

P versus **NP**, **NP** hard and
NP complete

Admin

- Assignment 6 is due tonight
- Assignment 7 has been released:
 - Typically would be due next Wed (April 21)
 - But....
 - April 21 and 22 are health days ! Woohoo!
 - Please use them to relax and rest
- As a result, is due next-to-next Wed (April 28), and is a bit longer
- Highly encourage you to divide it into two parts
 - Do questions 1-3 before health days, 4-6 after

Reminders

BECOME A TEACHING ASSISTANT OR TUTOR FOR COMPUTER SCIENCE

This is a great opportunity to teach other students and to work closely with faculty- it can be a very rewarding experience.

Application can be found here: <https://csci.williams.edu/tatutor-application/>

**APPLICATIONS DUE
APRIL 26, 202**

Reminders

- Fill out the **CSCI 256 TA survey** !
 - <https://forms.gle/cmUQdkZQbFaGzWWe6> (you must have received an email from Lauren with link)



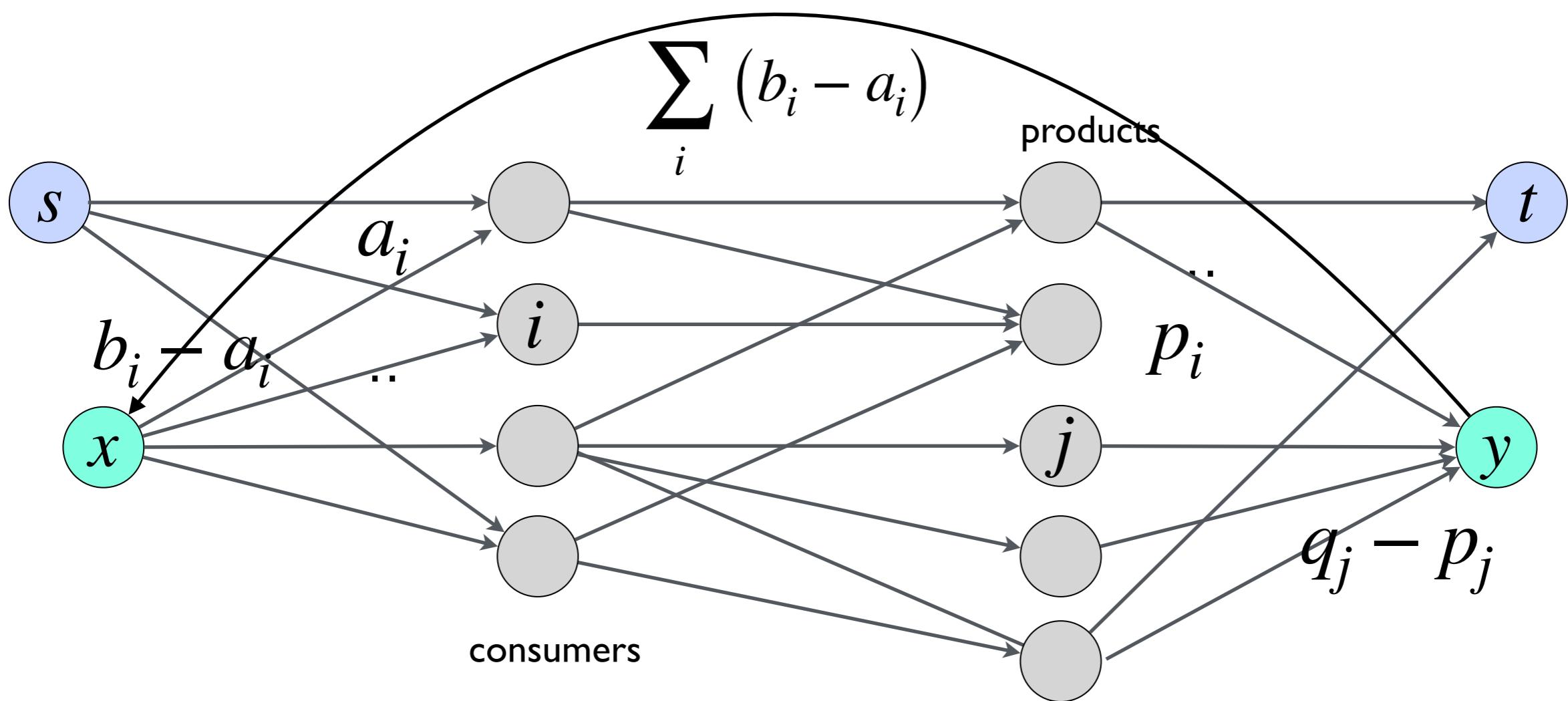
Reminders

- **Colloquium tonight** 7pm: Conversation about work at the intersection of art, music, and computer science



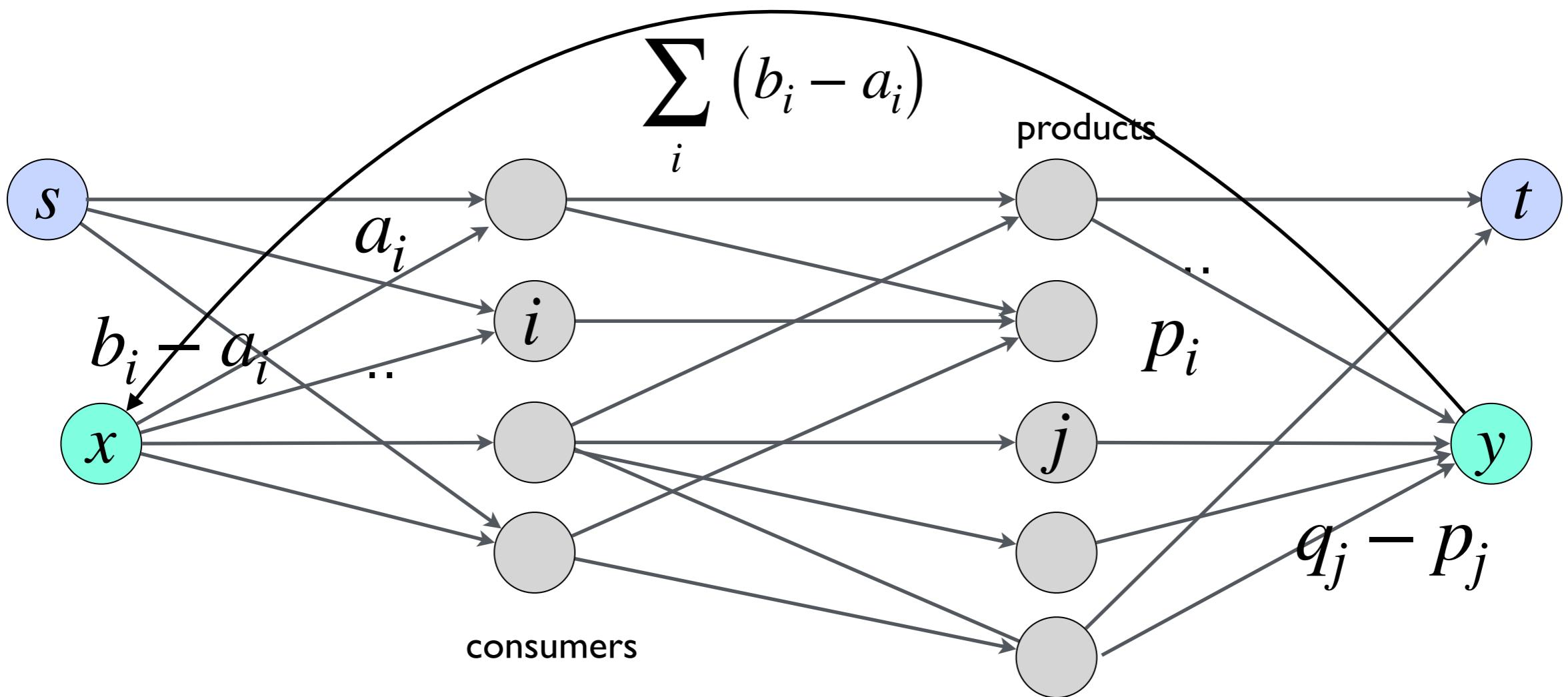
Reduction from Last Lecture

- Instead of going over the correctness today, created a handout
- Serves the additional goal of showing you how to structure your reductions



