

# CPS 296.1

Computational problems, algorithms, runtime,  
hardness

(a ridiculously brief introduction to theoretical computer science)

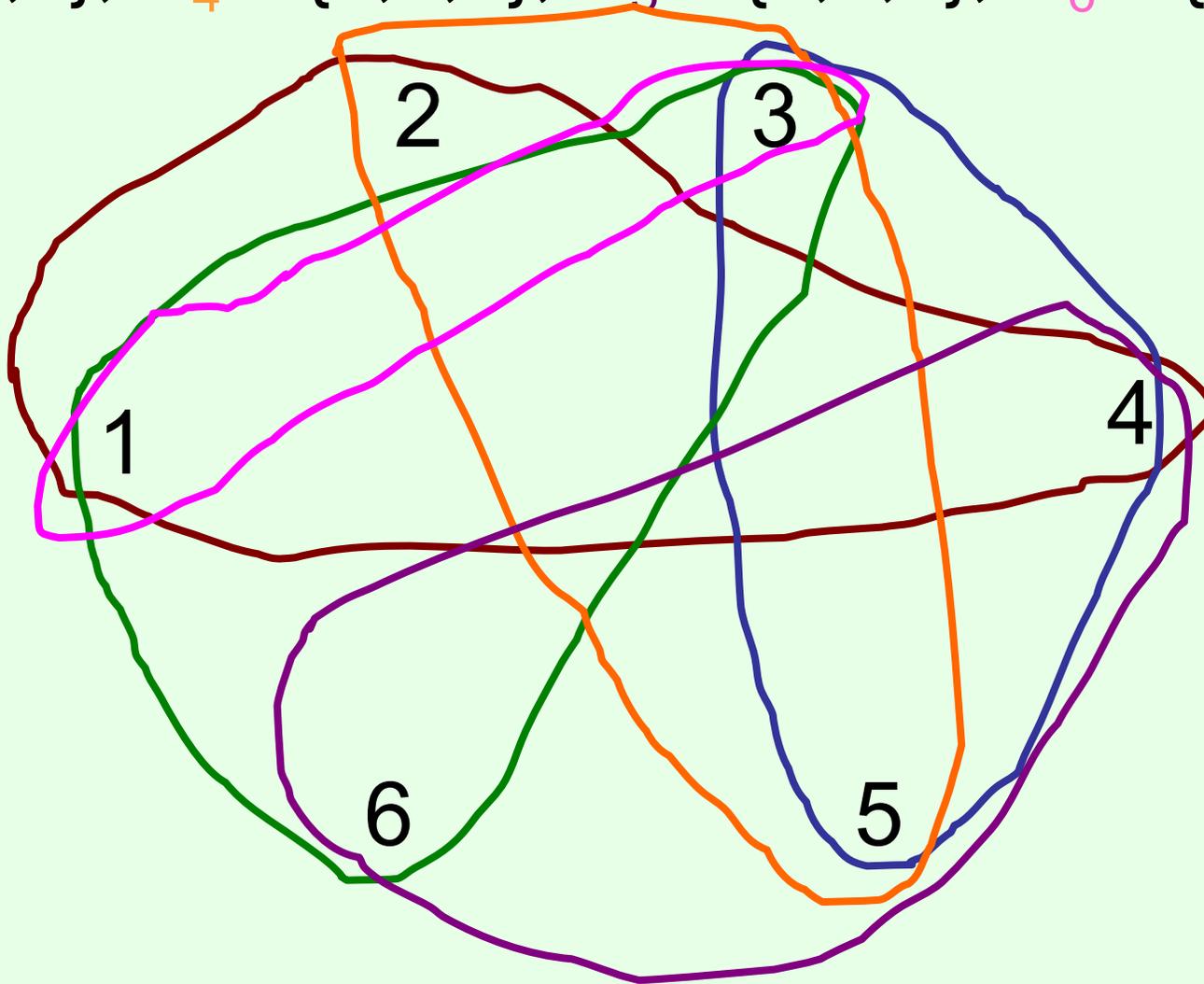
Vincent Conitzer

# Set Cover (a computational problem)

- We are given:
  - A finite set  $S = \{1, \dots, n\}$
  - A collection of subsets of  $S$ :  $S_1, S_2, \dots, S_m$
- We are asked:
  - Find a subset  $T$  of  $\{1, \dots, m\}$  such that  $\bigcup_{j \in T} S_j = S$
  - Minimize  $|T|$
- **Decision variant** of the problem:
  - we are additionally given a target size  $k$ , and
  - asked whether a  $T$  of size at most  $k$  will suffice
- One **instance** of the set cover problem:  
 $S = \{1, \dots, 6\}$ ,  $S_1 = \{1, 2, 4\}$ ,  $S_2 = \{3, 4, 5\}$ ,  $S_3 = \{1, 3, 6\}$ ,  $S_4 = \{2, 3, 5\}$ ,  $S_5 = \{4, 5, 6\}$ ,  $S_6 = \{1, 3\}$

