NP Completeness Approximation Algorithms

"Is this thing
that doesn't look like
another thing, really that
thing?"

Review: Dynamic Programming

- When applicable:
  - Optimal substructure: optimal solution to problem consists of optimal solutions to subproblems
  - Overlapping subproblems: few subproblems in total, many recurring instances of each
  - Basic approach:
    - Build a table of solved subproblems that are used to solve larger ones
    - *What is the difference between memoization and dynamic programming?*
    - *Why might the latter be more efficient?*

Review: Greedy Algorithms

- A *greedy algorithm* always makes the choice that looks best at the moment
  - The hope: a locally optimal choice will lead to a globally optimal solution
  - For some problems, it works
    - Yes: fractional knapsack problem
    - No: 0/1 knapsack problem
- Dynamic programming can be overkill; greedy algorithms tend to be easier to code

Review: The Knapsack Problem

- The *0-1 knapsack problem*:
  - A thief must choose among n items, where the ith item worth v_i dollars and weighs w_i pounds
  - Carrying at most W pounds, maximize value
- A variation, the *fractional knapsack problem*:
  - Thief can take fractions of items
  - Think of items in 0-1 problem as gold ingots, in fractional problem as buckets of gold dust

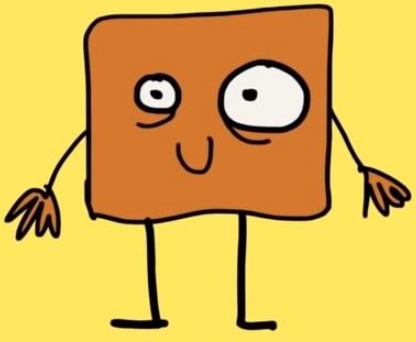
- *Why greedy choice algorithm works for the fractional problem but not for the 0-1 problem?*

NP-Completeness

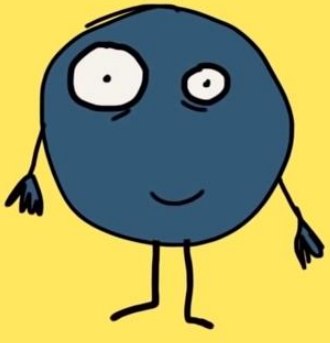
- Some problems are *intractable*:
  as they grow large, we are unable to solve them in reasonable time

- What constitutes reasonable time?
  - Standard working definition: *polynomial time*
  - On an input of size n the worst-case running time is $O(n^k)$ for some constant k
  - Polynomial time: $O(n^2)$, $O(n^3)$, $O(1)$, $O(n \lg n)$
  - Not in polynomial time: $O(2^n)$, $O(n^n)$, $O(n!)$

Matrix Multiplication



Sorting



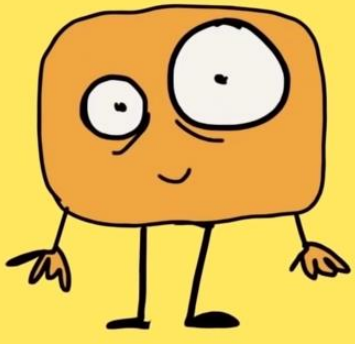
P

Polynomial

n^2 n^5

↑ Number of inputs

Matching

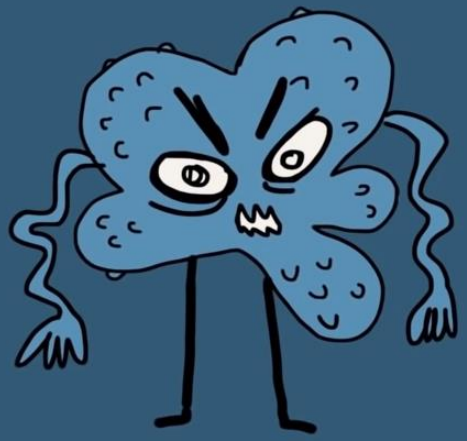


Polynomial-Time Algorithms

- Are some problems solvable in polynomial time?
  - Of course: every algorithm we have studied provides polynomial-time solution to some problem
  - We define **P** to be the class of problems solvable in polynomial time

- Are all problems solvable in polynomial time?
  - No: "Circuit Satisfiability Problem" is not solvable by any computer, no matter how much time is given
  - Such problems are clearly intractable, not in **P**