Circulations & Lower Bounds

- The textbook handles reductions with lower bounds using the concept of "circulations with demands"
- We avoided this framework to do a more direct reduction; but that's why we may have "circulation" that may not originate at s



Today: Overview and
Classes P and NP

What We Focused on So Far

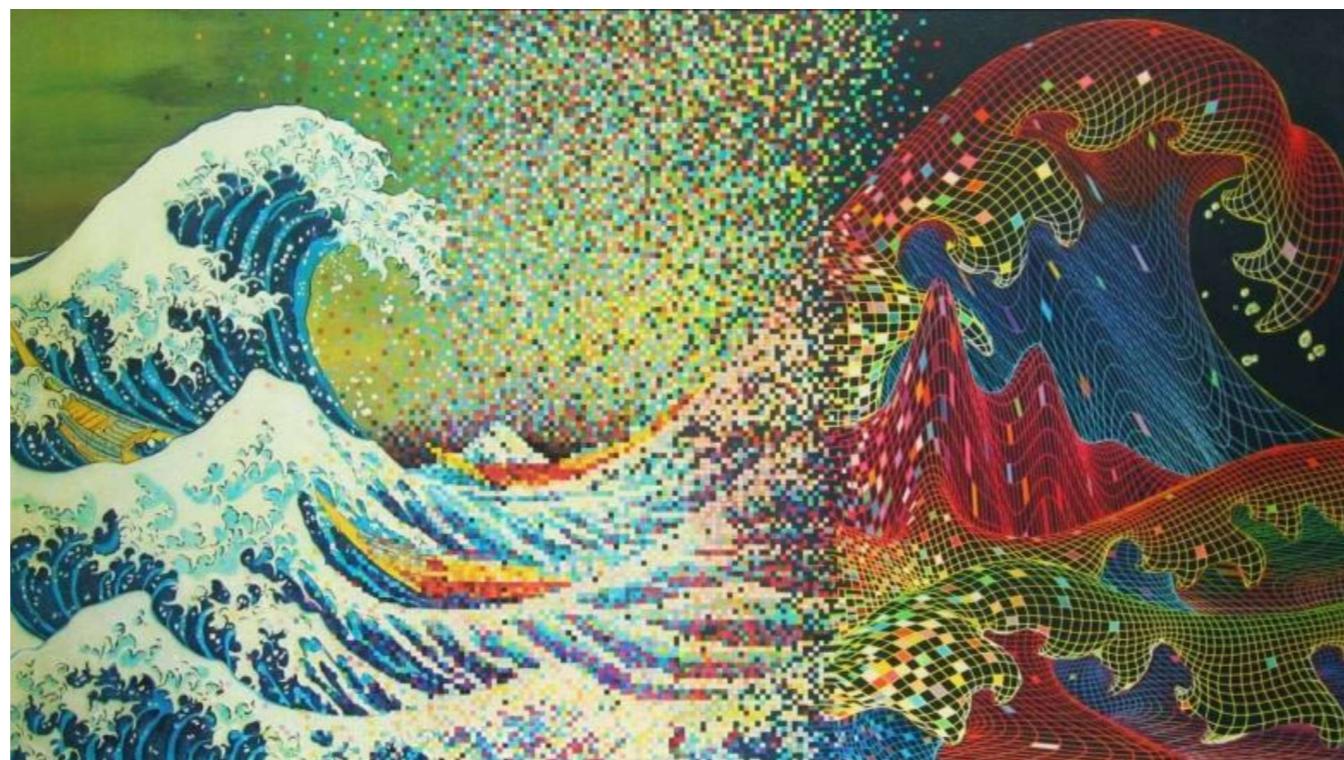
- Number of inversions—can solve in $O(n^2)$ easily, but $O(n \log n)$ with a clever algorithm
- Weighted interval scheduling—can try all combinations in $O(2^n)$, but can solve in $O(n^2)$ using Dynamic programming
- Network flow seems very difficult to solve, but we saw how to solve it in $O(nmC)$ and even $O(n^2m)$

Shifting Focus

- Most of the class has been about how to efficiently solve problems
- Now we're going to shift to a higher-level question
 - What problems can a computer solve efficiently?
 - What problem can a computer not solve efficiently?

Efficiency: Polynomial time

- What problems can a computer solve in polynomial time?
- What problems can a computer (probably) **not** solve in polynomial time?



Technical Setup

- We will now focus on **decision problems** — problems with a yes or no answer
 - “Does this DAG have a topological order?”
 - Is this graph bipartite?
 - Do these two strings have Edit Distance at most 10?
 - Does this flow network have a max flow of at least 20?

Technical Setup

- Most problems have a decision analog
 - Find the flow of this network -> “does this network have flow at least k ?”
 - Find the optimal schedule of these intervals -> “can we schedule at least k intervals?”
- These are (essentially) the same—after all, can always binary search for the optimal value

Technical Setup

- Decision problem means that every solution is “yes” or “no”
- Yes instances can represented as a set of inputs A
 - $x \in A$ means that the solution to x is “yes”
 - $x \notin A$ means that the solution to x is “no”
- So can have (for example): A is the set of all flow networks which permit flow at least k
- Or can have: A is the set of all pairs of strings (a, b) where the edit distance between a and b is at most k

Class P

- P: the class of decision problems that can be solved in polynomial time
 - Edit distance is in **P**
 - Max flow is in **P**
 - Bipartite matching is in **P**
 - Knapsack?
 - dynamic programming algorithm we saw is pseudo-polynomial! So we don't know yet

Class NP

Class NP—Intuition

- **NP** is the class of problems that can be *verified* in polynomial time
- If I give you helpful information, say a proposed solution, you can easily check that it is correct

Class NP—Intuition

8								
	3	6						
7		9	2					
5			7					
	4	5	7					
	1			3				
1				6	8			
	8	5			1			
9				4				

A114473 (C) Artoinkala www.aisudoku.com



5	3	4	6	7	8	9	1	2
6	7	2	1	9	5	3	4	8
1	9	8	3	4	2	5	6	7
8	5	9	7	6	1	4	2	3
4	2	6	8	5	3	7	9	1
7	1	3	9	2	4	8	5	6
9	6	1	5	3	7	2	8	4
2	8	7	4	1	9	6	3	5
3	4	5	2	8	6	1	7	9

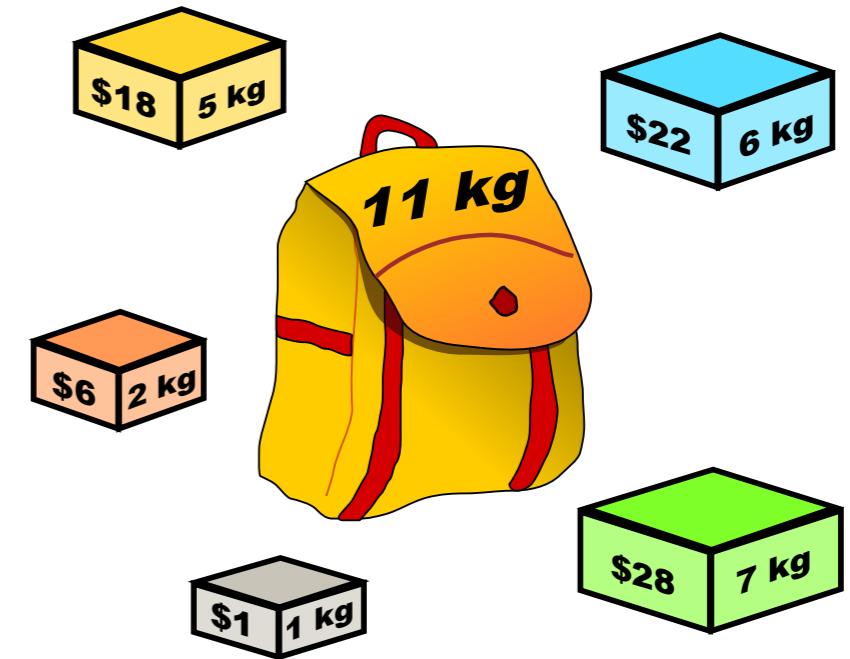
Sudoku is easy if I give you information (by giving you the solution). So sudoku is in **NP**

