Theory of Computation
Poly-time reductions, NP-completeness

The million dollar question

"What is the largest group of Facebook users that are all connected to each other"

The million dollar question

"What is the largest group of Facebook users that are all connected to each other"

- Can you write an *efficient* algorithm to answer this question?

The million dollar question

"What is the largest group of Facebook users that are all connected to each other"

- Can you write an *efficient* algorithm to answer this question?
- **Can you prove that no efficient algorithm exists for this problem?**

Poly-time computable functions

Poly-time computable functions

- **Recall:** A function $f : \Sigma^* \to \Sigma^*$ is **computable** if there is a Turing machine M that *computes* it

Poly-time computable functions

- **Recall:** A function $f : \Sigma^* \to \Sigma^*$ is **computable** if there is a Turing machine M that *computes* it
  - If we start with w on the tape, M will halt leave $f(w)$ on the tape

Poly-time computable functions

- **Recall:** A function $f : \Sigma^* \to \Sigma^*$ is **computable** if there is a Turing machine M that *computes* it
  - If we start with w on the tape, M will halt leave $f(w)$ on the tape
- **Def:** a computable function f is **poly-time computable** if M runs in polynomial time

Poly-time reductions

Poly-time reductions

- **Recall:** We say $A \leq_M B$ if there is a computable function $f : \Sigma^* \to \Sigma^*$ such that $w \in A \Leftrightarrow f(w) \in B$

Poly-time reductions

- **Recall:** We say $A \leq_M B$ if there is a computable function $f : \Sigma^* \to \Sigma^*$ such that $w \in A \Leftrightarrow f(w) \in B$
  - “YES maps to YES”

Poly-time reductions

- **Recall:** We say $A \leq_M B$ if there is a computable function $f : \Sigma^* \to \Sigma^*$ such that $w \in A \Leftrightarrow f(w) \in B$
  - "YES maps to YES"
  - "NO maps to NO"

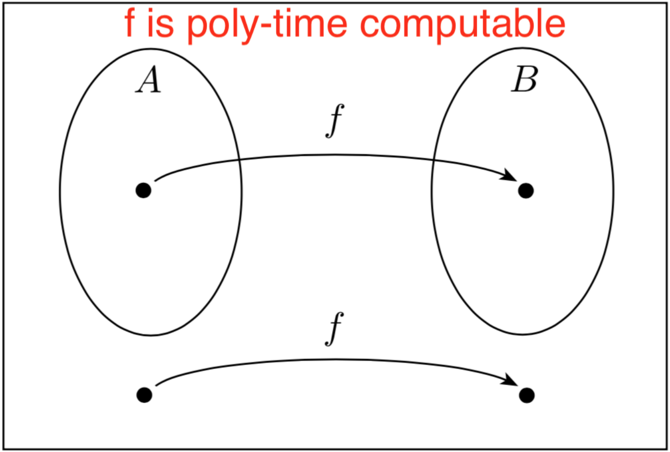
Poly-time reductions

- **Recall:** We say $A \leq_M B$ if there is a computable function $f : \Sigma^* \to \Sigma^*$ such that $w \in A \Leftrightarrow f(w) \in B$
  - "YES maps to YES"
  - "NO maps to NO"
- **Def:** We say A is **poly-time reducible** to B (denoted $A \leq_{\text{poly}} B$) if the reduction f is poly-time computable

Poly-time reductions

- **Recall:** We say $A \leq_M B$ if there is a computable function $f : \Sigma^* \to \Sigma^*$ such that $w \in A \Leftrightarrow f(w) \in B$
  - "YES maps to YES"
  - "NO maps to NO"
- **Def:** We say A is **poly-time reducible** to B (denoted $A \leq_{\text{poly}} B$) if the reduction f is poly-time computable
- Informally, it means that we can "convert" an instance of A to an instance of B in polynomial time

Poly-time reductions



Implications of poly-time reducibility

Theorem: If $B \in \mathrm{P}$ and $A \leq_{\mathrm{poly}} B$, then $A \in \mathrm{P}$

Implications of poly-time reducibility

Theorem: If $B \in \mathrm{P}$ and $A \leq_{\mathrm{poly}} B$, then $A \in \mathrm{P}$

- Since $B \in \mathrm{P}$, there is a machine M_B that decides B in poly-time

Implications of poly-time reducibility

Theorem: If $B \in \mathrm{P}$ and $A \leq_{\mathrm{poly}} B$, then $A \in \mathrm{P}$

- Since $B \in \mathrm{P}$, there is a machine M_B that decides B in poly-time
- Since $A \leq_{\mathrm{poly}} B$ there is a poly-time computable function f such that $w \in A \Leftrightarrow f(w) \in B$

Implications of poly-time reducibility

Theorem: If $B \in \mathrm{P}$ and $A \leq_{\mathrm{poly}} B$, then $A \in \mathrm{P}$

- Since $B \in \mathrm{P}$, there is a machine M_B that decides B in poly-time
- Since $A \leq_{\mathrm{poly}} B$ there is a poly-time computable function f such that $w \in A \Leftrightarrow f(w) \in B$
- Create the following machine poly-time M_A to decide A

Implications of poly-time reducibility

Theorem: If $B \in \mathrm{P}$ and $A \leq_{\mathrm{poly}} B$, then $A \in \mathrm{P}$

- Since $B \in \mathrm{P}$, there is a machine M_B that decides B in poly-time
- Since $A \leq_{\mathrm{poly}} B$ there is a poly-time computable function f such that $w \in A \Leftrightarrow f(w) \in B$
- Create the following machine poly-time M_A to decide A
  1. Compute $f(w)$ (poly-time)

Implications of poly-time reducibility

Theorem: If $B \in \mathrm{P}$ and $A \leq_{\mathrm{poly}} B$, then $A \in \mathrm{P}$