# Visualizing Set Cover

- $S = \{1, \dots, 6\}$ ,  $S_1 = \{1, 2, 4\}$ ,  $S_2 = \{3, 4, 5\}$ ,  $S_3 = \{1, 3, 6\}$ ,  $S_4 = \{2, 3, 5\}$ ,  $S_5 = \{4, 5, 6\}$ ,  $S_6 = \{1, 3\}$



# Using glpsol to solve set cover instances

- How do we model set cover as an integer program?
- See examples

# Algorithms and runtime

- We saw:
  - the **runtime** of glpsol on set cover instances increases rapidly as the instances' sizes increase
  - if we drop the integrality constraint, can scale to larger instances
- Questions:
  - Using glpsol on our integer program **formulation** is but one **algorithm** – maybe other algorithms are faster?
    - different formulation; different optimization package (e.g., CPLEX); simply going through all the combinations one by one; ...
  - What is “fast enough”?
  - Do (mixed) integer programs always take more time to solve than linear programs?
  - Do set cover instances **fundamentally** take a long time to solve?

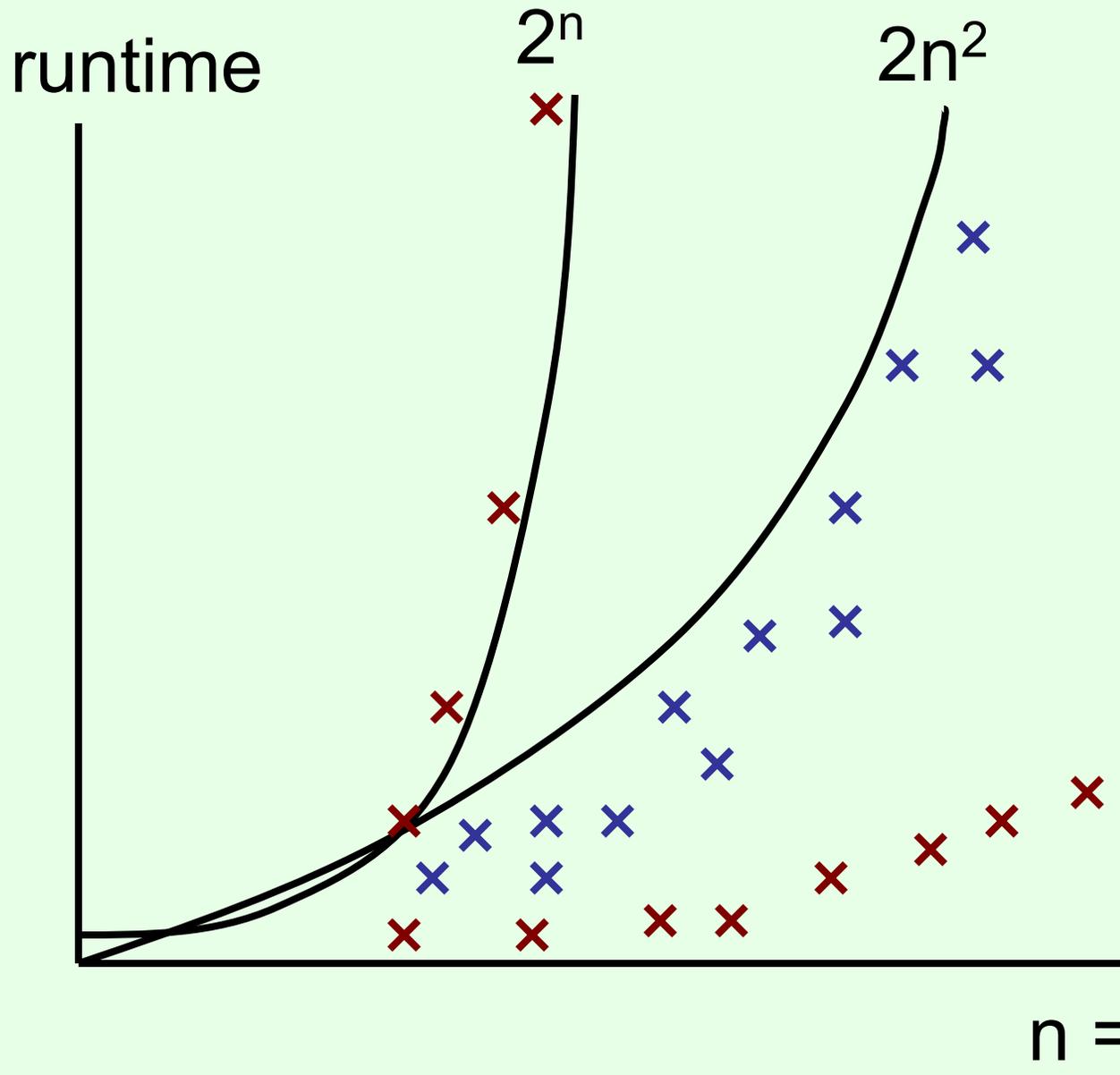
A simpler problem: sorting (see associated spreadsheet)