SAT



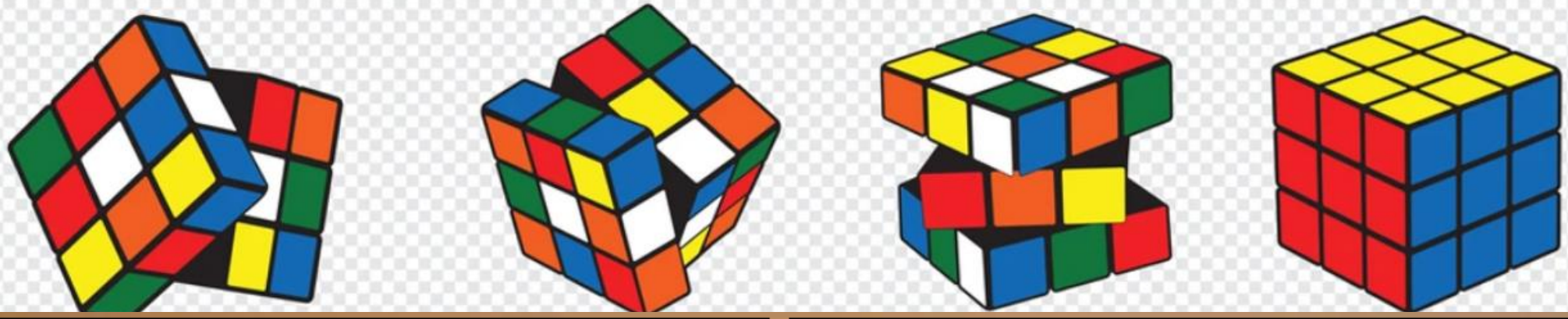
Travelling Salesman

NP

Exponential

2^n

3-Coloring

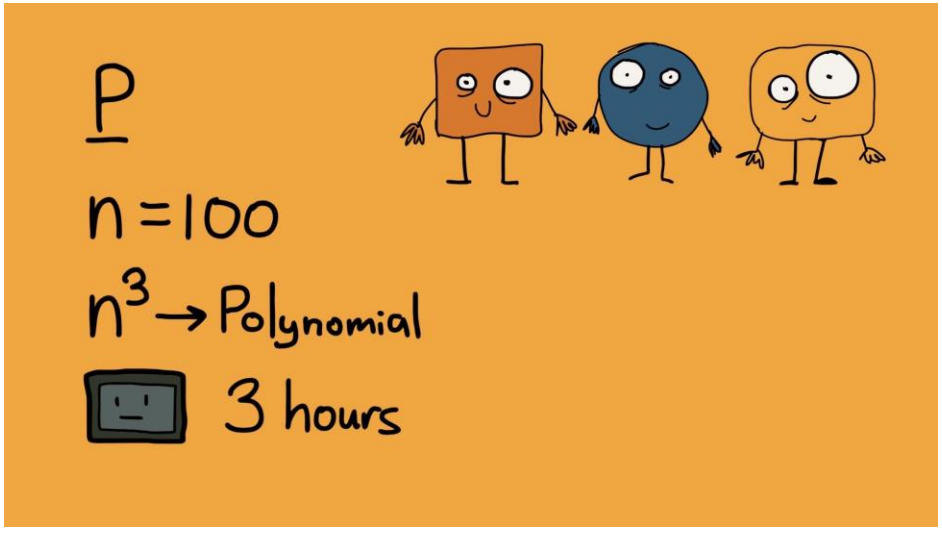


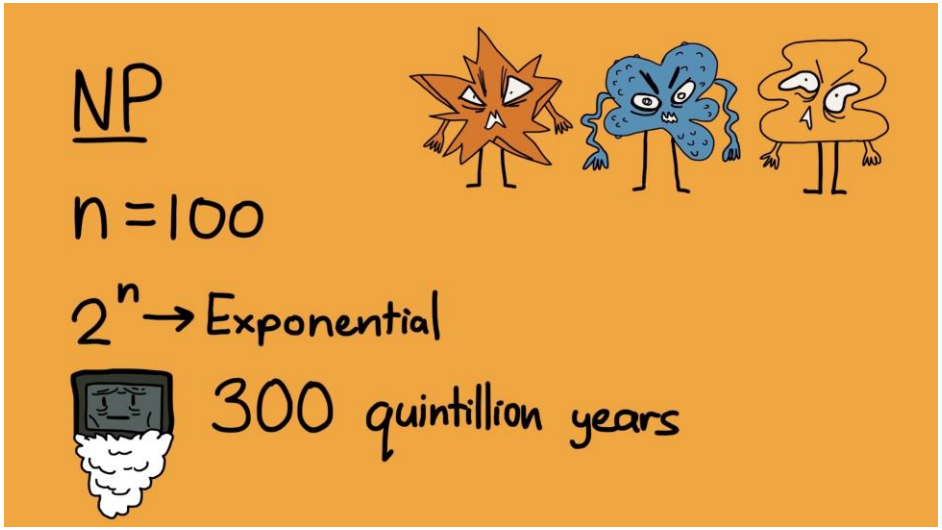
5	3			7				
6			1	9	5			
	9	8					6	
8				6				3
4			8		3			1
7				2				6
	6					2	8	
			4	1	9			5
				8			7	9

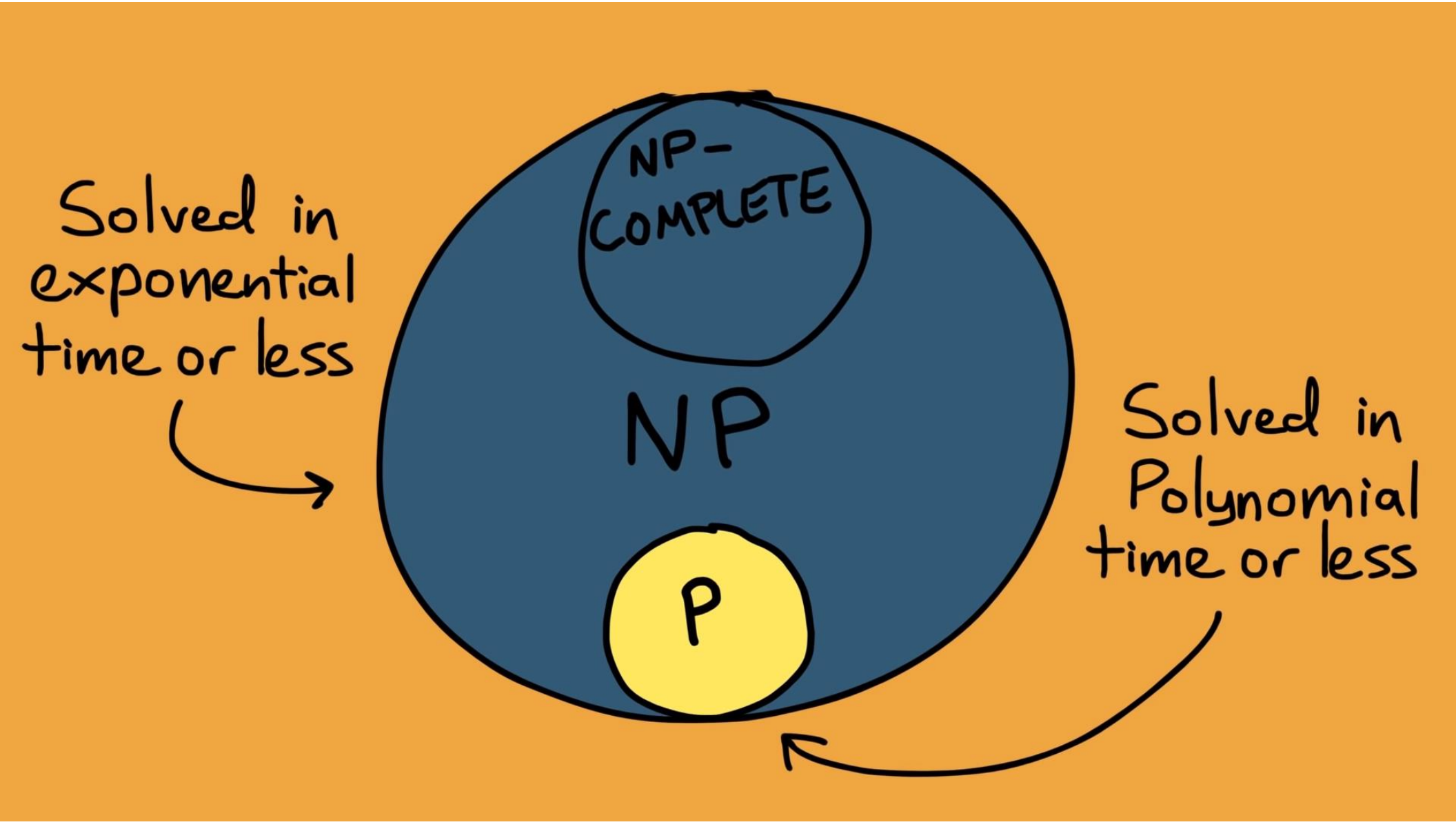
5	3	4	6	7	8	9	1	2
6	7	2	1	9	5	3	4	8
1	9	8	3	4	2	5	6	7
8	5	9	7	6	1	4	2	3
4	2	6	8	5	3	7	9	1
7	1	3	9	2	4	8	5	6
9	6	1	5	3	7	2	8	4
2	8	7	4	1	9	6	3	5
3	4	5	2	8	6	1	7	9

And Much More....