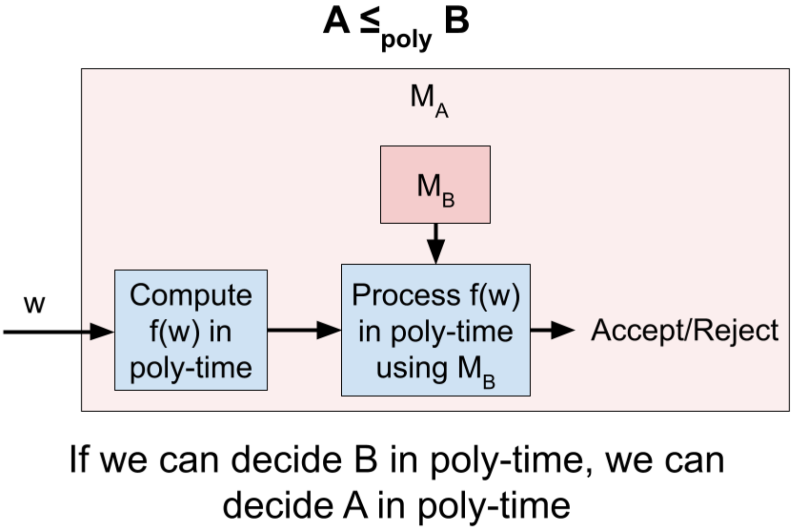
- Since $B \in \mathrm{P}$, there is a machine M_B that decides B in poly-time
- Since $A \leq_{\mathrm{poly}} B$ there is a poly-time computable function f such that $w \in A \Leftrightarrow f(w) \in B$
- Create the following machine poly-time M_A to decide A
  1. Compute $f(w)$ (poly-time)
  2. Run M_B on $f(w)$ (poly-time)

Implications of poly-time reducibility

Theorem: If $B \in \mathrm{P}$ and $A \leq_{\mathrm{poly}} B$, then $A \in \mathrm{P}$

- Since $B \in \mathrm{P}$, there is a machine M_B that decides B in poly-time
- Since $A \leq_{\mathrm{poly}} B$ there is a poly-time computable function f such that $w \in A \Leftrightarrow f(w) \in B$
- Create the following machine poly-time M_A to decide A
    1. Compute $f(w)$ (poly-time)
    2. Run M_B on $f(w)$ (poly-time)
    3. If M_B accepts $f(w)$ then M_A accepts w. Otherwise, M_A rejects w.

Implications of polytime-reducibility



IND-SET $\leq_{\text{poly}}$ CLIQUE

IND-SET $\leq_{\text{poly}}$ CLIQUE

We reduce from IND-SET to CLIQUE as follows:

IND-SET $\leq_{\text{poly}}$ CLIQUE

We reduce from IND-SET to CLIQUE as follows:

1. **Input:** A graph G with V vertices and E edges, and an integer k

IND-SET $\leq_{\text{poly}}$ CLIQUE

We reduce from IND-SET to CLIQUE as follows:

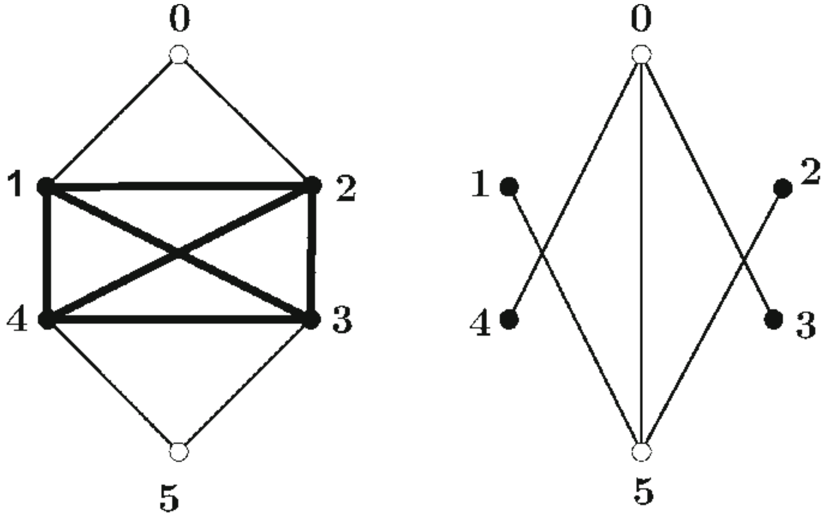
1. **Input:** A graph G with V vertices and E edges, and an integer k
2. Create the **complement graph** $\overline{G}$ by reversing all of the edges in G

IND-SET $\leq_{\text{poly}}$ CLIQUE

We reduce from IND-SET to CLIQUE as follows:

1. **Input:** A graph G with V vertices and E edges, and an integer k
2. Create the **complement graph** $\overline{G}$ by reversing all of the edges in G
3. Check if $\overline{G}$ has a clique of size k. If so, accept $\langle G, k\rangle$; otherwise reject

IND-SET $\leq_{\text{poly}}$ CLIQUE



IND-SET $\leq_{\text{poly}}$ CLIQUE

We reduce from IND-SET to CLIQUE as follows:

1. **Input:** A graph G with V vertices and E edges, and an integer k
2. Create the **complement graph** $\overline{G}$ by reversing all of the edges in G
3. Check if $\overline{G}$ has a clique of size k. If so, accept $\langle G, k \rangle$; otherwise reject

Poly-time: $O(E)$ to construct $\overline{G}$

IND-SET $\leq_{\text{poly}}$ CLIQUE

We reduce from IND-SET to CLIQUE as follows:

1. **Input:** A graph G with V vertices and E edges, and an integer k
2. Create the **complement graph** $\overline{G}$ by reversing all of the edges in G
3. Check if $\overline{G}$ has a clique of size k. If so, accept $\langle G, k \rangle$; otherwise reject

"YES maps to YES": If G has a k-independent set, then those same vertices will all be connected in $\overline{G}$