- Given a list of numbers, sort them
- **(Really) dumb algorithm:** Randomly perturb the numbers. See if they happen to be ordered. If not, randomly perturb the whole list again, etc.
- **Reasonably smart algorithm:** Find the smallest number. List it first. Continue on to the next number, etc.
- **Smart algorithm (MergeSort):**
  - It is easy to merge two lists of numbers, each of which is already sorted, into a single sorted list
  - So: divide the list into two equal parts, sort each part with some method, then merge the two sorted lists into a single sorted list
  - ... actually, to sort each of the parts, we can **again** use MergeSort! (The algorithm “calls itself” as a subroutine. This idea is called *recursion*.) Etc.

# Polynomial time

- Let  $|x|$  be the **size** of problem instance  $x$  (e.g., the size of the file in the .lp language)
- Let  $a$  be an algorithm for the problem
- Suppose that for **any**  $x$ ,  $\text{runtime}(a,x) < cf(|x|)$  for some constant  $c$  and function  $f$   
Then we say algorithm  $a$ 's runtime is  $O(f|x|)$
- $a$  is a **polynomial-time algorithm** if it is  $O(f(|x|))$  for some **polynomial** function  $f$
- **P** is the class of all problems that have at least one polynomial-time algorithm
- Many people consider an algorithm **efficient** if and only if it is polynomial-time

# Two algorithms for a problem



- × run of algorithm 1
- × run of algorithm 2

Algorithm 1 is  $O(n^2)$   
(a polynomial-time algorithm)

Algorithm 2 is not  $O(n^k)$   
for any constant  $k$   
(not a polynomial-time algorithm)

The problem is in P

# Linear programming and (mixed) integer programming

- LP and (M)IP are also computational problems
- LP is in  $P$ 
  - Ironically, the most commonly used LP algorithms are not polynomial-time (but “usually” polynomial time)
- (M)IP is not known to be in  $P$ 
  - Most people consider this unlikely

# Reductions

- Sometimes you can reformulate problem A in terms of problem B (i.e., **reduce** A to B)
  - E.g., we have seen how to formulate several problems as linear programs or integer programs
- In this case problem A is **at most** as hard as problem B
  - Since LP is in P, all problems that we can formulate using LP are in P
  - Caveat: only true if the linear program itself can be created in polynomial time!

# NP (“nondeterministic polynomial time”)

- Recall: **decision problems** require a yes or no answer
- **NP**: the class of all decision problems such that if the answer is yes, there is a simple proof of that
- E.g., “does there exist a set cover of size  $k$ ?”
- If yes, then just show which subsets to choose!
- Technically:
  - The proof must have polynomial length
  - The correctness of the proof must be verifiable in polynomial time

