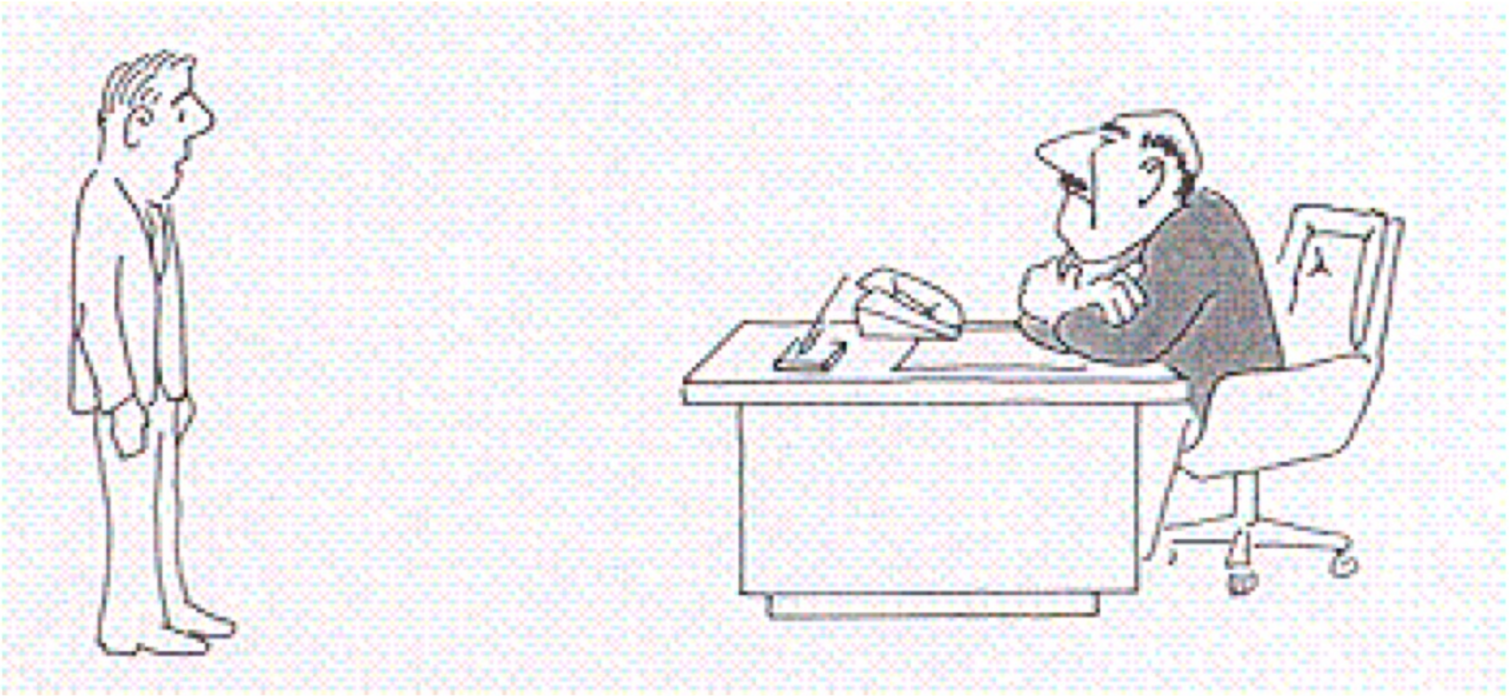
# The Limits of Algorithms

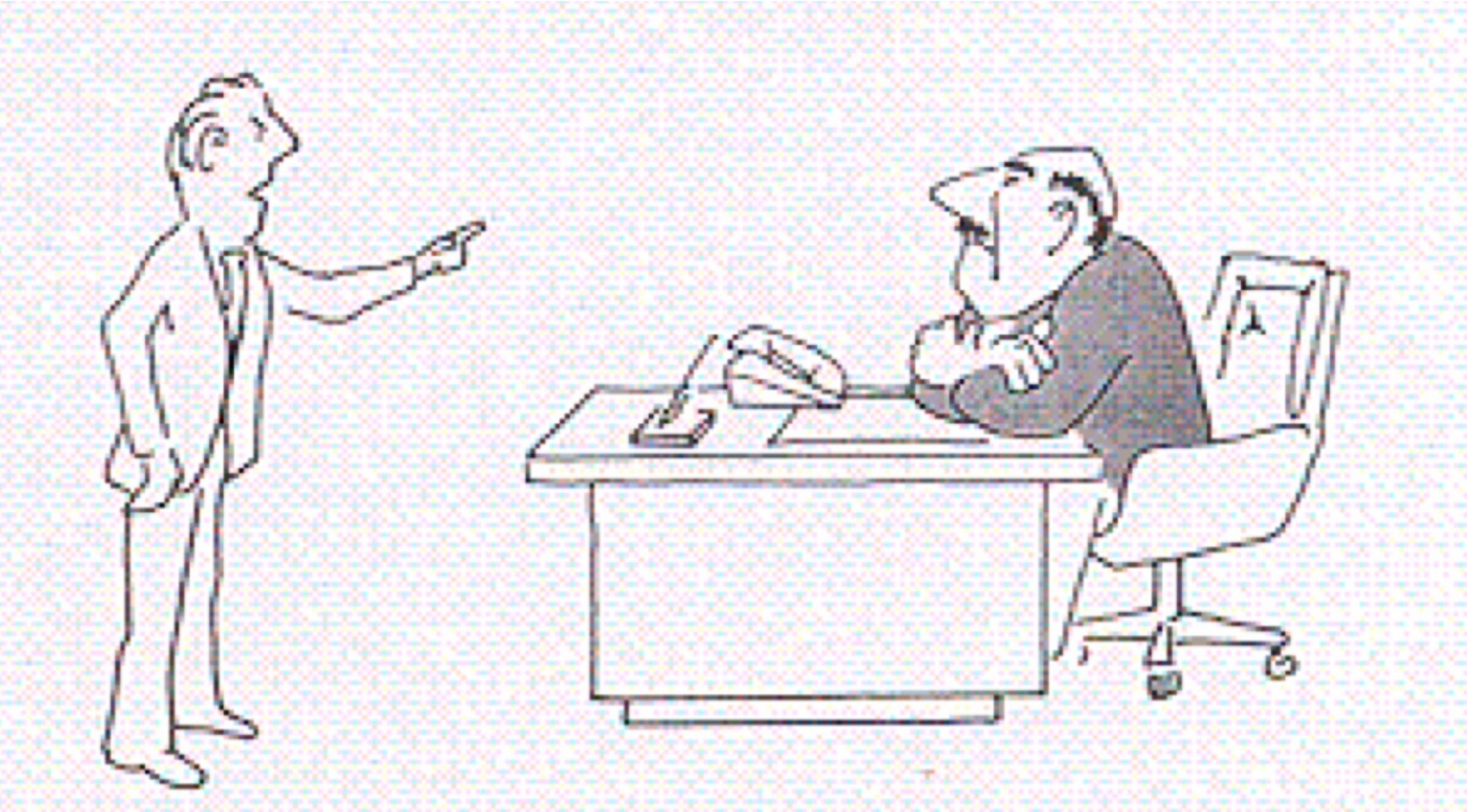
- This class is about problems that can be solved by **polynomial-time algorithms**
  - Given any input of size $n$, the algorithm runs in time $O(n^c)$ for some constant $c \geq 0$
- Can **every** problem be solved in polynomial-time?
  - No!