Time Comparison between P and Np






Solved in exponential time or less
NP-COMPLETE
NP
P
Solved in Polynomial time or less

NP-Complete Problems

- Shortest Paths Problem
  - Given a weighted graph G and a source vertex s, find the minimum weight path from s to each vertex v
  - The running time is $O(n \cdot m)$ [Bellman-Ford Algorithm]

- Longest Paths Problem
  - Find a longest path between two vertices
  - The problem is NP-Complete

NP-Complete Problems

- Euler Tour Problem
  - An Euler tour of a connected graph G is a cycle that traverses each edge of G exactly once
  - The Euler Tour Problem is to determine an Euler tour in a connected graph
  - The running time is $O(m)$

- Hamiltonian Cycle Problem
  - A Hamiltonian cycle of a connected graph G is a simple cycle that contains each vertex of G exactly once
  - The Hamiltonian cycle problem: given a graph G, does it have a Hamiltonian cycle?
  - The problem is NP-Complete

NP-Complete Problems

- The well-known *traveling salesman problem*:
  - Optimization variant: a salesman must travel to n cities, visiting each city exactly once and finishing where he begins. How to minimize travel cost?
  - We are given a weighted complete undirected graph G, and we must find a Hamiltonian cycle of G with minimum cost

- *How would we turn this into a decision problem?*
  - A: ask if there exists a TSP with cost $\leq k$

P and NP

- As mentioned, **P** is set of problems that can be solved in polynomial time
- **NP** (*Nondeterministic Polynomial time*) is the set of problems that can be solved in polynomial time by a *nondeterministic* computer
  - *What the hell is that?*

Nondeterminism

- Think of a non-deterministic computer as a computer that magically "guesses" a solution, then has to verify that it is correct
  - If a solution exists, computer always guesses it
  - One way to imagine it: a parallel computer that can freely spawn an infinite number of processes
    - Have one processor work on each possible solution
    - All processors attempt to verify that their solution works
    - If a processor finds it has a working solution
  - So, **NP** = problems *verifiable* in polynomial time