IND-SET $\leq_{\text{poly}}$ CLIQUE

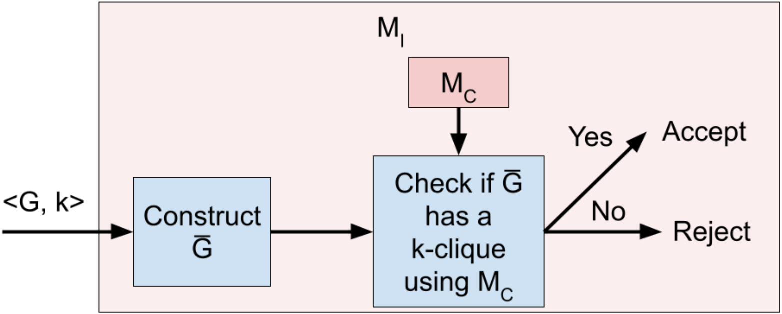
We reduce from IND-SET to CLIQUE as follows:

1. **Input:** A graph G with V vertices and E edges, and an integer k
2. Create the **complement graph** $\overline{G}$ by reversing all of the edges in G
3. Check if $\overline{G}$ has a clique of size k. If so, accept $\langle G, k \rangle$; otherwise reject

"NO maps to NO": If G doesn't have a k-independent set, then every set of k vertices has at least one edge. Those same vertices will be missing an edge in $\overline{G}$

IND-SET $\leq_{\text{poly}}$ CLIQUE

IND-SET $\leq_{\text{poly}}$ CLIQUE



If we can decide CLIQUE in poly-time,
we can decide IND-SET in poly-time

3-SAT $\leq_{\text{poly}}$ IND-SET

3-SAT $\leq_{\text{poly}}$ IND-SET

We reduce from IND-SET to CLIQUE as follows:

3-SAT $\leq_{\text{poly}}$ IND-SET

We reduce from IND-SET to CLIQUE as follows:

1. **Input:** a 3-CNF formula with n variables and m clauses

3-SAT $\leq_{\text{poly}}$ IND-SET

We reduce from IND-SET to CLIQUE as follows:

1. **Input:** a 3-CNF formula with n variables and m clauses
2. Create a graph G

3-SAT $\leq_{\text{poly}}$ IND-SET

We reduce from IND-SET to CLIQUE as follows:

1. **Input:** a 3-CNF formula with n variables and m clauses
2. Create a graph G
3. For each clause $(x \vee y \vee z)$, create three nodes x, y, z and connect them to form a "triangle"

3-SAT $\leq_{\text{poly}}$ IND-SET

We reduce from IND-SET to CLIQUE as follows:

1. **Input:** a 3-CNF formula with n variables and m clauses
2. Create a graph G
3. For each clause $(x \vee y \vee z)$, create three nodes x, y, z and connect them to form a "triangle"
4. If there are nodes x and $\neg x$, connect them with an edge

3-SAT $\leq_{\text{poly}}$ IND-SET

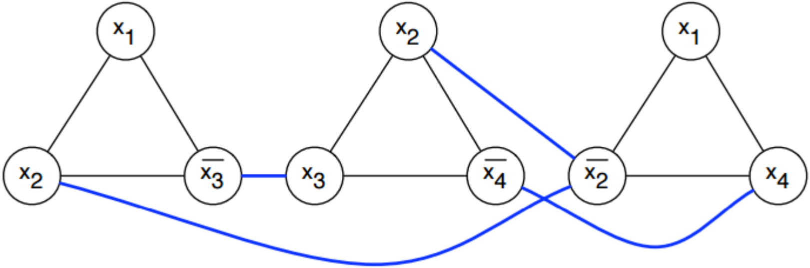
We reduce from IND-SET to CLIQUE as follows:

1. **Input:** a 3-CNF formula with n variables and m clauses
2. Create a graph G
3. For each clause $(x \vee y \vee z)$, create three nodes x, y, z and connect them to form a "triangle"
4. If there are nodes x and $\neg x$, connect them with an edge
5. Check if there is an independent set of size m

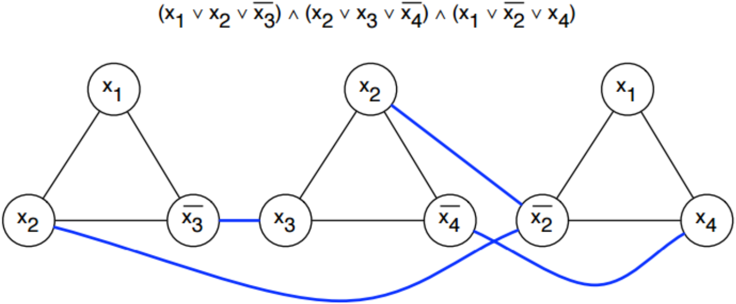
3-SAT $\leq_{\text{poly}}$ IND-SET

We reduce from IND-SET to CLIQUE as follows:

$$(x_1 \vee x_2 \vee \overline{x_3}) \wedge (x_2 \vee x_3 \vee \overline{x_4}) \wedge (x_1 \vee \overline{x_2} \vee x_4)$$



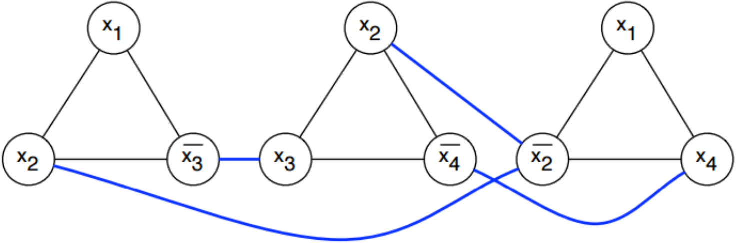
3-SAT $\leq_{\text{poly}}$ IND-SET: poly-time