# The Limits of Algorithms



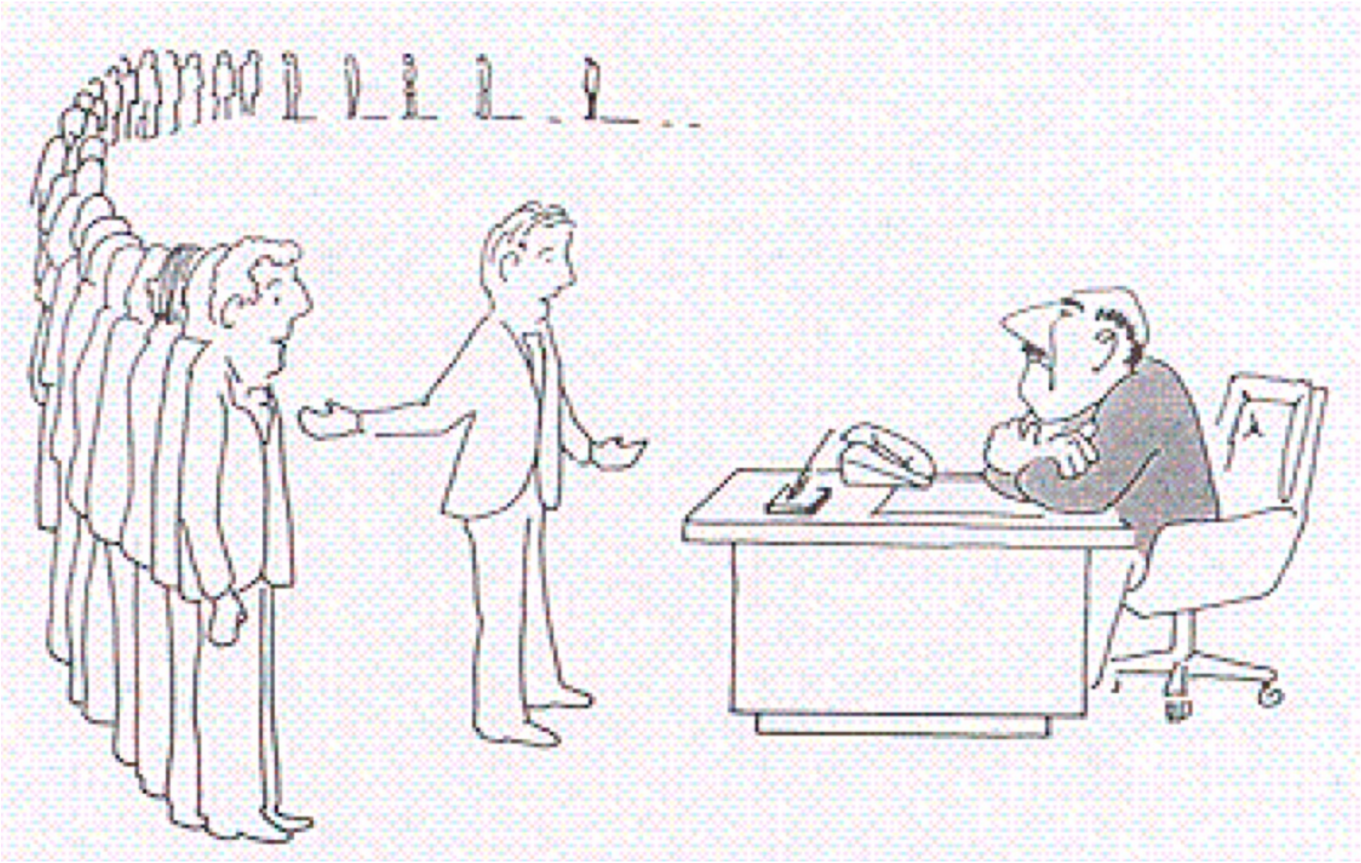
"I can't find an efficient algorithm. I guess I'm just too dumb."

# The Limits of Algorithms



"I can't find an efficient algorithm, because no such algorithm is possible."

# The Limits of Algorithms



“I can’t find an efficient algorithm, but neither can any of these famous people.”

**NP-Hardness:**

Part 1

- Problem in P: 2-SAT

# P, NP, NP-Complete

- Hard Problems
  - Cannot be solved in polynomial time

- Problem Classes
  - P – can be solved in poly. time   ("easy")
  - NP - solutions can be checked in poly. time
  - NP-Complete – "hardest" problems in NP

- Reductions
  - Used to show "hardness"

- For more: see Sipser's "Intro. To Theory of Computation"

# An "Easy" Problem: 2-SAT

- Variables (e.g. X, Y and Z) & literals (e.g. X, ~X; X is true if X is true, ~X ("not X") is true if X is false
- 2-Clause: 2 literals ORed together (e.g. $X \vee {\sim}Z$)
  - True if either literal is true, False if both are false
- OR truth table:
  - $T \vee T =$
  - $T \vee F =$
  - $F \vee T =$
  - $F \vee F =$
- 2-SAT:
  - **Input:** A formula of 2-clauses ANDed together
  - **Question:** Is there an assignment of variables that makes the formula true (e.g. that satisfies every clause)
  - E.g. $(a \vee {\sim}b) \wedge ({\sim}b \vee c) \wedge ({\sim}a \vee d) \wedge ({\sim}d \vee {\sim}c)$

# An “Easy” Problem: 2-SAT

- $(a \vee \sim b) \wedge (\sim b \vee c) \wedge (\sim a \vee d) \wedge (\sim d \vee \sim c)$