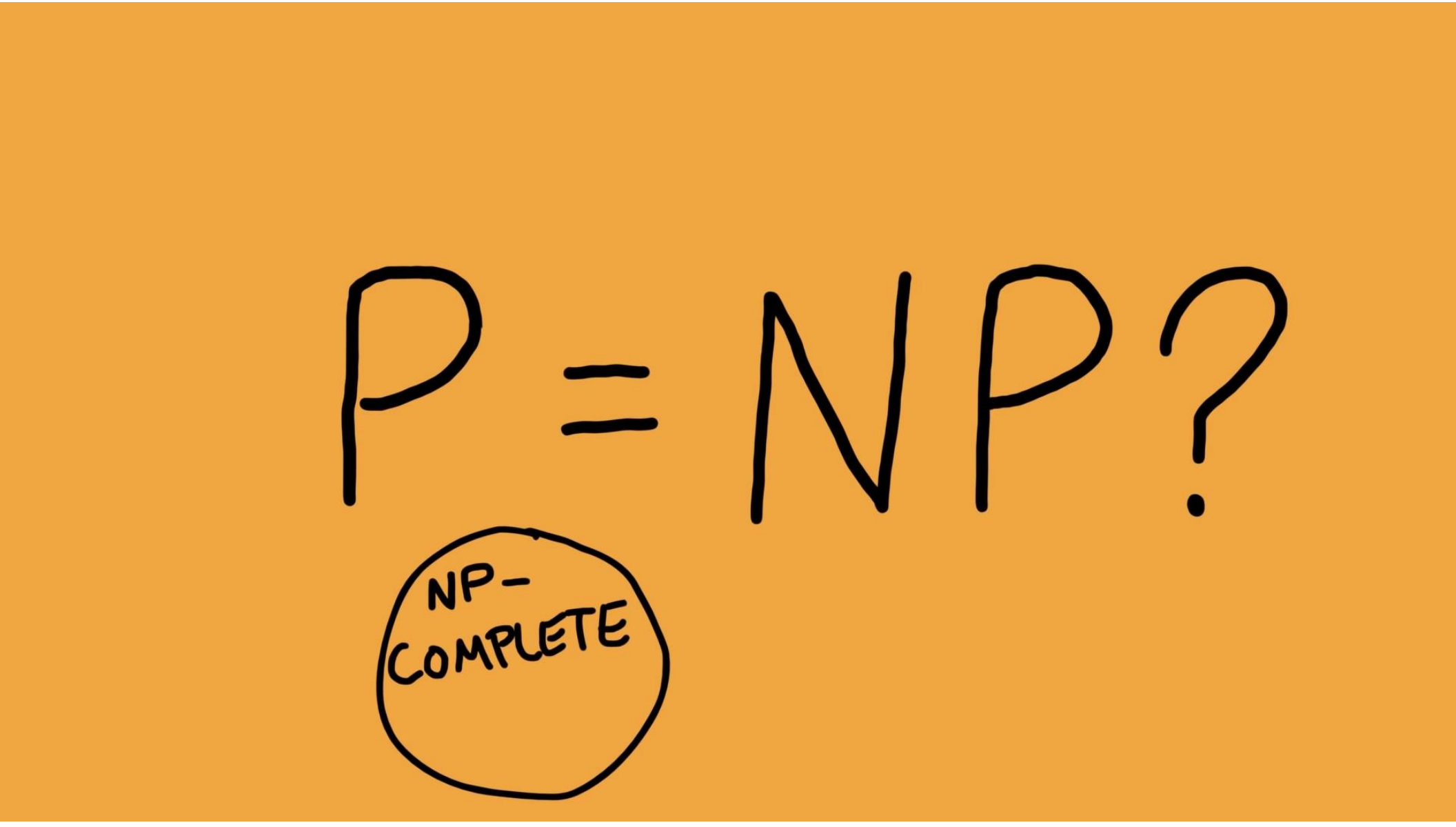
Review: **P** and **NP**

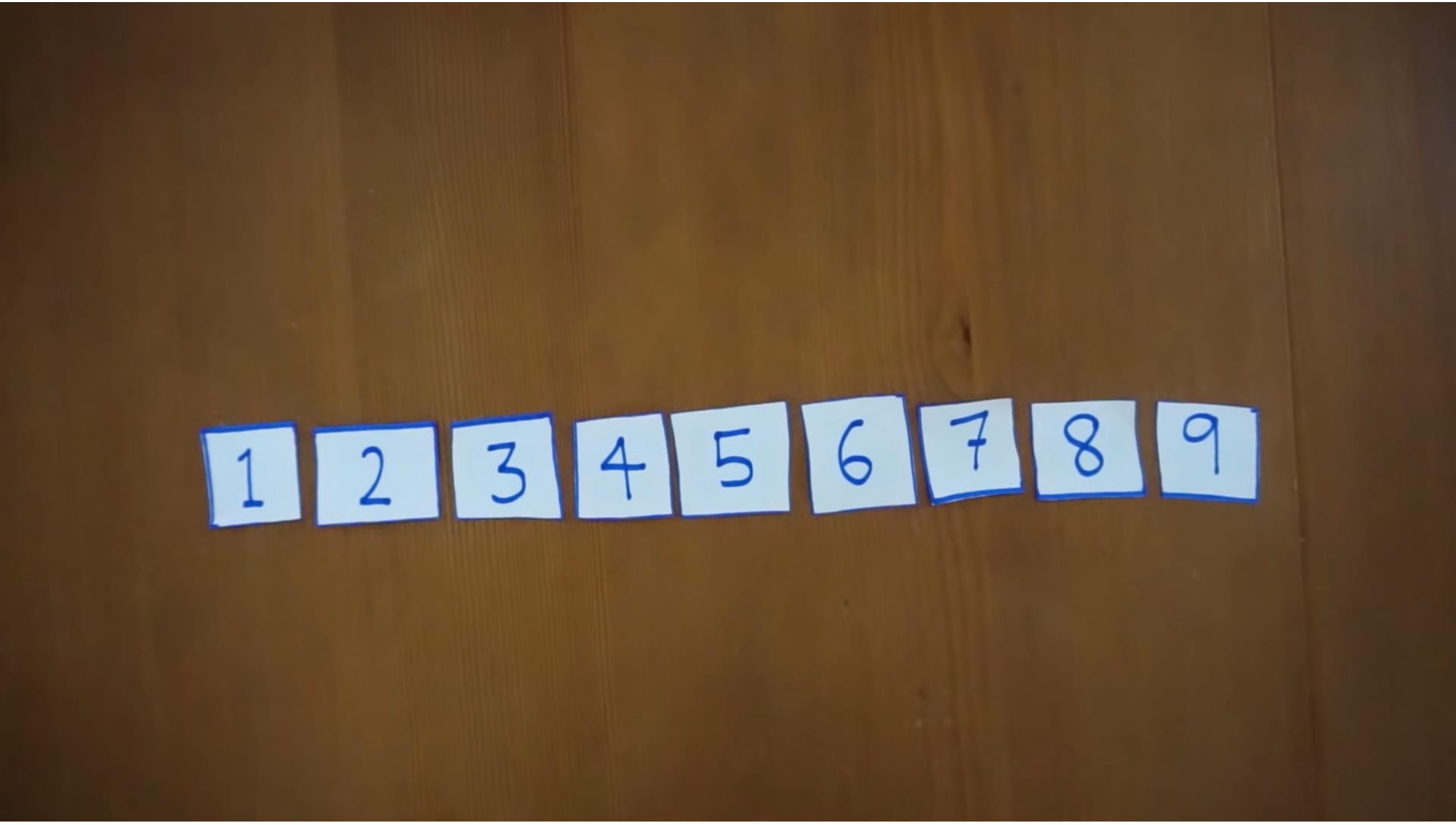
- *What do we mean when we say a problem is in **P*** ?
  - A solution can be found in polynomial time
- *What do we mean when we say a problem is in **NP*** ?
  - A solution can be verified in polynomial time
- *What is the relationship between **P** and **NP*** ?
  - Ans: **P** $\subseteq$ **NP**, but no one knows whether **P = NP**