- $O(m)$ vertices

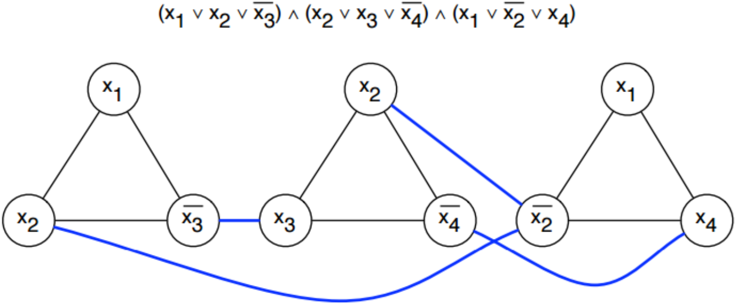
3-SAT $\leq_{\text{poly}}$ IND-SET: poly-time

$(x_1 \vee x_2 \vee \overline{x_3}) \wedge (x_2 \vee x_3 \vee \overline{x_4}) \wedge (x_1 \vee \overline{x_2} \vee x_4)$



- $O(m)$ vertices
- $O(m) + O(n^2)$ edges

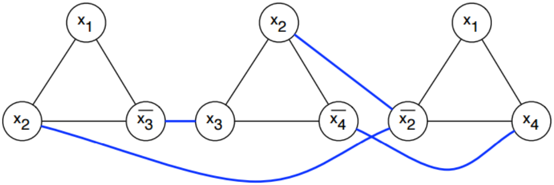
3-SAT $\leq_{\text{poly}}$ IND-SET: poly-time



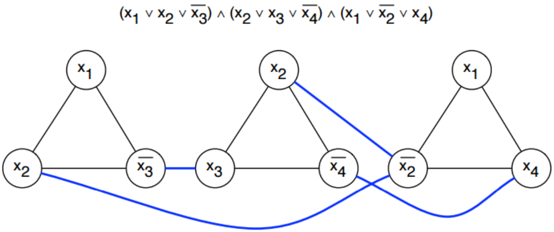
- $O(m)$ vertices
- $O(m) + O(n^2)$ edges
- $O(m) + O(n^2) =$ poly-time

3-SAT $\leq_{\text{poly}}$ IND-SET: yes $\rightarrow$ yes

$$(x_1 \vee x_2 \vee \overline{x_3}) \wedge (x_2 \vee x_3 \vee \overline{x_4}) \wedge (x_1 \vee \overline{x_2} \vee x_4)$$



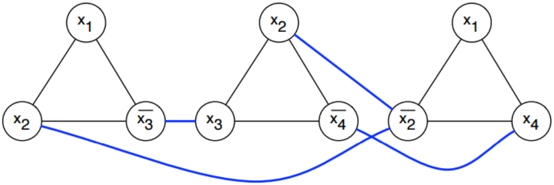
3-SAT $\leq_{\text{poly}}$ IND-SET: yes $\rightarrow$ yes



- Suppose F has a satisfying assignment

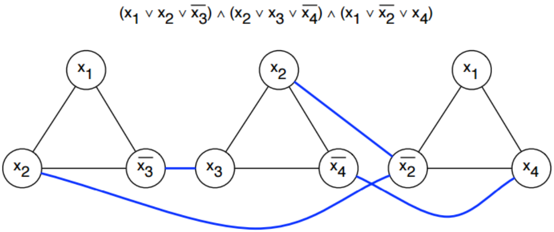
3-SAT $\leq_{\text{poly}}$ IND-SET: yes $\rightarrow$ yes

$(x_1 \vee x_2 \vee \overline{x_3}) \wedge (x_2 \vee x_3 \vee \overline{x_4}) \wedge (x_1 \vee \overline{x_2} \vee x_4)$



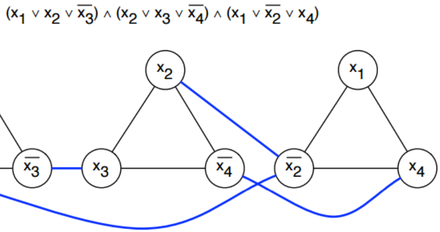
- Suppose F has a satisfying assignment
- For each "triangle", pick one of the TRUE vertices to be in the independent set

3-SAT $\leq_{\text{poly}}$ IND-SET: yes $\rightarrow$ yes



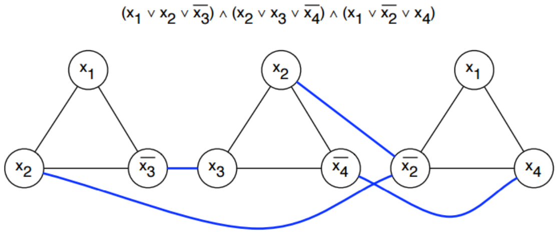
- Suppose F has a satisfying assignment
- For each "triangle", pick one of the TRUE vertices to be in the independent set
  - Every clause has at least one true variable

3-SAT $\leq_{\text{poly}}$ IND-SET: yes $\to$ yes



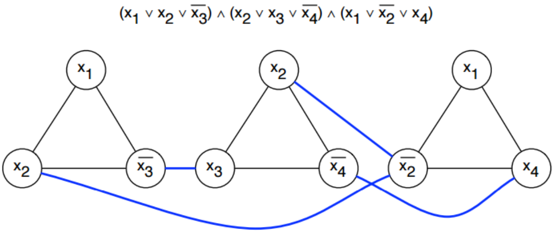
- Suppose F has a satisfying assignment
- For each "triangle", pick one of the TRUE vertices to be in the independent set
  - Every clause has at least one true variable
  - Variables from different clauses are not connected

3-SAT $\leq_{\text{poly}}$ IND-SET: yes → yes



- Suppose F has a satisfying assignment
- For each "triangle", pick one of the TRUE vertices to be in the independent set
  - Every clause has at least one true variable
  - Variables from different clauses are not connected
  - Truth assignment will not let us pick x and $\neg x$

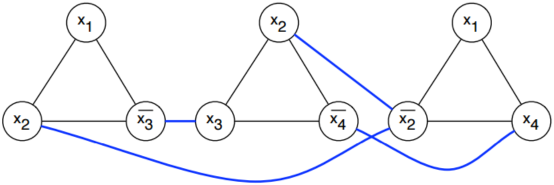
3-SAT $\leq_{\text{poly}}$ IND-SET: yes $\rightarrow$ yes