- **Find an assignment that satisfies the above formula, or argue why there isn’t one**

# An “Easy” Problem: 2-SAT

- 2-SAT is “easy” because it can be solved in polynomial time

- Consider a 2-clause $(x \vee y)$
  - If x is false, then y must be true
  - If y is false, then x must be true
  - $(x \vee y)$ is equivalent to:

# An “Easy” Problem: 2-SAT

- 2-SAT is “easy” because it can be solved in polynomial time

- Consider a 2-clause (x V y)
  - If x is false, then y must be true (~x -> y)
  - If y is false, then x must be true (~y -> x)
  - ***Equivalent to: (~x -> y)  AND (~y -> x)

- Given an instance of 2-SAT, we can create an implication graph *G*
  - Each literal is a **node**
  - Each clause is 2 **edges** (e.g (x V y) -> (~x, y), (~y, x)   *note: directed graph*)
  - A **path** in *G* from x to y means that if x is true, then y must be true

# An "Easy" Problem: 2-SAT

- Each literal is a **node,** each clause is 2 **edges** (e.g $(x \vee y)$ -> $(\sim x, y)$, $(\sim y, x)$), a **path** in *G* from x to y means that if x is true, then y must be true

- Ex. $(a \vee \sim b) \wedge (b \vee c) \wedge (\sim a \vee d) \wedge (\sim d \vee \sim c)$

- How to check for existence of valid assignment?

# An “Easy” Problem: 2-SAT

- If there is a variable x such that there is a path from x to ~x and from ~x to x, then the formula is not satisfiable
- **Otherwise,** it is satisfiable

- **Runtime:**
- 1 node/variable ->
- 2 edges/clause ->
- For each variable x, check if PATH(x, ~x) AND PATH(x,~x)

# 3-SAT

- Same as 2-SAT, except each clause has 3 literals

- Ex. $(a \vee \sim b \vee d) \wedge (b \vee c \vee e) \wedge (\sim a \vee b \vee d) \wedge (\sim d \vee \sim e \vee \sim c)$

- Can you solve this in polynomial time?

**NP-Hardness**

Part 2

- Problems in NP

# A Not-So Easy Problem: 3-SAT

- Same as 2-SAT, except each clause has 3 literals
- Ex. $(a \vee \sim b \vee d) \wedge (b \vee c \vee e) \wedge (\sim a \vee b \vee d) \wedge (\sim d \vee \sim e \vee \sim c)$
- Can you solve this in polynomial time?

# A Not-So Easy Problem: 3-SAT

- Same as 2-SAT, except each clause has 3 literals
- Ex. $(a \vee \sim b \vee d) \wedge (b \vee c \vee e) \wedge (\sim a \vee b \vee d) \wedge (\sim d \vee \sim e \vee \sim c)$
- Can you solve this in polynomial time? *TBD, but probably not*
- Solving seems to require trying almost all possibilities

# A Not-So Easy Problem: 3-SAT

- Why has no poly. time algorithm been found for 3-SAT?
  - Lack of structure in solution space -> no efficient search
- Ways to get rich:
  - Prove no poly. time solution exists for 3-SAT
  - Find a poly. time solution for 3-SAT

# 2-SAT & 3-SAT and P & NP

- P – can be solved in poly. time   (“easy”)
- NP - solutions can be *checked* in poly. time (given a solution, you can verify its correctness in poly. time)

- Which of these problems are in P?
  - 2-SAT
  - 3-SAT

- Which of these problems are in NP?
  - 2-SAT
  - 3-SAT

# 2-SAT & 3-SAT and P & NP

- P – can be solved in poly. time   ("easy")
- NP - solutions can be *checked* in poly. time (given a solution, you can verify its correctness in poly. time)

- Which of these problems are in P?
    - **2-SAT**
    - 3-SAT – TBD but unlikely!

- Which of these problems are in NP?
    - **2-SAT**
    - **3-SAT**

- Finding a solution is harder than checking one, so P $\subseteq$ NP!