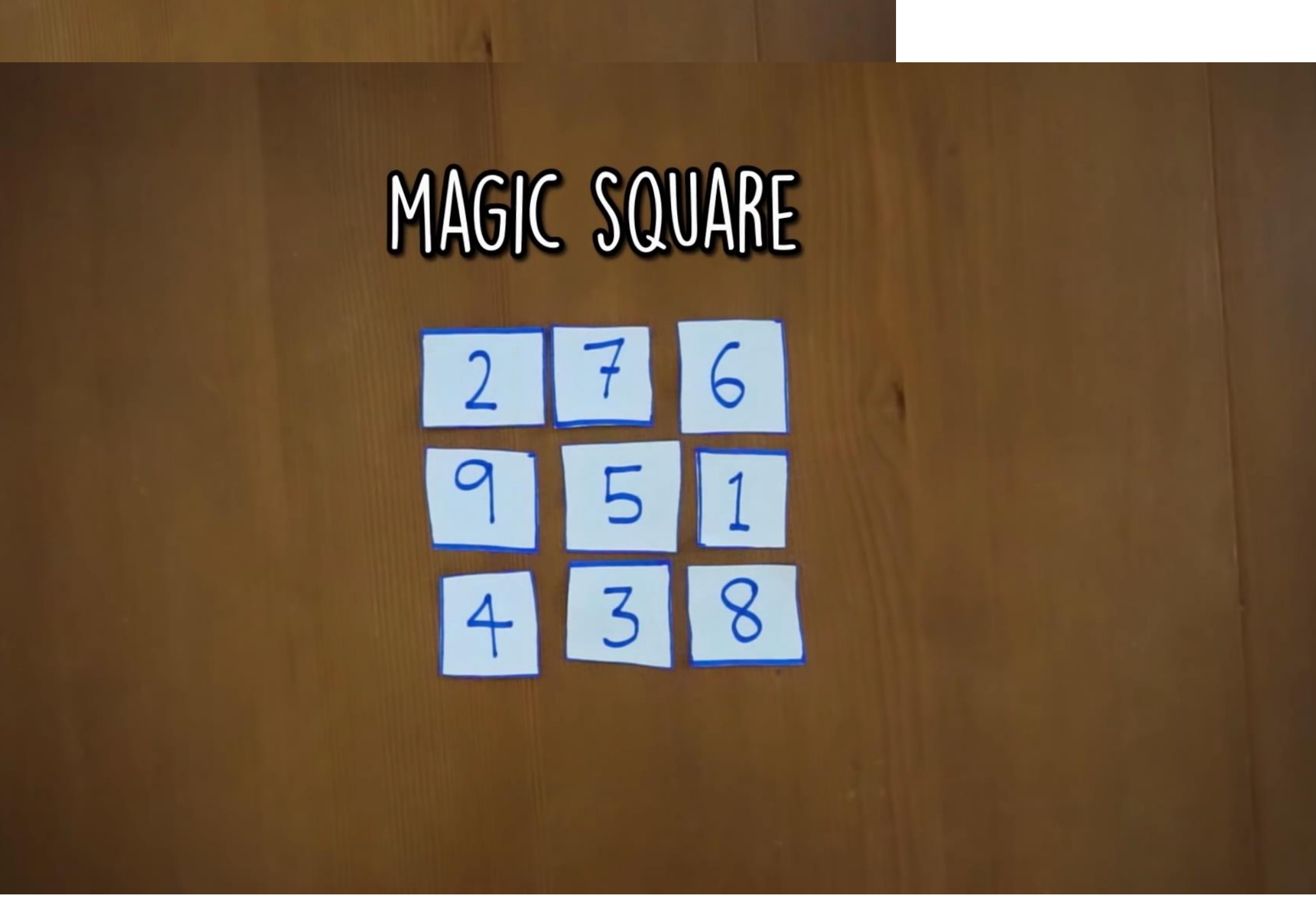
P and NP

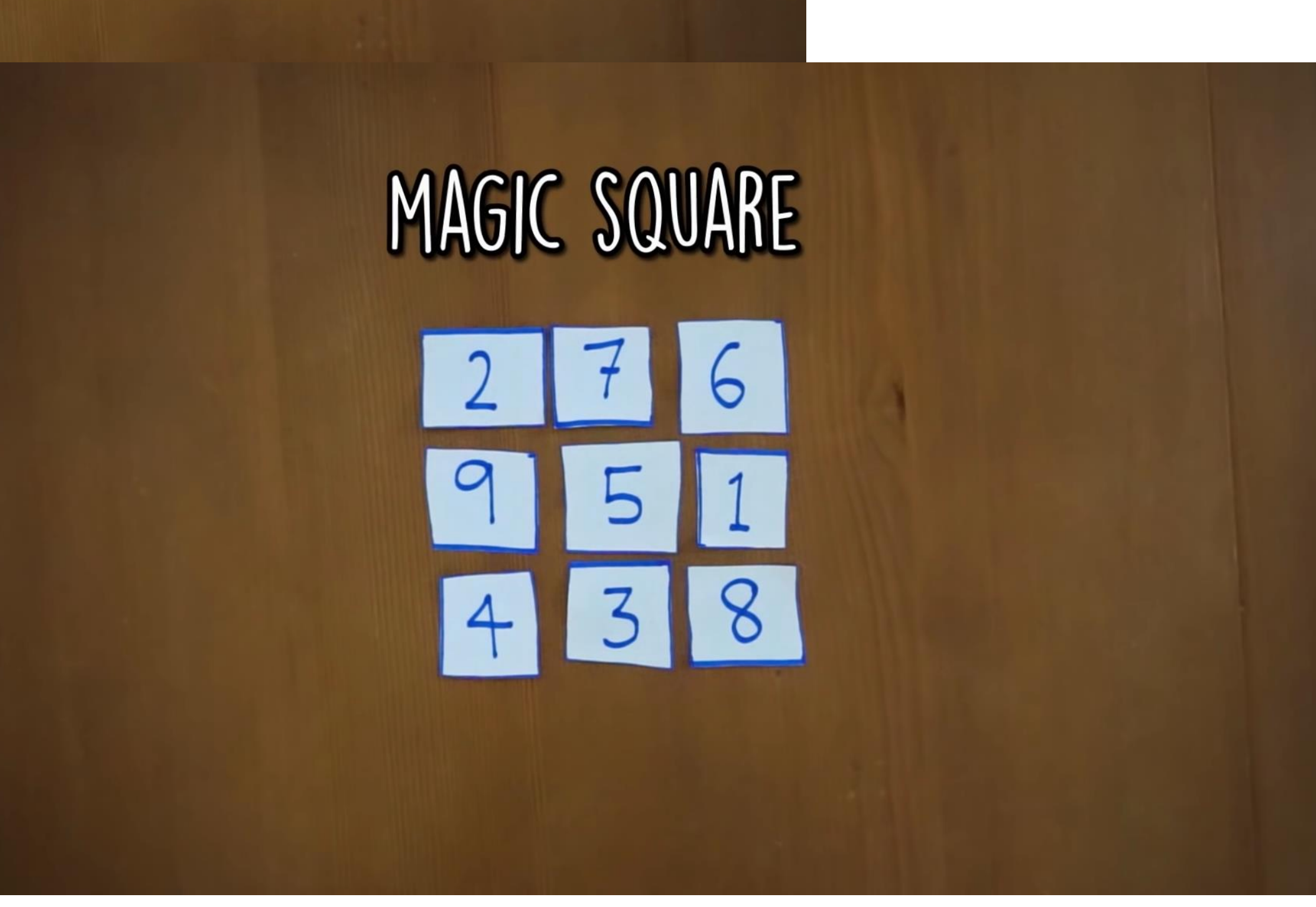
- Summary so far:
  - **P** = problems that can be solved in polynomial time
  - **NP** = problems for which a solution can be verified in polynomial time
  - **P** $\subseteq$ **NP**
  - The big question: Does **P = NP** ? (most suspect not)

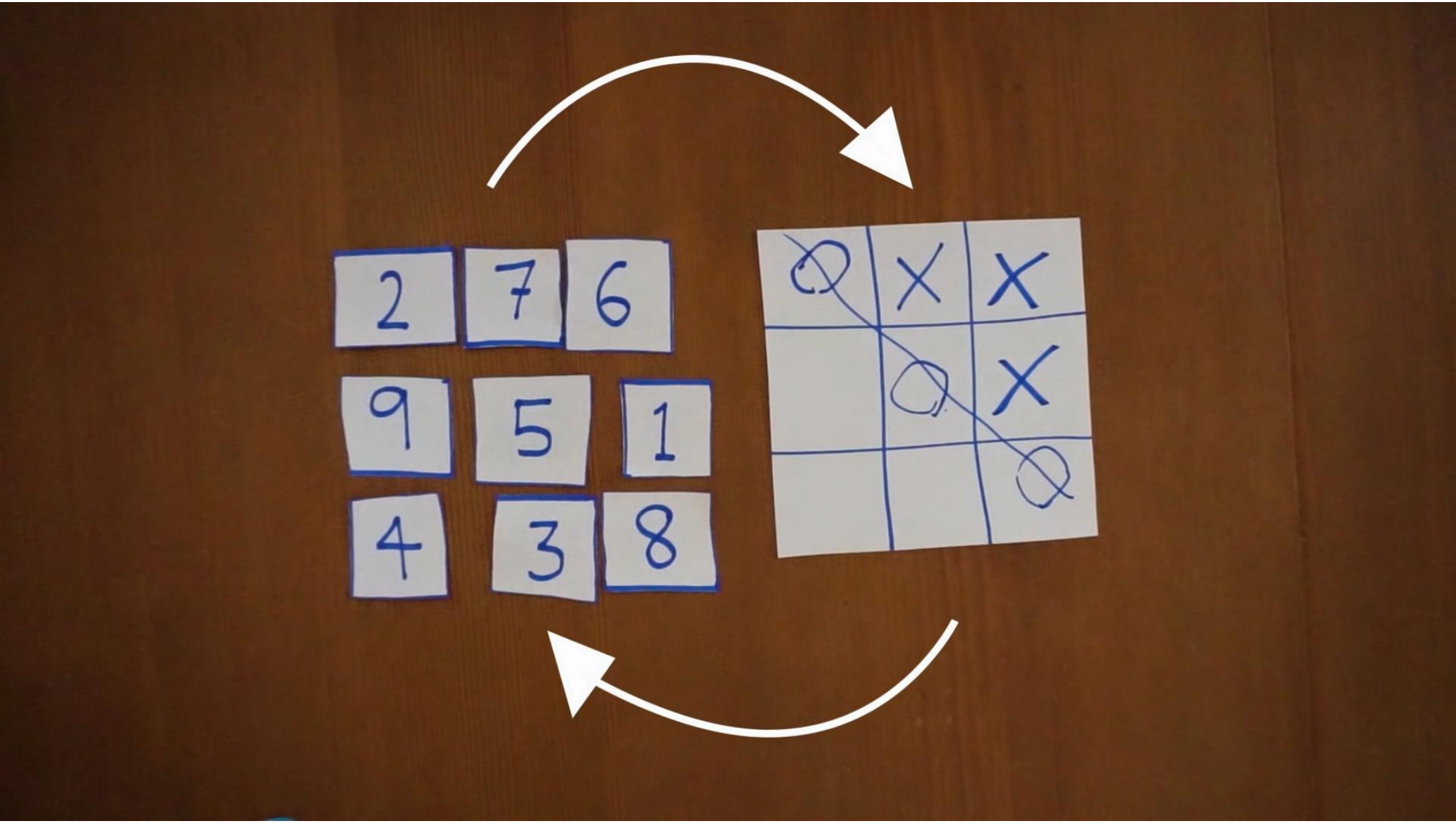
- Hamiltonian-cycle problem is in **NP**:
  - Cannot be solved in polynomial time
  - Easy to verify a solution in polynomial time (*How?*)