# P vs. NP

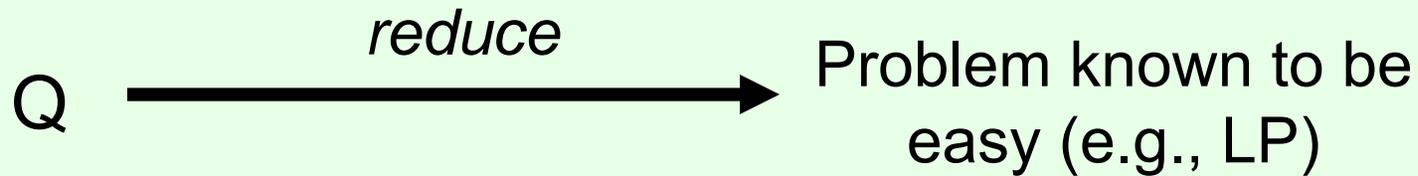
- **Open problem:** is it true that  $P=NP$ ?
- The most important open problem in theoretical computer science (maybe in mathematics?)
- \$1,000,000 Clay Mathematics Institute Prize
- Most people believe  $P$  is not  $NP$
- If  $P$  **were** equal to  $NP$ ...
  - Current cryptographic techniques can be broken in polynomial time
  - Computers can probably solve many difficult mathematical problems...
    - ... including the other Clay Mathematics Institute Prizes! 😊

# NP-hardness

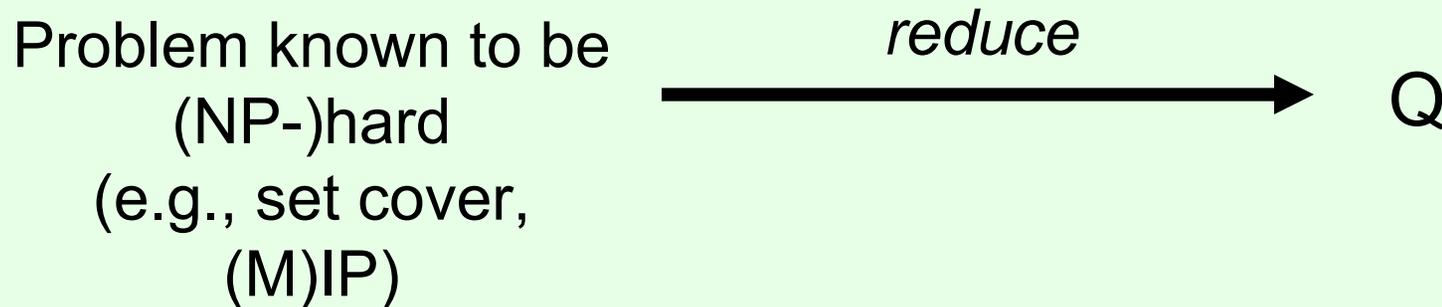
- A problem is **NP-hard** if the following is true:
  - Suppose that it is in P
  - Then  $P=NP$
- So, trying to find a polynomial-time algorithm for it is like trying to prove  $P=NP$
- Set cover is NP-hard
- Typical way to prove problem Q is NP-hard:
  - Take a known NP-hard problem Q'
  - Reduce it to your problem Q
    - (in polynomial time)
- E.g., (M)IP is NP-hard, because we have already reduced set cover to it
  - (M)IP is more general than set cover, so it can't be easier
- A problem is **NP-complete** if it is 1) in NP, and 2) NP-hard

# Reductions:

To show problem Q is easy:



To show problem Q is (NP-)hard:

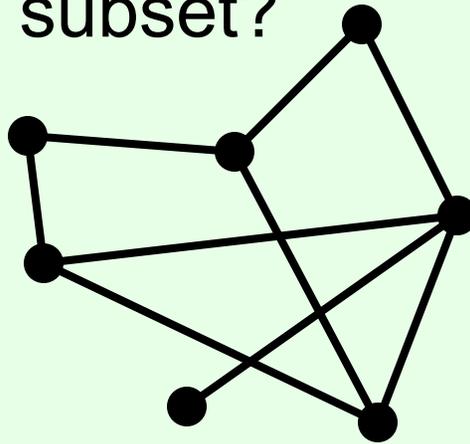


**ABSOLUTELY NOT A PROOF OF NP-HARDNESS:**



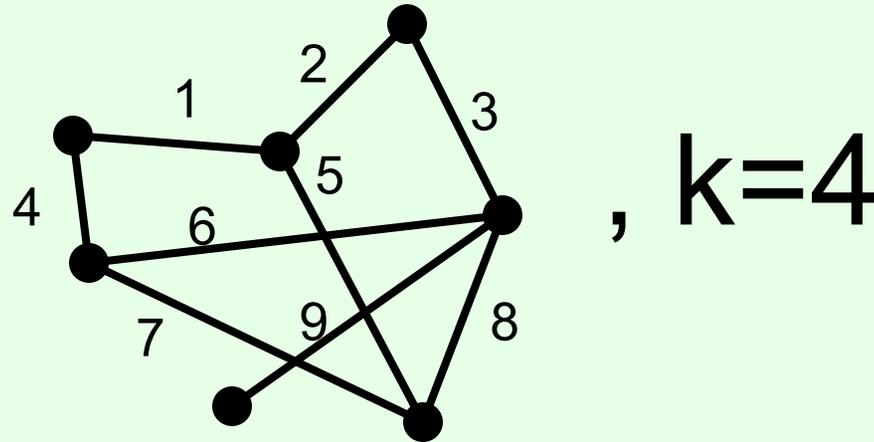
# Independent Set

- In the below graph, does there exist a subset of **vertices**, of size 4, such that there is no **edge** between members of the subset?



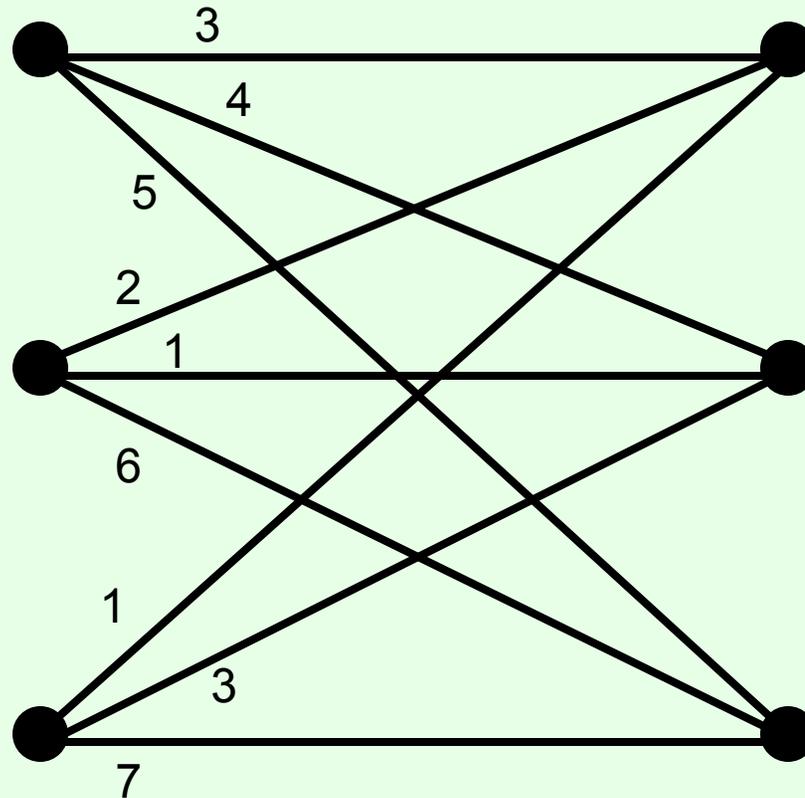
- General problem (decision variant): given a graph and a number  $k$ , are there  $k$  vertices with no edges between them?
- NP-complete

# Reducing independent set to set cover



- In set cover instance (decision variant),
  - let  $S = \{1,2,3,4,5,6,7,8,9\}$  (set of edges),
  - for each vertex let there be a subset with the vertex's adjacent edges:  $\{1,4\}$ ,  $\{1,2,5\}$ ,  $\{2,3\}$ ,  $\{4,6,7\}$ ,  $\{3,6,8,9\}$ ,  $\{9\}$ ,  $\{5,7,8\}$
  - target size = #vertices -  $k = 7 - 4 = 3$
- Claim: answer to both instances is the same (why??)
- So which of the two problems is harder?

# Weighted bipartite matching



- Match each node on the left with one node on the right (can only use each node once)
- Minimize total cost (weights on the chosen edges)

# Weighted bipartite matching...

- minimize  $\sum_{ij} c_{ij} x_{ij}$
- subject to
- for every  $i$ ,  $\sum_j x_{ij} = 1$
- for every  $j$ ,  $\sum_i x_{ij} = 1$
- for every  $i, j$ ,  $x_{ij} \geq 0$
  
- Theorem [Birkhoff-von Neumann]: this linear program always has an optimal solution consisting of just integers
  - and typical LP solving algorithms will return such a solution
  
- So weighted bipartite matching is in P