# Other Problems in NP but not known to be in P

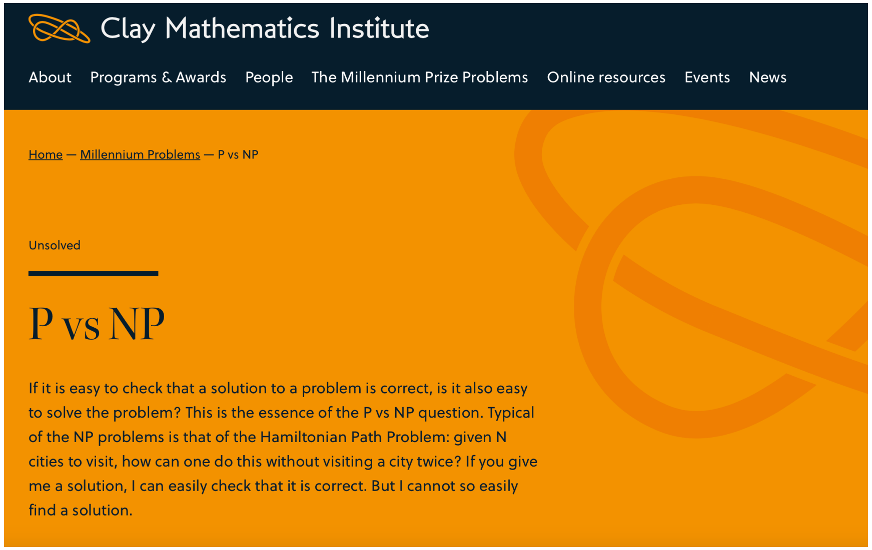
- P – can be solved in poly. time (“easy”)
- NP - solutions can be *checked* in poly. time (given a solution, you can verify its correctness in poly. time)

- **3-SAT**
- **CLIQUE**
- **COLORING**
- **VERTEX COVER**
- **HAMILTONIAN CYCLE**
- **TRAVELING SALESMAN**
- **SUBSET-SUM**
- **…**

# Other Problems in NP but not P

- P – can be solved in poly. time ("easy")
- NP - solutions can be *checked* in poly. time (given a solution, you can verify its correctness in poly. time)

- **$1M Question: Does P = NP?**
  - Assuming not, problem X being in NP and not P → X is hard



**NP-Hardness**

Part 3

- NP-Completeness

# NP-Complete

- P – can be solved in poly. time (“easy”)
- NP - solutions can be *checked* in poly. time (given a solution, you can verify its correctness in poly. time)

- **NP-Complete:** problems in NP that are “as hard” as any other problem in NP

- **Importance of NP-C:**

*1. If any NP-complete problem can be solved in poly-time => P=NP*

*2. If you want to show P ≠ NP, then focus on NP-complete problems*

- How do we show one problem is “as hard” as another?

# As Hard?

- If a problem A is *poly-time reducible* to problem B, then B is at least *as hard* as A
- **Ex.** If a problem A cannot be solved in poly-time and is *poly-time reducible* to problem B, B also cannot be solved in poly-time)

- **NP-Complete:** problems in NP that are at least "as hard" as any other problem in NP

- If A is **NP-Complete**, then any problem in NP is poly-time reducible to A.

**NP-Hardness**

Part 4

- Reductions

# Poly-Time Reductions

- **Def.** Problem *A* is poly-time reducible to problem *B* if given an instance *X* of *A*, we can create an instance *f* (*X*) of *B* in poly-time such that a solution to *f* (*X*) can be used to solve *X* in poly-time

| | | | |
|---|---|---|---|
| • Problems | A | | B |
| • Instances | X | $\xrightarrow{\text{Poly-time}}$ | f(X) |
| • Solutions | S(X) | $\xleftarrow{\text{Poly-time}}$ | S(f(X)) |

- We work with *decision problems: YES/NO answers*
- Reductions allow us to use complexity class of 1 problem to draw conclusions about more problems

# Proof Sketch

- **Claim:** If a problem A cannot be solved in poly-time and is *poly-time reducible* to problem B, then B cannot be solved in poly-time
- **Def.** Problem $A$ is poly-time reducible to problem $B$ if given an instance $X$ of $A$, we can create an instance $f(X)$ of $B$ in poly-time such that a solution to $f(X)$ can be used to solve $X$ in poly-time
- **Pf.**
- **Cor.** If A is NP-Complete, and B is in NP, and A is poly-time reducible to B, then B is NP-Complete.

