times W$
- What's the efficiency of the Dynamic Programming Algorithm?

A. $O(n)$
B. $O(W)$
C. $O(nW)$
D. $O(W^n)$
E. None of the above

Knapsack?

- Complexity of Dynamic Programming algorithm is in $O(nW)$
- So why is Knapsack in NP?

DEFINITION 1 We say that an algorithm solves a problem in polynomial time if its worst-case time efficiency belongs to $O(p(n))$ where $p(n)$ is a polynomial of the problem's input size n. [Levitin, p. 401]

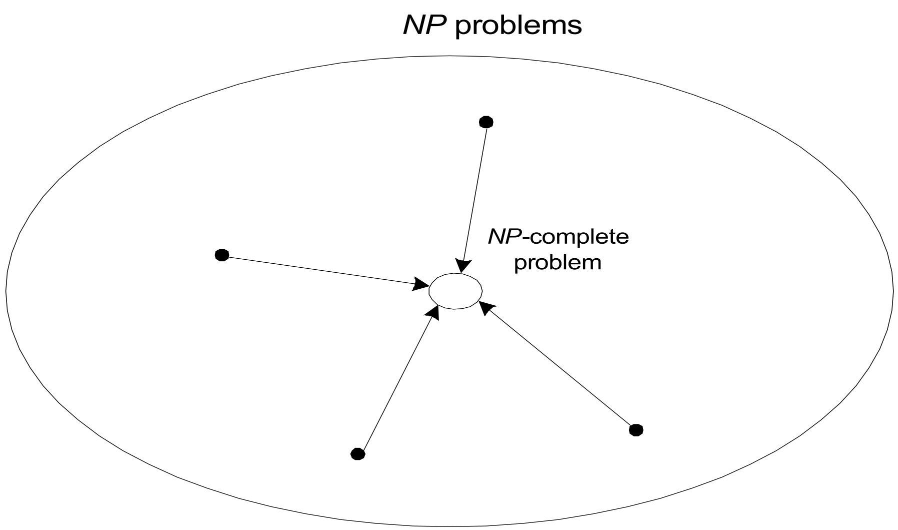
Knapsack?

- Complexity of Dynamic Programming algorithm is in $O(nW)$
- So why is Knapsack in NP?

DEFINITION 1 We say that an algorithm solves a problem in polynomial time if its worst-case time efficiency belongs to $O(p(n))$ where $p(n)$ is a polynomial of the problem's input size n. [Levitin, p. 401]

P = NP ?

- P = NP would imply that every problem in NP, including all NP-complete problems, could be solved in polynomial time
- If a polynomial-time algorithm for just one NP-complete problem is discovered, then every problem in NP can be solved in polynomial time, i.e., P = NP



- Most (but not all) researchers believe that P $\neq$ NP , *i.e.*, P is a proper subset of NP

The Status of the P Versus NP Problem

It's one of the fundamental mathematical problems of our time, and its importance grows with the rise of powerful computers.

Lance Fortnow

Communications of the ACM
Vol. 52 No. 9, Pages 78-86

November 2009

COMMUNICATIONS OF THE ACM

On P, NP, and Computational Complexity

Moshe Y. Vardi

Communications of the ACM
Vol. 53 No. 11, Page 5
10.1145/1839676.1839677

The second week of August was an exciting week. On Friday, August 6, Vinay Deolalikar announced a claimed proof that **P ≠ NP**. Slashdotted blogs broke the news on August 7 and 8, and suddenly the whole world was paying attention. Richard Lipton's August 15 blog entry at blog@CACM was viewed by about 10,000 readers within a week. Hundreds of computer scientists and mathematicians, in a massive Web-enabled collaborative effort, dissected the proof in an intense attempt to verify its validity. By the time the *New York Times* published an article on the topic on August 16, major gaps had been identified, and the excitement was starting to subside. The **P** vs. **NP** problem withstood another challenge and remained wide open.

During and following that exciting week many people have asked me to explain the problem and why it is so important to computer science. "If everyone believes that **P** is different than **NP**," I was asked, "why it is so important to prove the claim?" The answer, of course, is that believing is not the same as knowing. The conventional "wisdom" can be wrong. While our intuition does tell us that finding solutions ought to be more difficult than checking solutions, which is what the **P** vs. **NP** problem is about, intuition can be a poor guide to the truth. Case in point: modern physics.

Problem

Problem

A certain problem can be solved by an algorithm whose running time is in $O(n^{\lg n})$. Which of the following assertions is true?

A. The problem is tractable.

B. The problem is intractable.

C. It's impossible to tell.

Problem

A certain problem can be solved by an algorithm whose running time is in $O(n^{\lg n})$. Which of the following assertions is true?

A. The problem is tractable.

B. The problem is intractable.

C. It's impossible to tell.

Hint: First, decide whether $n^{\lg n}$ is polynomial

Problem

- Give examples of the following graphs or explain why such examples cannot exist:

(a) graph with a Hamiltonian circuit but without an Eulerian circuit

(b) graph with an Eulerian circuit but without a Hamiltonian circuit

(c) graph with both a Hamiltonian circuit and an Eulerian circuit

(d) graph with a cycle that includes all the vertices but with neither a Hamiltonian circuit nor an Eulerian circuit

Unsolvable (by computer) Problem

- Suppose that you could write a program

```
boolean halts(Program p, Input i);
```

that returns **`true`** if **`p`** halts on input **`i`**, and **`false`** if it doesn't.

- Then I can write

```
boolean loopIfHalts(Program p, Input i) {
    if (halts(p,i))
        while (true) ;
    else
        return true;
}
```

which loops if **`p`** halts on input **`i`**, and **`true`** if it doesn't

- And I can write

```
boolean testSelf(Program p) {
    return loopIfHalts(p,p);
}
```

which loops if `p` halts on `p`, and answers `true` if `p` loops.

- What does `testSelf(testSelf)` do?
  - suppose that it returns true?
  - suppose that it loops?

- And I can write

```
boolean testSelf(Program p) {
    return loopIfHalts(p,p);
}
```

which loops if `p` halts on `p`, and answers `true` if `p` loops.

- What does `testSelf(testSelf)` do?
  - suppose that it returns true?
  - suppose that it loops?

A contradiction!

- And I can write

```
boolean testSelf(Program p) {
    return loopIfHalts(p,p);
}
```

which loops if `p` halts on `p`, and answers `true` if `p` loops.

- What does `testSelf(testSelf)` do?
    - suppose that it returns true?
    - suppose that it loops?

A contradiction!

Therefore, no program `halts` can exist.