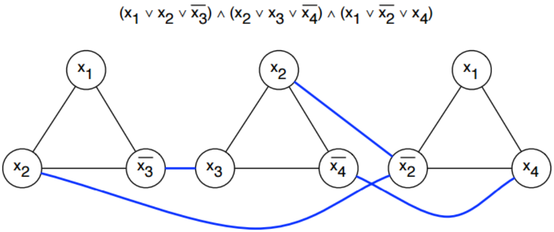
- Suppose F has a satisfying assignment
- For each "triangle", pick one of the TRUE vertices to be in the independent set
  - Every clause has at least one true variable
  - Variables from different clauses are not connected
  - Truth assignment will not let us pick x and $\neg x$
- m clauses $\rightarrow$ m triangles $\rightarrow$ m-independent set

3-SAT $\leq_{poly}$ IND-SET: no → no

$(x_1 \vee x_2 \vee \overline{x_3}) \wedge (x_2 \vee x_3 \vee \overline{x_4}) \wedge (x_1 \vee \overline{x_2} \vee x_4)$

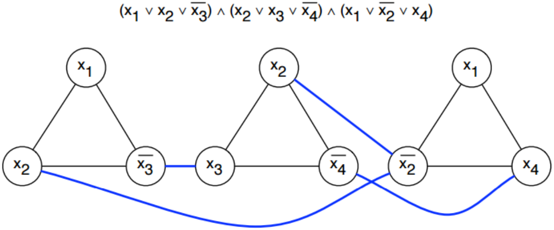


3-SAT $\leq_{\text{poly}}$ IND-SET: no $\rightarrow$ no



Show the contrapositive: yes $\leftarrow$ yes

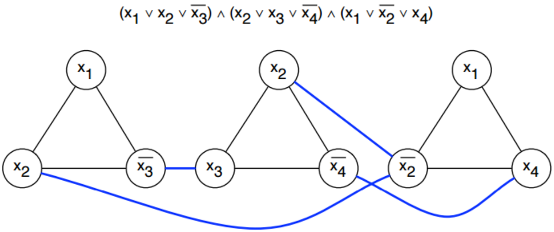
3-SAT $\leq_{\text{poly}}$ IND-SET: no $\rightarrow$ no



Show the contrapositive: yes $\leftarrow$ yes

- Suppose G has a an independent set of size m

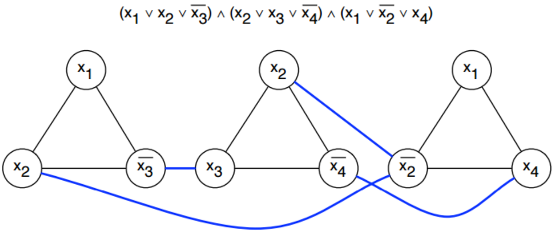
3-SAT $\leq_{\text{poly}}$ IND-SET: no $\rightarrow$ no



Show the contrapositive: <u>yes $\leftarrow$ yes</u>

- Suppose G has a an independent set of size m
- Set the variables that are part of the independent set to be TRUE

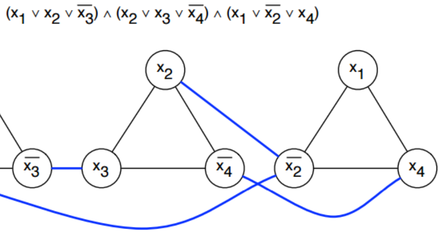
3-SAT $\leq_{\text{poly}}$ IND-SET: no $\rightarrow$ no



Show the contrapositive: yes $\leftarrow$ yes

- Suppose G has a an independent set of size m
- Set the variables that are part of the independent set to be TRUE
  - There must be one vertex from each "triangle" in the set, so every clause will be satisfied

3-SAT $\leq_{\text{poly}}$ IND-SET: no $\rightarrow$ no



Show the contrapositive: <u>yes $\leftarrow$ yes</u>

- Suppose G has a an independent set of size m
- Set the variables that are part of the independent set to be TRUE
  - There must be one vertex from each "triangle" in the set, so every clause will be satisfied
  - x and $\neg x$ are connected, so our independent set will not include a contradictory assignment

NP-completeness

NP-completeness

- **Def:** A language L is **NP-Hard** if *every* language in NP is poly-time reducible to L

NP-completeness

- **Def:** A language L is **NP-Hard** if *every* language in NP is poly-time reducible to L
  - $A \in \mathrm{NP} \implies A \leq_{\mathrm{poly}} L$

NP-completeness

- **Def:** A language L is **NP-Hard** if *every* language in NP is poly-time reducible to L
  - $A \in \mathrm{NP} \implies A \leq_{\mathrm{poly}} L$
- **Def:** L is NP-complete if:

NP-completeness

- **Def:** A language L is **NP-Hard** if *every* language in NP is poly-time reducible to L
  - $A \in \mathrm{NP} \implies A \leq_{\mathrm{poly}} L$
- **Def:** L is NP-complete if:
  1. $L \in \mathrm{NP}$

NP-completeness

- **Def:** A language L is **NP-Hard** if *every* language in NP is poly-time reducible to L
  - $A \in \mathrm{NP} \implies A \leq_{\mathrm{poly}} L$
- **Def:** L is NP-complete if:
  1. $L \in \mathrm{NP}$
  2. L is NP-Hard