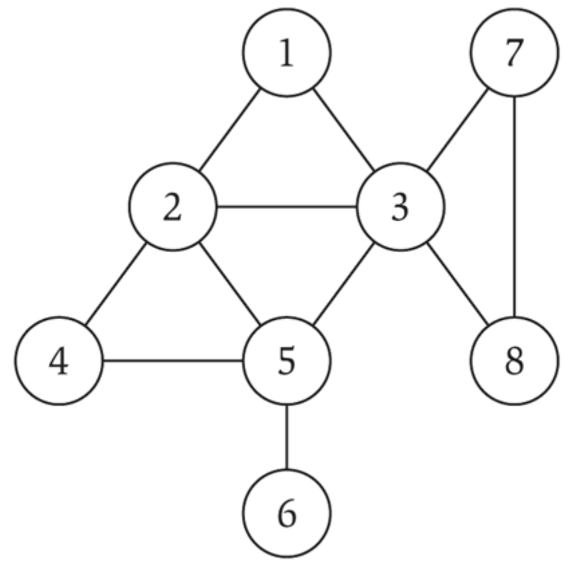
# Reduction: 3-SAT -> IndSet

- **We saw 3-SAT earlier**
  - **NP-Complete**

- Now: IndSet is NP-Complete

- An *independent set* is a set of nodes such that no two of them are adjacent.
- **IndSet:** does an independent set of size *k* exist?

- **Claim:** IndSet is NP-Complete.



# Reduction: 3-SAT -> IndSet

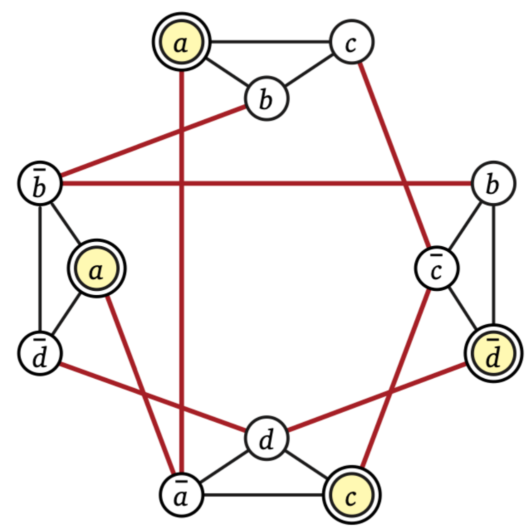
- **Def.** Problem $A$ is poly-time reducible to problem $B$ if given an instance $X$ of $A$, we can create an instance $f(X)$ of $B$ in poly-time such that a solution to $f(X)$ can be used to solve $X$ in poly-time.
- **Cor:** *If A is poly-time reducible to B*, A is NP-Complete, and B is in NP, then B is NP-Complete.

- **Claim:** 3SAT is poly-time reducible to IndSet.

- By the Cor above, this implies IndSet is also NP-complete.

# Reduction: 3-SAT -> IndSet

- **Def.** Problem $A$ is poly-time reducible to problem $B$ if given an instance $X$ of $A$, we can create an instance $f(X)$ of $B$ in poly-time such that a solution to $f(X)$ can be used to solve $X$ in poly-time.
- **Claim:** 3SAT is poly-time reducible to IndSet.
- Construction of f(X):
  - K = number of clauses
  - 3 Vertices for each clause, each for a literal
  - Add an edge between each literal and its negations
  - Add a triangle between literals of each clause




$(a \vee b \vee c) \wedge (b \vee \bar{c} \vee \bar{d}) \wedge (\bar{a} \vee c \vee d) \wedge (a \vee \bar{b} \vee \bar{d})$

# Reduction: 3-SAT -> IndSet

- **Def.** Problem $A$ is poly-time reducible to problem $B$ if given an instance $X$ of $A$, we can create an instance $f(X)$ of $B$ in poly-time such that a solution to $f(X)$ can be used to solve $X$ in poly-time.
- **Claim:** 3SAT is poly-time reducible to IndSet.

Pf (->) If IndSet is YES => 3SAT is YES

# Reduction: 3-SAT -> IndSet

- **Def.** Problem $A$ is poly-time reducible to problem $B$ if given an instance $X$ of $A$, we can create an instance $f(X)$ of $B$ in poly-time such that a solution to $f(X)$ can be used to solve $X$ in poly-time.
- **Claim:** 3SAT is poly-time reducible to IndSet.

Pf (<-) If 3SAT is YES => IndSet is YES