P = NP?
NP-
COMPLETE

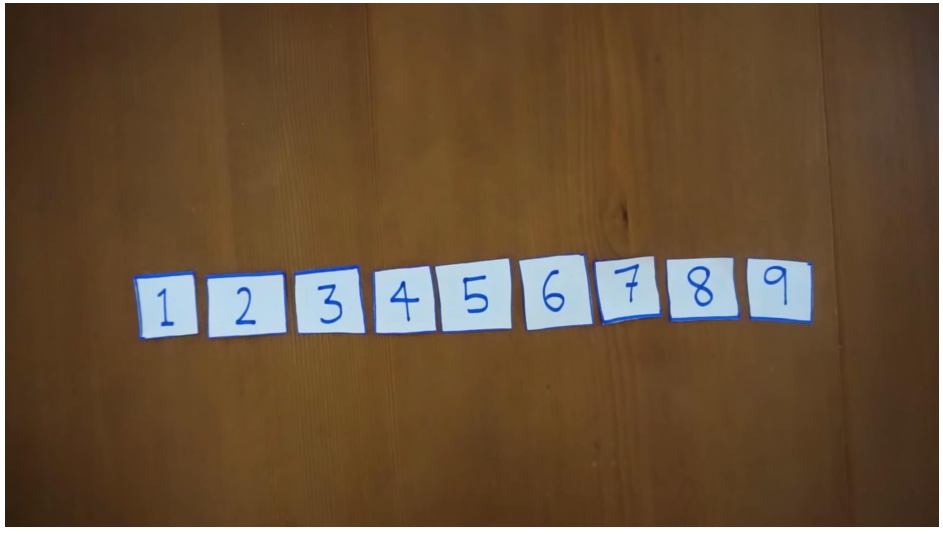

1 2 3 4 5 6 7 8 9

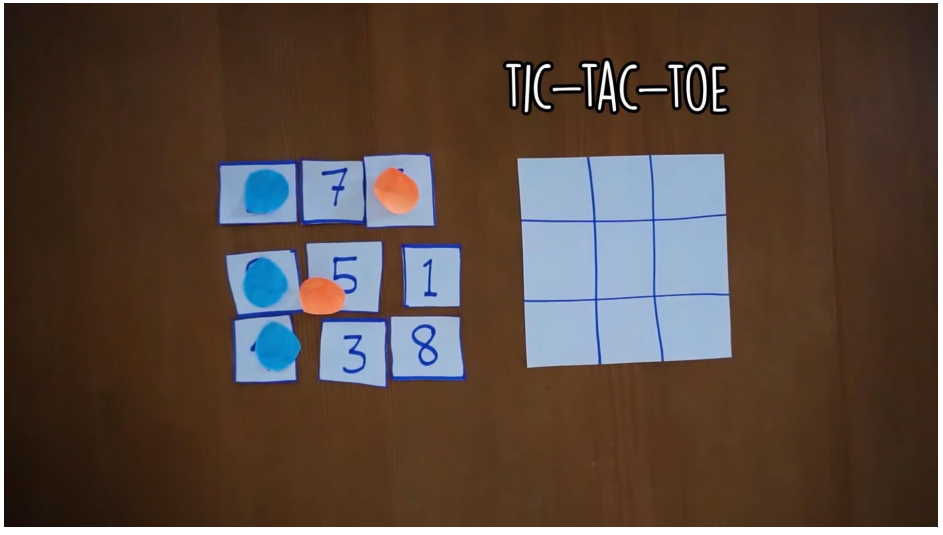

MAGIC SQUARE
2 7 6
9 5 1
4 3 8


MAGIC SQUARE
2 7 6
9 5 1
4 3 8


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

Reduction





Reduction

- A problem R can be *reduced* to another problem Q if any instance of R can be rephrased to an instance of Q, the solution to which provides a solution to the instance of R.
  - This rephrasing is called "*Reducibility*"
- Intuitively: If R reduces in polynomial time to Q,

  R is "no harder to solve" than Q
- Example: *lcm*(*m*, *n*) = *m* * *n* / *gcd*(*m*, *n*),

  lcm(*m*, *n*) problem is reduced to *gcd*(*m*, *n*) problem

NP-Complete and NP-Hard

- *What do we mean when we say a problem is in **NP-Hard*** ?
  - A problem is said to be in *NP-hard* if every problem in NP can be polynomial-time reducible to it.
- *What do we mean when we say a problem is in **NP-Complete*** ?
  - A problem is said to be in *NP-Complete* if
    - it is in NP-Hard, and
    - it is in NP
- *What is the relationship between **NP-Hard** and **NP-Complete*** ?
  - ***NP-Complete*** $\subseteq$ ***NP-Hard***, but
  - No one knows whether ***NP-Complete = NP-Hard***

NP-Complete and NP-Hard

- *What do we mean when we say a problem is in **NP-Hard*** ?
  - A problem is said to be in *NP-hard* if every problem in NP can be polynomial-time reducible to it.
- *What do we mean when we say a problem is in **NP-Complete*** ?
  - A problem is said to be in *NP-Complete* if
    - it is in NP-Hard, and
    - it is in NP
- *What is the relationship between **NP-Hard** and **NP-Complete*** ?
  - ***NP-Complete*** $\subseteq$ ***NP-Hard***, but
  - No one knows whether ***NP-Complete = NP-Hard***

NP-Complete Problems

- We will see that NP-Complete problems are the "hardest" problems in NP:
  - If any *one* NP-Complete problem can be solved in polynomial time…
  - …then *every* NP-Complete problem can be solved in polynomial time…
  - …and in fact *every* problem in **NP** can be solved in polynomial time (which would show **P = NP**)

  - Thus: solve Hamiltonian-cycle problem in $O(n^{100})$ time, you've proved that **P = NP**.

Be rich & famous (Turing Award) !!!