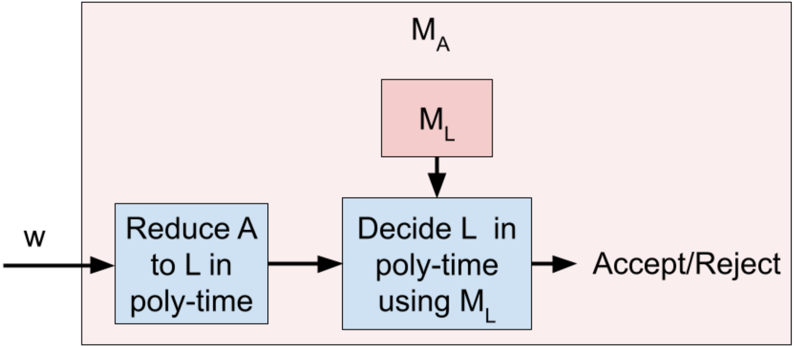
NP-completeness

- **Def:** A language L is **NP-Hard** if *every* language in NP is poly-time reducible to L
  - $A \in \mathrm{NP} \implies A \leq_{\mathrm{poly}} L$
- **Def:** L is NP-complete if:
  1. $L \in \mathrm{NP}$
  2. L is NP-Hard
- L is the "hardest" or "most expressive" problem in NP

NP-completeness

L is NP-Hard

$A \in NP$



If we can decide L in poly-time, we can decide *every* NP language in poly-time!

3-SAT is NP-complete

$\mathrm{3\text{-}SAT}$ is NP-complete

Cook-Levin theorem: $\mathrm{CIRCUIT-SAT}$ is NP-complete

3-SAT is NP-complete

Cook-Levin theorem: $\text{CIRCUIT} - \text{SAT}$ is NP-complete

- Like 3-SAT, but we can use any combination of $\neg, \vee, \wedge$

3-SAT is NP-complete

Cook-Levin theorem: $\mathrm{CIRCUIT} - \mathrm{SAT}$ is NP-complete

- Like 3-SAT, but we can use any combination of $\neg, \vee, \wedge$
- **Proof idea:** create a boolean circuit that checks if the input string eventually leads to an accepting computation history

3-SAT is NP-complete

Cook-Levin theorem: $\text{CIRCUIT} - \text{SAT}$ is NP-complete

- Like 3-SAT, but we can use any combination of $\neg, \vee, \wedge$
- **Proof idea:** create a boolean circuit that checks if the input string eventually leads to an accepting computation history

Karp's theorem: 3-SAT is NP-complete

3-SAT is NP-complete

Cook-Levin theorem: $\text{CIRCUIT} - \text{SAT}$ is NP-complete

- Like 3-SAT, but we can use any combination of $\neg, \vee, \wedge$
- **Proof idea:** create a boolean circuit that checks if the input string eventually leads to an accepting computation history

Karp's theorem: 3-SAT is NP-complete

- Every boolean circuit can be converted to a 3-CNF circuit

3-SAT is NP-complete

Cook-Levin theorem: $\text{CIRCUIT} - \text{SAT}$ is NP-complete

- Like 3-SAT, but we can use any combination of $\neg, \vee, \wedge$
- **Proof idea:** create a boolean circuit that checks if the input string eventually leads to an accepting computation history

Karp's theorem: 3-SAT is NP-complete

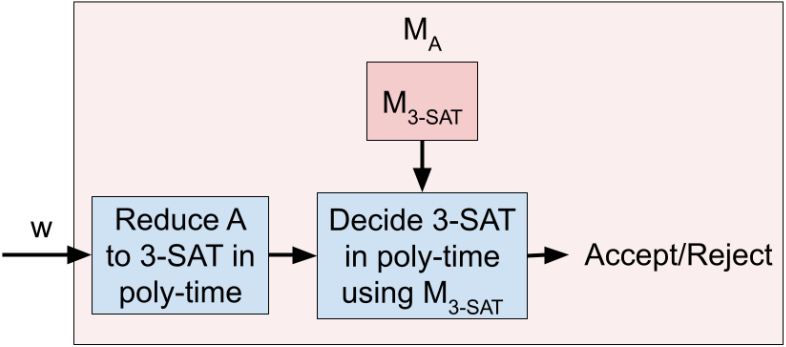
- Every boolean circuit can be converted to a 3-CNF circuit

See Sipser for full proof

3-SAT is NP-complete

3-SAT is NP-Complete

A ∈ NP



If we can decide 3-SAT in poly-time, we can decide *every* NP language in poly-time!

Transitivity of $\leq_{\text{poly}}$

Transitivity of $\leq_{\mathrm{poly}}$

Proposition: If $A \leq_{\mathrm{poly}} B$ and $B \leq_{\mathrm{poly}} C$, then $A \leq_{\mathrm{poly}} C$

Transitivity of $\leq_{\text{poly}}$

Proposition: If $A \leq_{\text{poly}} B$ and $B \leq_{\text{poly}} C$, then $A \leq_{\text{poly}} C$

- There exists a poly-time computable function f such that $w \in A \Leftrightarrow f(w) \in B$

Transitivity of $\leq_{\text{poly}}$

Proposition: If $A \leq_{\text{poly}} B$ and $B \leq_{\text{poly}} C$, then $A \leq_{\text{poly}} C$

- There exists a poly-time computable function f such that $w \in A \Leftrightarrow f(w) \in B$
- There exists a poly-time computable function g such that $w \in B \Leftrightarrow g(w) \in C$

Transitivity of $\leq_{\text{poly}}$

Proposition: If $A \leq_{\text{poly}} B$ and $B \leq_{\text{poly}} C$, then $A \leq_{\text{poly}} C$

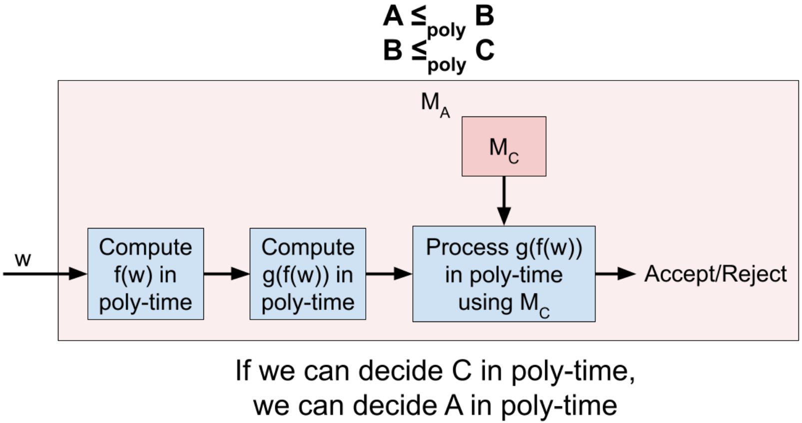
- There exists a poly-time computable function f such that $w \in A \Leftrightarrow f(w) \in B$
- There exists a poly-time computable function g such that $w \in B \Leftrightarrow g(w) \in C$
- $w \in A \Leftrightarrow f(w) \in B \Leftrightarrow g(f(w)) \in C$