Class NP—Intuition

- Example (Knapsack capacity $C = 11$)
 - $\{3, 4\}$ has value \$40 (and weight 11)

i	v_i	w_i
1	\$1	1 kg
2	\$6	2 kg
3	\$18	5 kg
4	\$22	6 kg
5	\$28	7 kg

**knapsack instance
(weight limit $W = 11$)**



Knapsack is easy if I give you information (by giving you the solution). So knapsack is in NP

Class NP: Formally

Definition. Algorithm $V(s, c)$ is a verifier for problem X if for every string $s \in X$ iff there exists a certificate, a string c , such that $V(s, c) = \text{yes}$ iff $s \in X$.

Definition. $\text{NP} =$ set of decision problems for which there exists a polynomial-time verifier

- $V(s, c)$ is a polynomial time algorithm
- Certificate c is of polynomial size:
 - $|c| \leq p(|s|)$ for some polynomial $p(\cdot)$

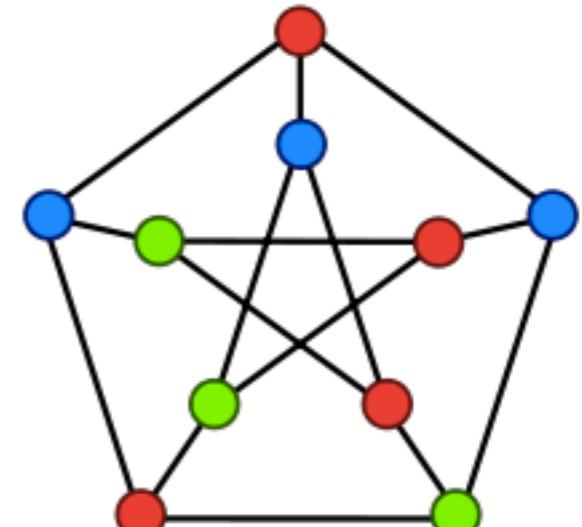
Quick Question

- Is $P \subseteq NP$?
 - If a problem is in P , does that mean that it is in NP ?
- Yes! If a problem can be solved in polynomial time, it can be verified in polynomial time.
- Just solve directly (Can just set $c = ""$)

Graph-Coloring \in NP

Graph-Coloring. Given a graph $G = (V, E)$, is it possible to color the vertices of G using only three colors, such that no edge has both end points colored with the same color.

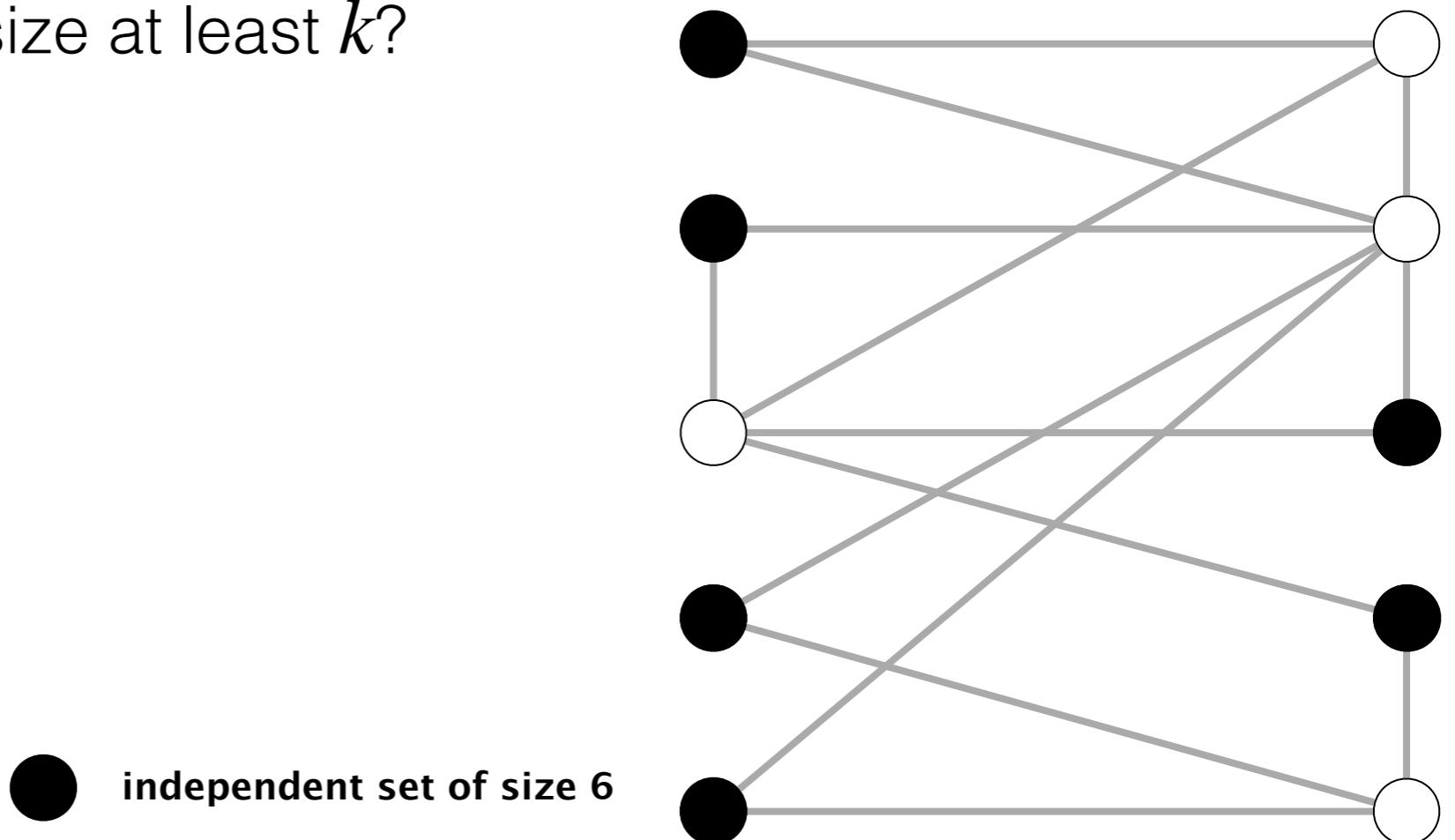
- Graph-Coloring \in NP
 - **Certificate:** assignment of colors to vertices
 - **Poly-time verifier:** check if at most 3 colors used, check for each edge if ends points same color or not



A 3-colorable graph

Independent Set

- Given a graph $G = (V, E)$, an independent set is a subset of vertices $S \subseteq V$ such that no two of them are adjacent, that is, for any $x, y \in S$, $(x, y) \notin E$
- IND-SET Problem.**
Given a graph $G = (V, E)$ and an integer k , does G have an independent set of size at least k ?



IND-SET \in NP

- Given a graph $G = (V, E)$, an independent set is a subset of vertices $S \subseteq V$ such that no two of them are adjacent, that is, for any $x, y \in S$, $(x, y) \notin E$
- IND-SET Problem.** Given a graph $G = (V, E)$ and an integer k , does G have an independent set of size at least k ?
- IND-SET \in NP.**
 - Certificate:** a subset of vertices
 - Poly-time verifier:** check if any two vertices are adjacent and check if size is at least k

Satisfiability

- The next problem is the classic example of a problem in **NP**
 - (and, as we'll soon see, probably not in **P**)
- Many different small variations on the same problem (we'll see a couple)
- **Idea:** given a logical equation, can we assign “true” and “false” to the variables to satisfy the equation?