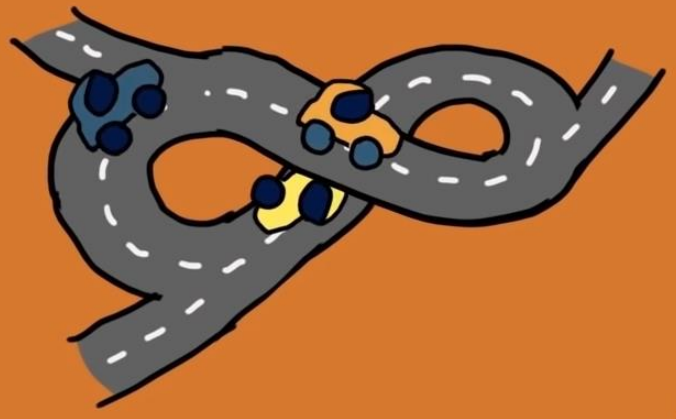
Million Dollars Award !!!

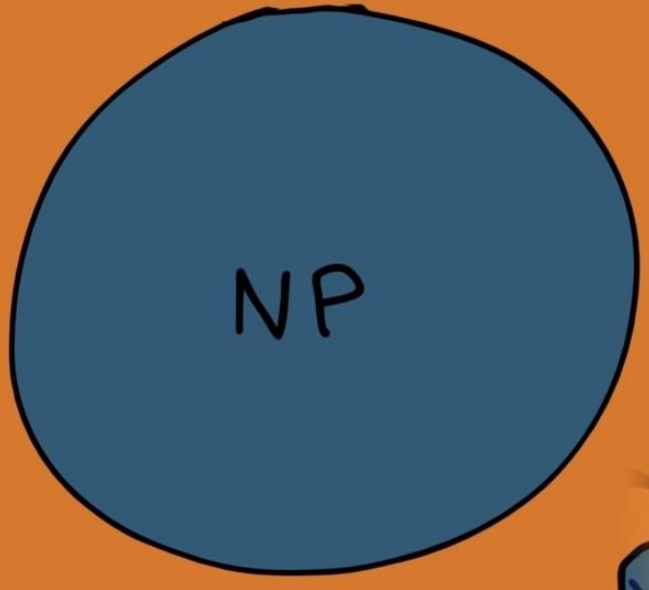

Where is my million dollars?!
P = NP
N = 1
P/P = N
NP = P

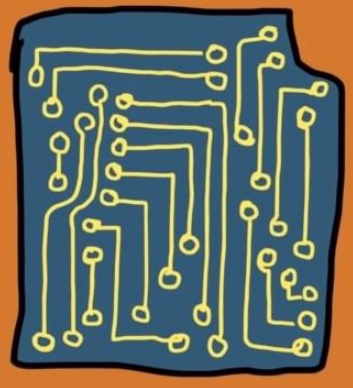
Why Prove NP-Completeness?

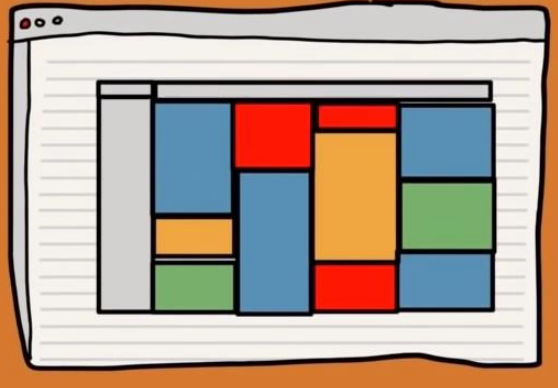
- Though nobody has proven that **P** != **NP**, if you prove a problem NP-Complete, most people accept that it is probably intractable
- Therefore it can be important to prove that a problem is NP-Complete
  - Don't need to come up with an efficient algorithm
  - Can instead work on *approximation algorithms*

IF P=NP...









How P ≠ NP saves us?

How P ≠ NP saves us?

When Everyone's a Super, No One Will Be!

Coping with NP-Hardness

Brute-force algorithms

- Develop clever enumeration strategies
- Guaranteed to find optimal solution
- No guarantees on running time

Heuristics

- Develop intuitive algorithms
- Guaranteed to run in polynomial time
- No guarantees on quality of solution

Approximation algorithms

- Guaranteed to run in polynomial time
- Guaranteed to find "high quality" solution, say within 10% of optimum
- Obstacle: need to prove a solution's value is close to optimum, without even knowing what optimum value is !

Approximation Algorithms

- **Goal:** Find an algorithm which returns near-optimal solution to a hard optimization problem
- What does "near-optimal" mean?
- An approximate algorithm has an approximation ratio of $\rho(n)$ if for any input of size n,

$$\max\{C/C^*, C^*/C\} \leq \rho(n)$$

where C = cost of solution produced by approx. algo.

C^* = cost of optimal solution

- Applies to both minimization and maximization problems
- Also called $\rho(n)$-approximation algorithm

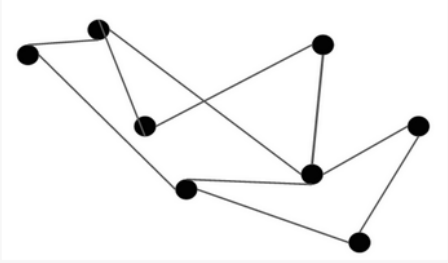
Performance Ratios [cont.]

- Maximization problem
  - $0 < C <= C^*$
  - C^* / C factor by which the cost of the optimal solution is larger than the cost of the approximate solution
- Minimization problem
  - $0 < C^* <= C$
  - C / C^* factor by which the cost of the approximate solution is larger than the cost of the optimal solution

EXAMPLE

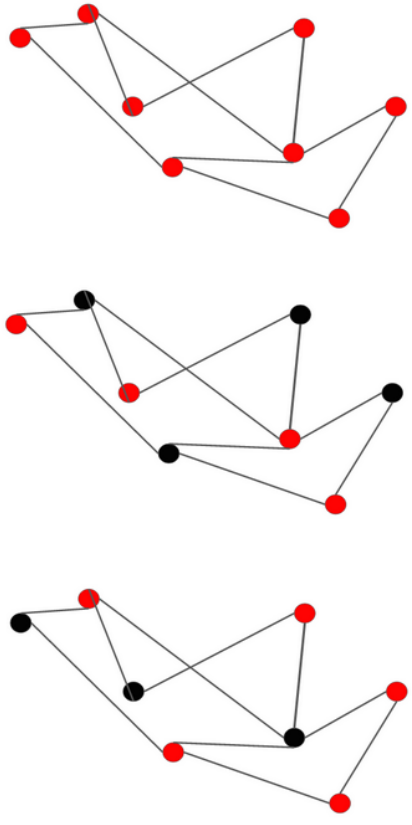
Say you have an art gallery with many hallways and turns. Your gallery is displaying very valuable paintings, and you want to keep them secure. You are planning to install security cameras in each hallway so that the cameras have every painting in view. If there is a security camera in a hallway, it can see every painting in the hallway. If there is a camera in the corner where two hallways meet (the turn), it can view paintings in both hallways. We can model this system as a graph where the nodes represent the places where the hallways meet or when a hallway becomes a dead end, and the edges are the hallways.

In this graph, show where you would place the cameras such that all paintings are covered — this is a vertex cover! (There are many solutions).