Transitivity of $\leq_{\text{poly}}$

Proposition: If $A \leq_{\text{poly}} B$ and $B \leq_{\text{poly}} C$, then $A \leq_{\text{poly}} C$

- There exists a poly-time computable function f such that $w \in A \Leftrightarrow f(w) \in B$
- There exists a poly-time computable function g such that $w \in B \Leftrightarrow g(w) \in C$
- $w \in A \Leftrightarrow f(w) \in B \Leftrightarrow g(f(w)) \in C$
- $g \circ f$ is a poly-time reduction from A to C!

Transitivity of $\leq_{\text{poly}}$



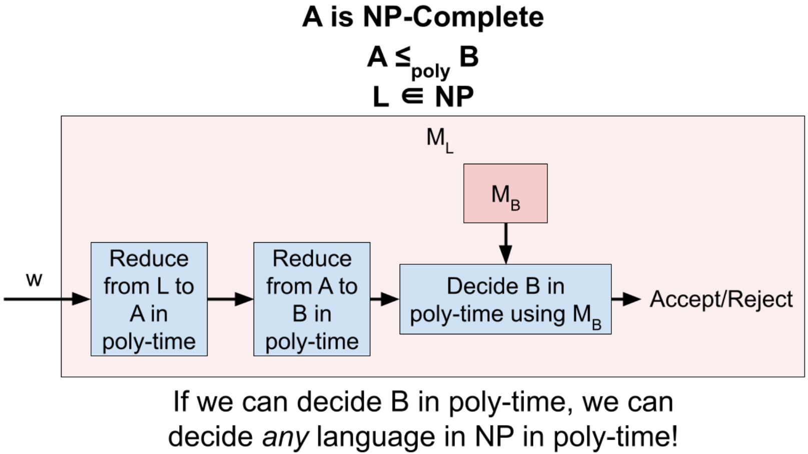
Transitivity of NP-Completeness

Transitivity of NP-Completeness

Corollary: If A is NP-complete, and $A \leq_{\mathrm{poly}} B$, then B is NP-complete

Transitivity of NP-Completeness

Corollary: If A is NP-complete, and $A \leq_{\text{poly}} B$, then B is NP-complete



Implications of 3-SAT NP-Completeness

Implications of 3-SAT NP-Completeness

- **We can use 3-SAT to prove that other languages are NP-complete!**

Implications of 3-SAT NP-Completeness

- **We can use 3-SAT to prove that other languages are NP-complete!**
  - If we can show that $3\text{-SAT} \leq_{\text{poly}} L$, it follows that L is also complete!

Implications of 3-SAT NP-Completeness

- **We can use 3-SAT to prove that other languages are NP-complete!**
  - If we can show that 3-SAT $\leq_{\text{poly}} L$, it follows that L is also complete!
- And we can use those other languages to show that even more languages are NP-complete

Implications of 3-SAT NP-Completeness

IND-SET is NP-Complete

$\mathrm{IND\text{-}SET}$ is NP-Complete

- $3\text{-}\mathrm{SAT}$ is known to be NP-complete

$\mathrm{IND\text{-}SET}$ is NP-Complete

- $\mathrm{3\text{-}SAT}$ is known to be NP-complete
- We proved that $\mathrm{3\text{-}SAT} \leq_{\mathrm{poly}} \mathrm{IND\text{-}SET}$

IND-SET is NP-Complete

- 3-SAT is known to be NP-complete
- We proved that 3-SAT $\leq_{\text{poly}}$ IND-SET
- Thus, IND-SET is NP-complete

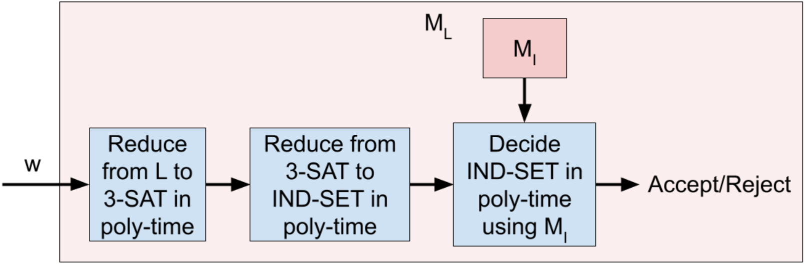
IND-SET is NP-Complete

- 3-SAT is known to be NP-complete
- We proved that 3-SAT $\leq_{\text{poly}}$ IND-SET
- Thus, IND-SET is NP-complete

3-SAT is NP-Complete

3-SAT $\leq_{poly}$ IND-SET

L $\in$ NP



If we can decide IND-SET in poly-time, we can decide *any* language in NP in poly-time!