SAT, 3SAT \in NP

- **SAT.** Given a CNF formula ϕ , does it have a satisfying truth assignment?
- **3SAT.** A SAT formula where each clause contains exactly 3 literals (corresponding to different variables)
- $\phi = (\overline{x_1} \vee x_2, \vee x_3) \wedge (x_1, \overline{x_2} \vee x_3) \wedge (\overline{x_1} \vee x_2 \vee x_4)$
- Satisfying instance: $x_1 = 1, x_2 = 1, x_3 = 0, x_4 = 0$, where 1 : true, 0 : false
- SAT, 3-SAT \in NP
 - Certificate: truth assignment to variables
 - Poly-time verifier: check if assignment evaluates to true

P versus NP

P vs NP

- We know that every problem in P is also in NP
- What about the reverse? That is to say:
 - If a problem can be efficiently *verified*, does that mean it can be efficiently solved in the first place?
 - Or, do there exist problems that can be verified quickly that are *impossible* to solve quickly?

Why Do We Care?

- If $P = NP$, the consequences:
 - Lots of important problems can be solved quickly!
 - Can build things better, faster, more efficiently
 - (Public key) cryptography does not exist
- If $P \neq NP$:
 - Many problems can't be solved quickly
 - Can stop trying to solve them
 - We believe this to be true

Million Dollar Question: P vs NP

- The biggest open problem in computer science
- One of the biggest in math as well
- We are not even close to solving it!

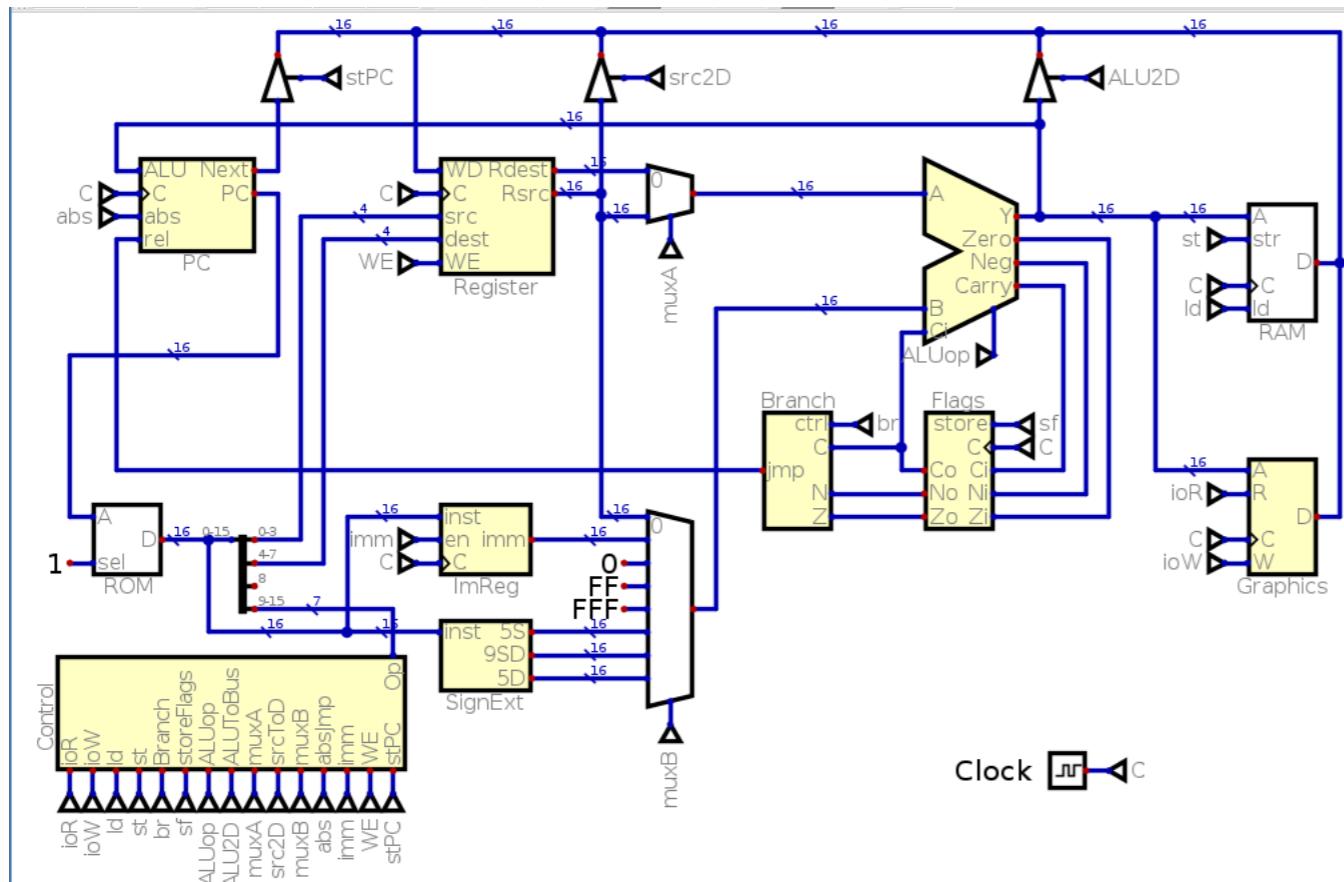
NP-hard and NP-Complete Problems

Cook-Levin Theorem

- If **SAT** can be solved in polynomial time, then *any* problem in **NP** can be solved in polynomial time
- So if **SAT** can be solved in polynomial time, then **P = NP**
- How is this possible?

Cook-Levin Theorem

- Idea: any computer program can be represented by a circuit.
- Solve **SAT** in poly time -> can figure out the answer given by the circuit for NP problem in poly time



You'll see the
proof in CS 361

NP-Hard Problems

- A problem X is **NP-hard** if:
 - If X can be solved in polynomial time, then any problem in **NP** can be solved in polynomial time
 - That is, if X can be solved in polynomial time, then $\mathbf{P} = \mathbf{NP}$

What Does This Mean?

- We think that, probably, $P \neq NP$
- So if a problem is **NP-hard**, then you probably cannot obtain a polynomial-time algorithm for it

Classifying Problems as Hard

- We are frustratingly unable to prove a lot of problems are **impossible** to solve efficiently
- Instead, we say problem X is likely very hard to solve by saying, *if a polynomial-time algorithm was found for X , then something we all believe is impossible will happen*
- We say X is **NP-hard** \Rightarrow if $X \in P$, then $P = NP$