Show Answer

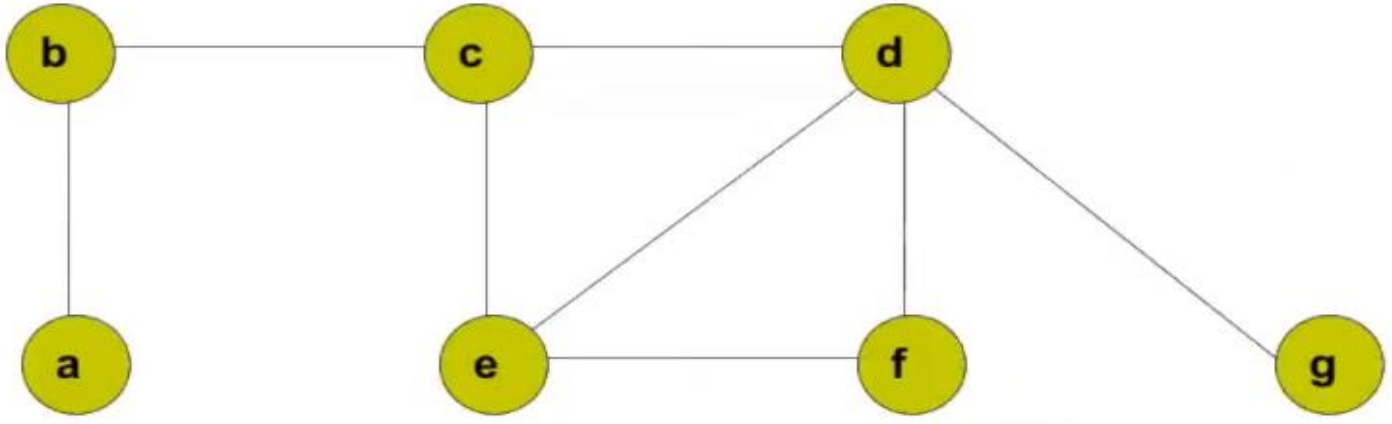
The first is a trivial solution — have cameras at all of the nodes. By definition, this is a vertex cover since all of the edges in graph must be connected to at least one of the vertices in the cover.



The vertex-cover problem

- Let G=(V,E) be an undirected graph
- A vertex cover
  - subset V' of V
  - such that if (u,v) is an edge of G, then either u belongs in V' or v belongs in V' (or both)
- The size of a vertex cover is the number of vertices in it

```
def greedy(E, V)
C = {}
while E not empty:
    select any edge with endpoints (u,v) from E
    add (u,v) to C
    remove all edges incident to u or v from E
return C
```

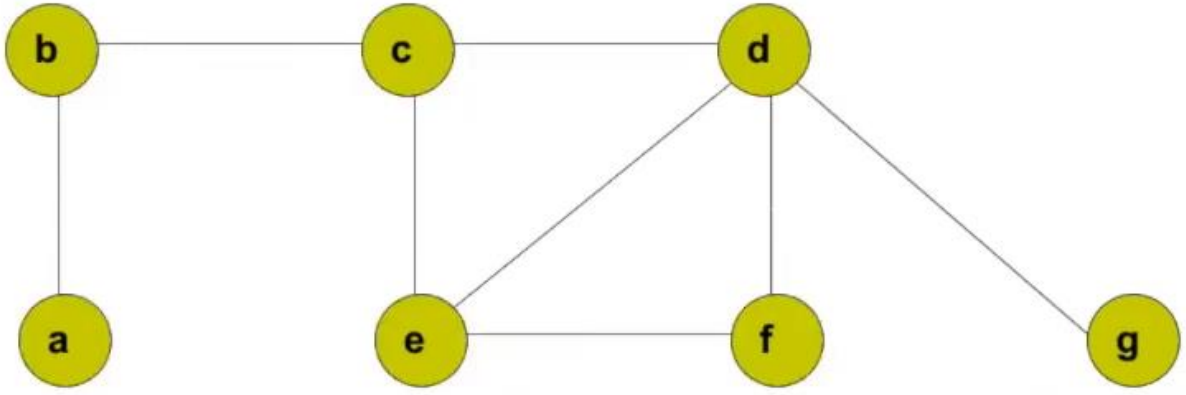


C :

E' : (a,b) (b,c) (c,d) (c,e) (d,e) (d,f) (d,g) (e,f)

By Sorting with Maximum Edge Degree

Improved version:

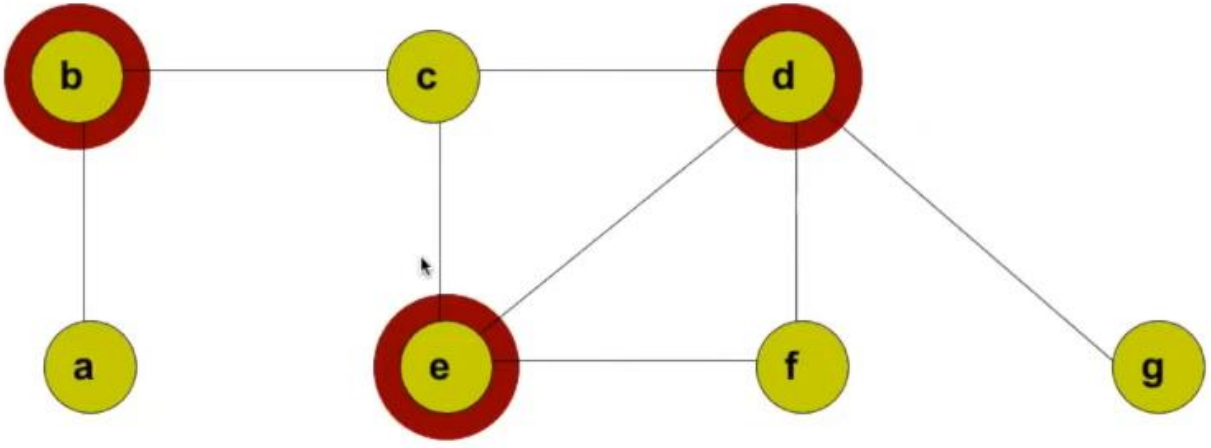


C :

E' : (d,e) (c,d) (c,e) (d,f) (d,g) (e,f) (b,c) (a,b)

Most Optimal Solution

Consider the graph:



By inspection, optimal vertex cover is {b,d,e}

C* = size of optimal solution = 3

Theorem: APPROX-VERTEX-COVER is a polynomial-time 2-approximation algorithm

- Proof:
  - Polynomial time algorithm because : Complexity = O(V+E)
  - Let, A = set of edges picked by approximation algorithm
  - No 2 edges in A share an endpoint.
  - Thus no 2 edges in A are covered by the same vertex in C*
  - **So, Lower bound:** $|C^*| \geq |A|$
  - We pick an edge for which neither of its endpoints are already in C
  - **So, Upper bound:** $|C| = 2|A|$
  - **Therefore:** $|C| = 2|A| \leq 2|C^*|$

Referrence

- CLRS Textbook
- YouTube Channel: Up and Atom
- Brilliant.org