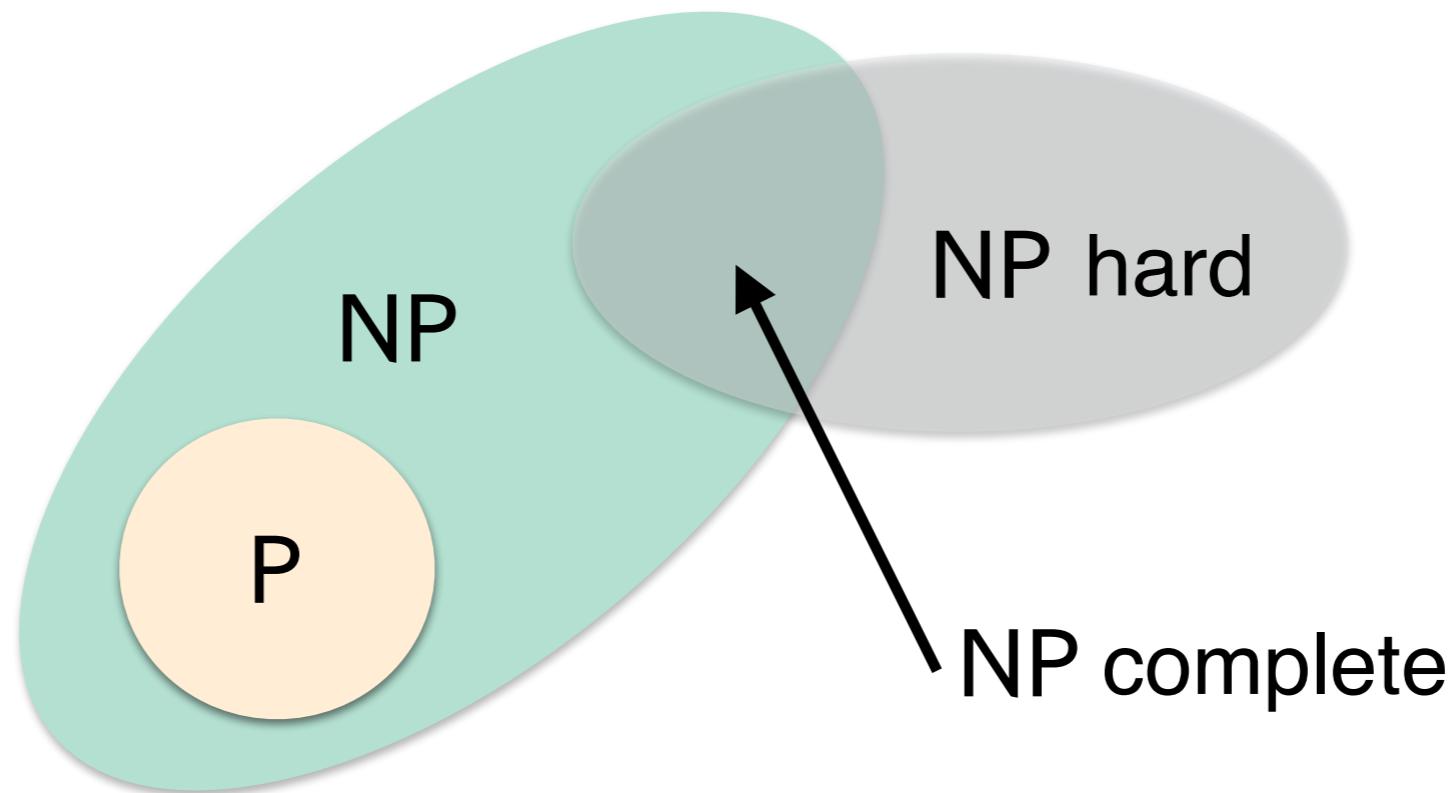
Classifying Problems as Hard

- Instead, we say problem X is likely very hard to solve by saying, *if a polynomial-time algorithm was found for X , then something we all believe is impossible will happen*
- We say X is **NP-hard** \Leftrightarrow if $X \in P$, then $P = NP$
- (Erickson) Calling a problem **NP** hard is like saying, “*If I own a dog, then it can speak fluent English*”
 - You probably don’t know whether or not I own a dog, but you are definitely sure I don’t own *a talking dog*
 - Corollary: No one should believe that I own a dog
 - If a problem is **NP** hard, no one should believe it can be solved in polynomial time



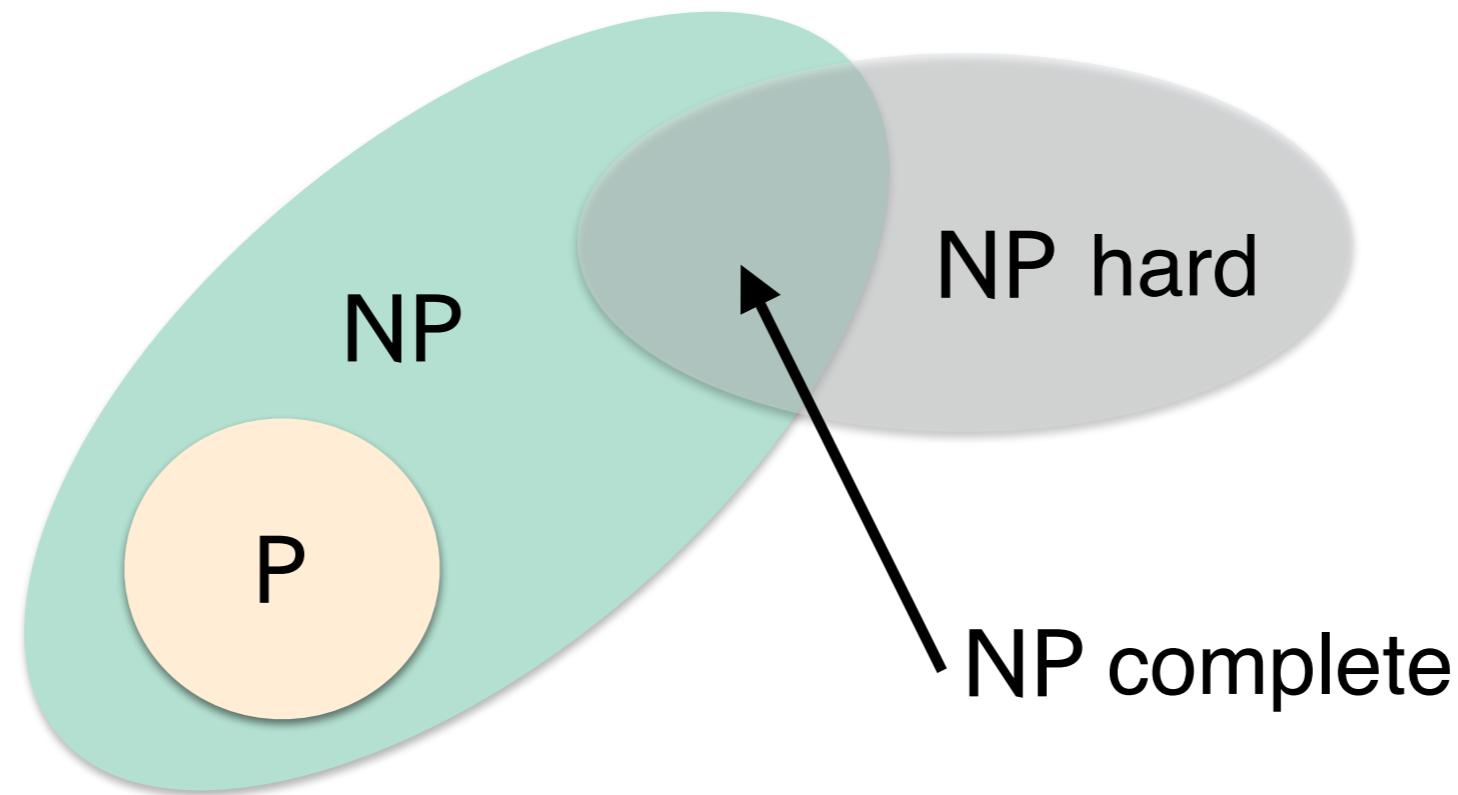
NP Completeness

- **Definition.** A problem X is **NP complete** if X is NP hard and $X \in \text{NP}$
- SAT is **NP complete**
 - $\text{SAT} \in \text{NP}$: given an assignment to input gates (certificate), can verify whether output is one or zero in poly-time
 - SAT is **NP hard** (Cook-Levin Theorem)



Summary

- X is NP-hard \Leftrightarrow if $X \in P$, then $P = NP$
- A problem X is NP complete if X is NP hard and $X \in NP$
- Alternate definition of NP hard:
 - X is NP hard if all languages in NP reduce it to in polynomial time
- Thus, NP-complete problems are the hardest problems in NP



Relative Hardness

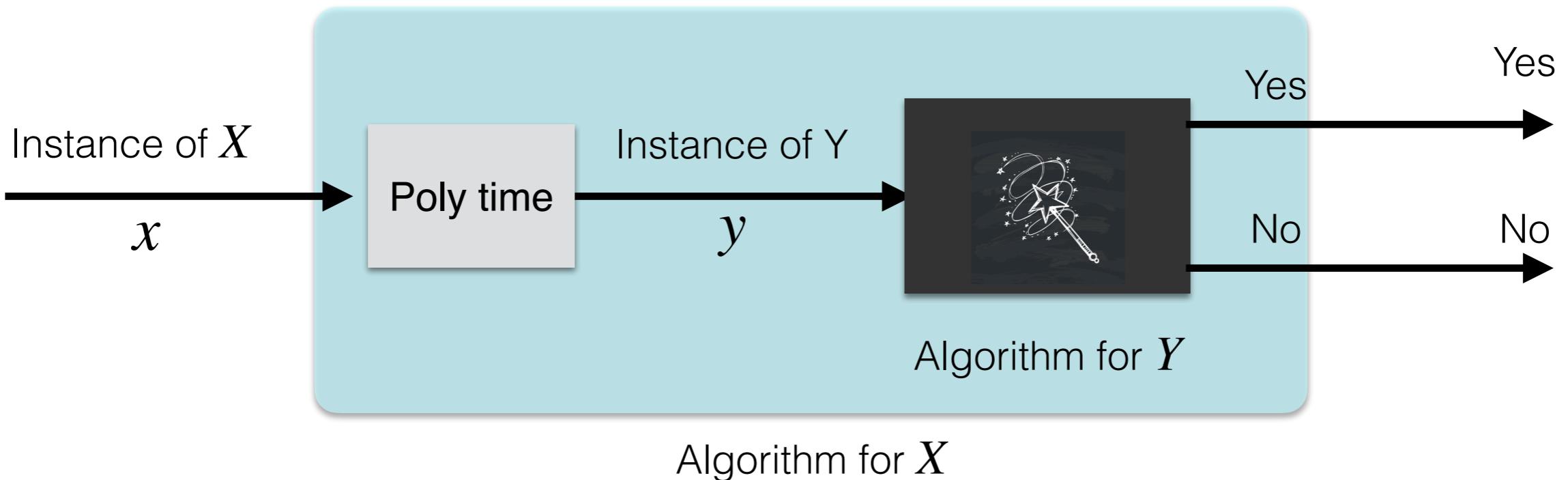
- Suppose we know problem X is NP hard, how can we use that to show problem Y is also hard to solve?
- How do we compare the relative hardness of problems
- Recurring idea in this class: **reductions!**
- Informally, we say a problem X reduces to a problem Y , if can use an algorithm for Y to solve X
 - Bipartite matching reduces to max flow
 - Edge-disjoint paths reduces to max flow

Intuitively, if problem X reduces to problem Y ,
then solving X is no harder than solving Y

[Karp] Reductions

Definition. Decision problem X polynomial-time (Karp) reduces to decision problem Y if given any instance x of X , we can construct an instance y of Y in polynomial time s.t $x \in X$ if and only if $y \in Y$.

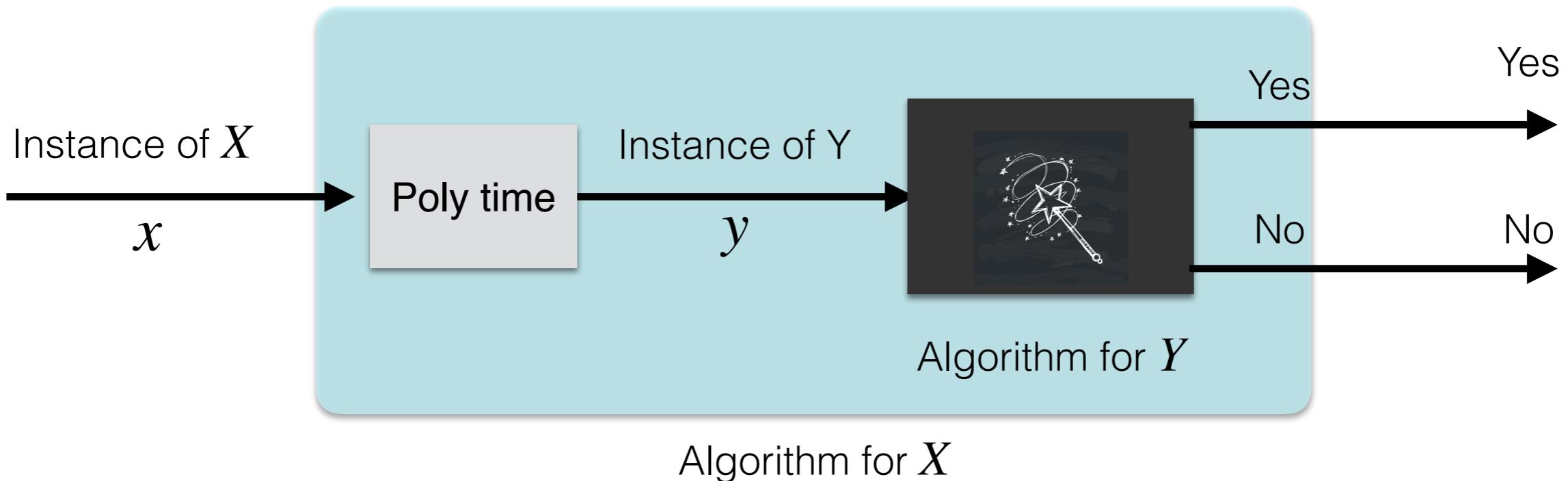
Notation. $X \leq_p Y$



Reductions Quiz

Say $X \leq_p Y$. Which of the following can we infer?

- A. If X can be solved in polynomial time, then so can Y .
- B. X can be solved in poly time iff Y can be solved in poly time.
- C. If X cannot be solved in polynomial time, then neither can Y .
- D. If Y cannot be solved in polynomial time, then neither can X .



Digging Deeper

- Graph 2-Color reduces to Graph 3-color
 - Just replace the third color with either of the two
- Graph 2-Color can be solved in polynomial time
 - How?
 - We can decide if a graph is bipartite in $O(n + m)$ time using traversal
- Graph 3-color (we'll show) is NP hard

Intuitively, if problem X reduces to problem Y ,
then solving X is no harder than solving Y

Use of Reductions: $X \leq_p Y$

Design algorithms:

- If Y can be solved in polynomial time, we know X can also be solved in polynomial time

Establish intractability:

- If we know that X is known to be impossible/hard to solve in polynomial-time, then we can conclude the same about problem Y

Establish Equivalence:

- If $X \leq_p Y$ and $Y \leq_p X$ then X can be solved in poly-time iff Y can be solved in poly time and we use the notation $X \equiv_p Y$

NP hard: Definition

- **New definition of NP hard using reductions.**
 - A problem Y is NP hard, if for any problem $X \in \text{NP}$, $X \leq_p Y$
 - Recall we said Y is NP hard if $Y \in \text{P}$, then $\text{P} = \text{NP}$.
 - Lets show that both definitions are equivalent
 - (\Rightarrow) every problem in **NP** reduces to Y , and if $Y \in \text{P}$, then $\text{P} = \text{NP}$
 - (\Leftarrow) Suppose $Y \in \text{P}$, then $\text{P} = \text{NP}$: which means every problem in **NP**(= P) reduces to Y

Proving NP Hardness

- To prove problem Y is **NP-hard**
 - Difficult to prove every problem in **NP** reduces to Y
 - Instead, we use a known-NP-hard problem Z
 - We know every problem X in **NP**, $X \leq_p Z$
 - Notice that \leq_p is transitive
 - Thus, enough to prove $Z \leq_p Y$

**TO PROVE THAT A PROBLEM Y IS NP HARD,
REDUCE A KNOWN NP HARD PROBLEM Z TO Y**

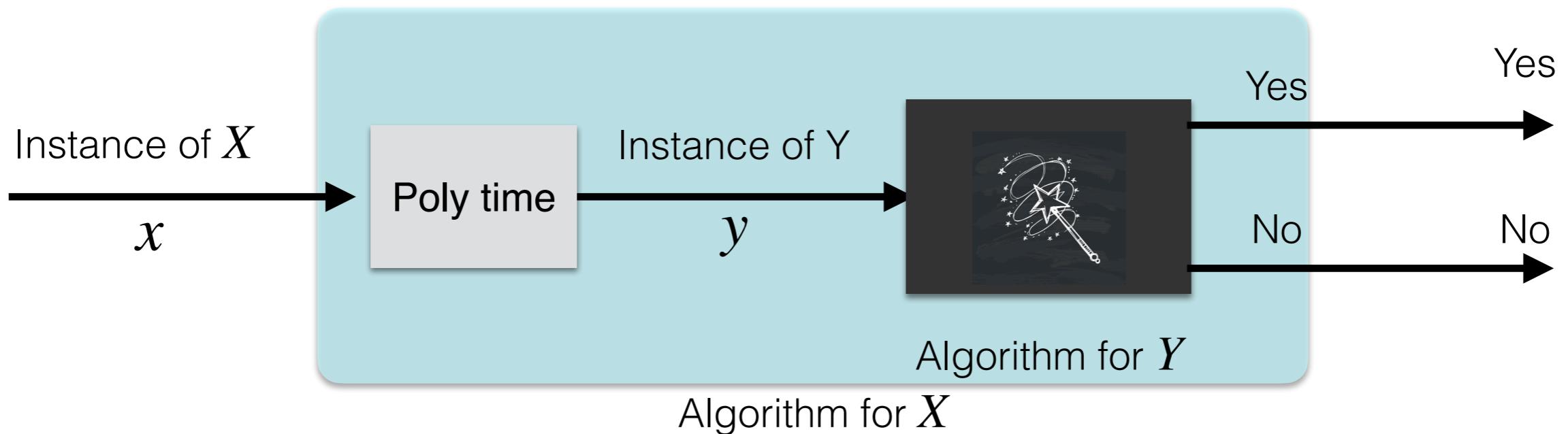
Known NP Hard Problems?

- For now: **SAT** (Cook-Levin Theorem)
- We will prove a whole repertoire of NP hard and NP complete problems by using reductions
- Before reducing SAT to other problems to prove them NP hard, let us practice some easier reductions first

**TO PROVE THAT A PROBLEM Y IS NP HARD,
REDUCE A KNOWN NP HARD PROBLEM Z TO Y**

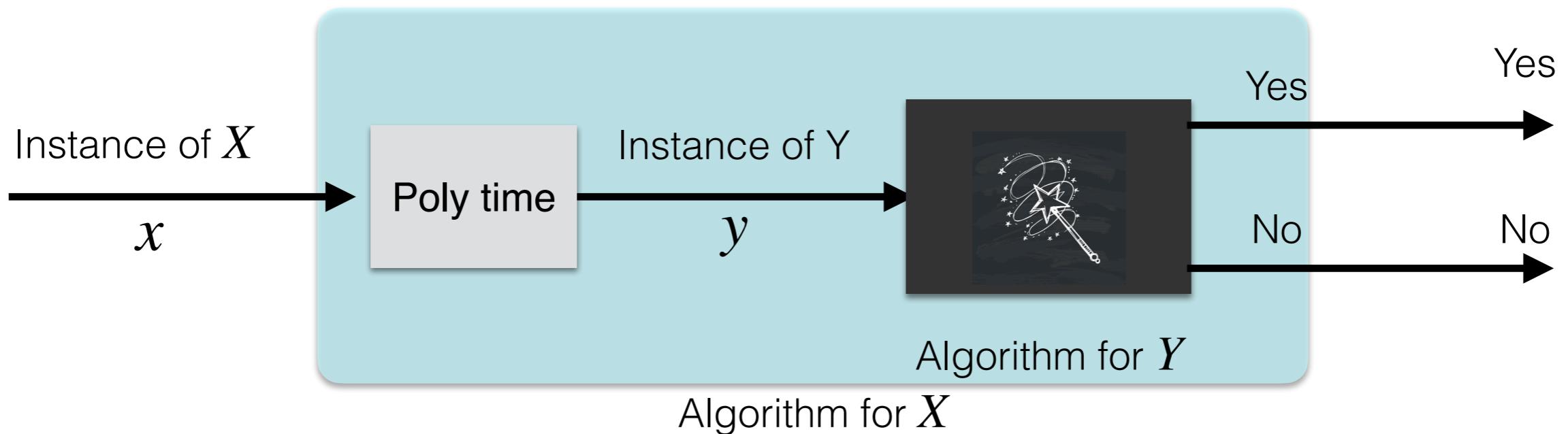
Reductions: General Pattern

- Describe a polynomial-time algorithm to transform an arbitrary instance x of Problem X into a special instance y of Problem Y
- Prove that:
 - If x is a “yes” instance of X , then y is a “yes” instance of Y
 - If y is a “yes” instance of Y , then x is a “yes” instance of X



Reductions: General Pattern

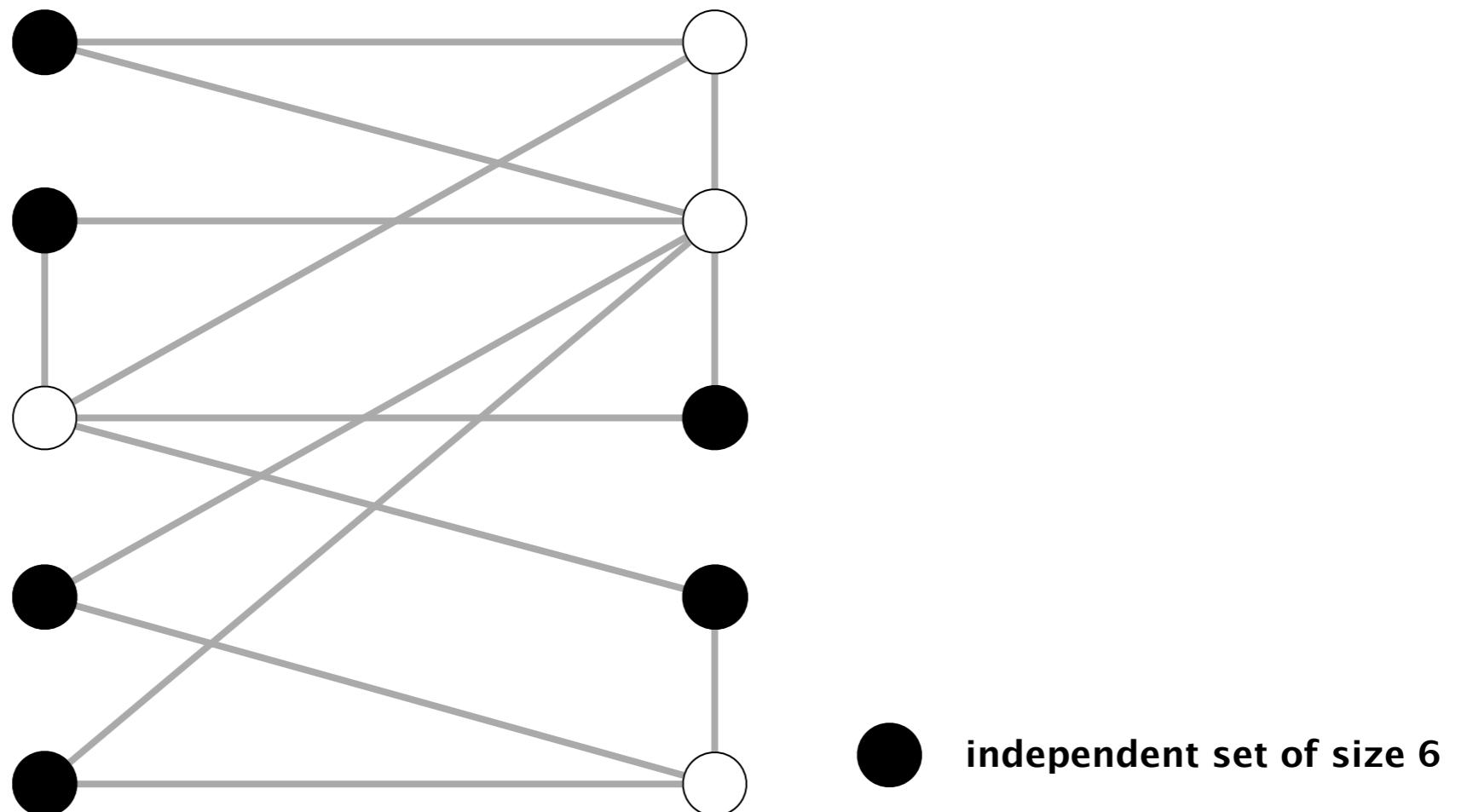
- Describe a polynomial-time algorithm to transform an arbitrary instance x of Problem X into a special instance y of Problem Y
- Notice that correctness of reductions are not symmetric:
 - the “if” proof needs to handle arbitrary instances of X
 - the “only if” needs to handle the special instance of Y



VERTEX-COVER \equiv_p **IND-SET**

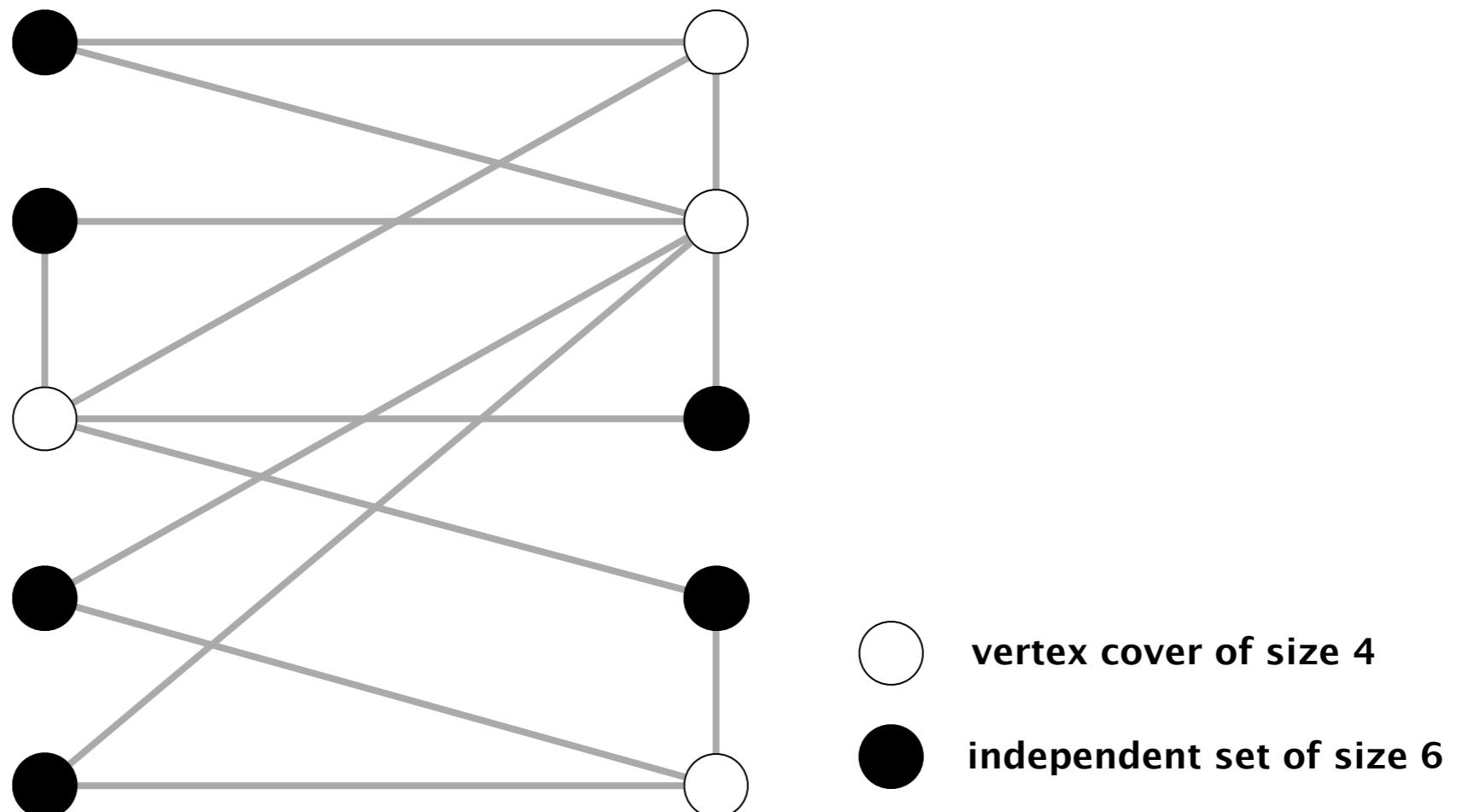
IND-SET

- Given a graph $G = (V, E)$, an independent set is a subset of vertices $S \subseteq V$ such that no two of them are adjacent, that is, for any $x, y \in S$, $(x, y) \notin E$
- IND-SET Problem.** Given a graph $G = (V, E)$ and an integer k , does G have an independent set of size at least k ?



Vertex-Cover

- Given a graph $G = (V, E)$, a vertex cover is a subset of vertices $T \subseteq V$ such that for every edge $e = (u, v) \in E$, either $u \in T$ or $v \in T$.
- VERTEX-COVER Problem.** Given a graph $G = (V, E)$ and an integer k , does G have a vertex cover of size at most k ?



Our First Reduction

- **VERTEX-COVER** \leq_p **IND-SET**
 - Suppose we know how to solve independent set, can we use it to solve vertex cover?
- **Claim.** S is an independent set of size k iff $V - S$ is a vertex cover of size $n - k$.
- **Proof.** (\Rightarrow) Consider an edge $e = (u, v) \in E$
 - S is independent: u, v both cannot be in S
 - At least one of $u, v \in V - S$
 - $V - S$ covers e ■

Our First Reduction

- **VERTEX-COVER** \leq_p **IND-SET**
 - Suppose we know how to solve independent set, can we use it to solve vertex cover?
- **Claim.** S is an independent set of size k iff $V - S$ is a vertex cover of size $n - k$.
- **Proof.** (\Leftarrow) Consider an edge $e = (u, v) \in E$
 - $V - S$ is a vertex cover: at least one of u, v or both must be in $V - S$
 - Both u, v cannot be in S
 - Thus, S is an independent set. ■

Vertex Cover \equiv_p IND Set

- VERTEX-COVER \leq_p IND-SET
- Reduction. Let $G' = G$, $k' = n - k$.
 - (\Rightarrow) If G has a vertex cover of size at most k then G' has an independent set of size at least k'
 - (\Leftarrow) If G' has an independent set of size at least k' then G has a vertex cover of size at most k
- IND-SET \leq_p VERTEX-COVER
 - Same reduction works: $G' = G$, $k' = n - k$
- VERTEX-COVER \equiv_p IND-SET

P versus NP

Acknowledgments

- Some of the material in these slides are taken from
 - Kleinberg Tardos Slides by Kevin Wayne (<https://www.cs.princeton.edu/~wayne/kleinberg-tardos/pdf/04GreedyAlgorithmsI.pdf>)
 - Jeff Erickson's Algorithms Book (<http://jeffe.cs.illinois.edu/teaching/algorithms/book/Algorithms-JeffE.pdf>)