CLIQUE is NP-Complete

CLIQUE is NP-Complete

- IND-SET is known to be NP-Complete

CLIQUE is NP-Complete

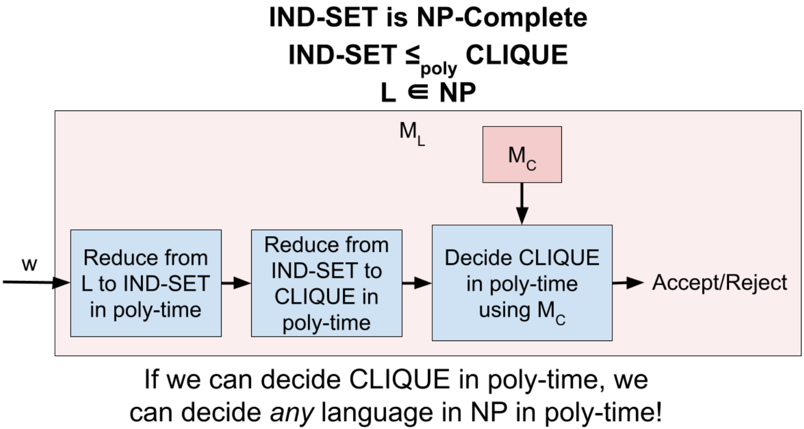
- IND-SET is known to be NP-Complete
- We proved that IND-SET $\leq_{\text{poly}}$ CLIQUE

CLIQUE is NP-Complete

- IND-SET is known to be NP-Complete
- We proved that IND-SET $\leq_{\text{poly}}$ CLIQUE
- Thus, CLIQUE is NP-Complete

CLIQUE is NP-Complete

- IND-SET is known to be NP-Complete
- We proved that IND-SET $\leq_{\text{poly}}$ CLIQUE
- Thus, CLIQUE is NP-Complete



SUBSET-SUM is NP-Complete

SUBSET-SUM is NP-Complete

Proof: Reduce from 3-SAT

SUBSET-SUM is NP-Complete

Proof: Reduce from 3-SAT

1. We will create a number for each variable x_i and its negation

SUBSET-SUM is NP-Complete

Proof: Reduce from 3-SAT

1. We will create a number for each variable x_i and its negation
   - The digits of the number correspond to which clauses that variable can satisfy

SUBSET-SUM is NP-Complete

Proof: Reduce from 3-SAT

1. We will create a number for each variable x_i and its negation
   - The digits of the number correspond to which clauses that variable can satisfy
2. We will set the target sum such that it can only be reached through a satisfying assignment

SUBSET-SUM is NP-Complete

Proof: Reduce from 3-SAT

1. We will create a number for each variable x_i and its negation
   - The digits of the number correspond to which clauses that variable can satisfy
2. We will set the target sum such that it can only be reached through a satisfying assignment
   - To reach the target, each clause needs to have at least of its true

SUBSET-SUM is NP-Complete

Proof: Reduce from 3-SAT

1. We will create a number for each variable x_i and its negation
   - The digits of the number correspond to which clauses that variable can satisfy
2. We will set the target sum such that it can only be reached through a satisfying assignment
   - To reach the target, each clause needs to have at least of its true
3. We will set the desired sum such that each clause needs to satisfied

3-SAT $\leq_{\text{poly}}$ SUBSET-SUM: variables

3-SAT $\leq_{\text{poly}}$ SUBSET-SUM: variables

- We want our numbers to correspond to assigning each variable to TRUE or FALSE

3-SAT $\leq_{\text{poly}}$ SUBSET-SUM: variables

- We want our numbers to correspond to assigning each variable to TRUE or FALSE
- For each variable x_i, we will create two numbers: x_i^{TRUE} and x_i^{FALSE}

3-SAT $\leq_{\text{poly}}$ SUBSET-SUM: variables

- We want our numbers to correspond to assigning each variable to TRUE or FALSE
- For each variable x_i, we will create two numbers: x_i^{TRUE} and x_i^{FALSE}
- We will design our desired total so that exactly one of these two numbers must be picked

3-SAT $\leq_{\text{poly}}$ SUBSET-SUM: variables

$$F = (x_1 \lor x_2 \lor \neg x_3) \land (\neg x_1 \lor x_2 \lor \neg x_3) \land \ldots \land (x_2 \lor x_3)$$

n digits (1 per unique variable)

2n numbers (2 per unique variable)

x_1^{TRUE}	= 1	0	0	0	…	0
x_1^{FALSE}	= 1	0	0	0	…	0
x_2^{TRUE}	= 0	1	0	0	…	0
x_2^{FALSE}	= 0	1	0	0	…	0
x_3^{TRUE}	= 0	0	1	0	…	0
x_3^{FALSE}	= 0	0	1	0	…	0
…						
x_n^{TRUE}	= 0	0	0	0	…	1
x_n^{FALSE}	= 0	0	0	0	…	1
B	**= 1**	**1**	**1**	**1**	**…**	**1**

Each variable must be TRUE or FALSE

3-SAT $\leq_{\text{poly}}$ SUBSET-SUM: clauses

3-SAT $\leq_{\text{poly}}$ SUBSET-SUM: clauses

- We want our numbers to correspond to satisfying certain clauses

3-SAT $\leq_{\text{poly}}$ SUBSET-SUM: clauses

- We want our numbers to correspond to satisfying certain clauses
- For each number, we will add an extra digit for each clause

3-SAT $\leq_{\text{poly}}$ SUBSET-SUM: **clauses**

- We want our numbers to correspond to satisfying certain clauses
- For each number, we will add an extra digit for each clause
  - The extra digits signifiy which variables satisfy which clauses

3-SAT $\leq_{\text{poly}}$ SUBSET-SUM: clauses

- We want our numbers to correspond to satisfying certain clauses
- For each number, we will add an extra digit for each clause
  - The extra digits signifiy which variables satisfy which clauses
- We will design our desired total so that (at least) one variable must be picked for each clause

3-SAT $\leq_{\text{poly}}$ SUBSET-SUM: clauses

$F = (x_1 \lor x_2 \lor \neg x_3) \land (\neg x_1 \lor x_2 \lor \neg x_3) \land \ldots \land (x_2 \lor x_3)$

m digits (1 per clause)

							c_1	c_2		c_m
x_1^{TRUE}	= 1	0	0	0	…	0	1	0	…	0
x_1^{FALSE}	= 1	0	0	0	…	0	0	1	…	0
x_2^{TRUE}	= 0	1	0	0	…	0	1	1	…	1
x_2^{FALSE}	= 0	1	0	0	…	0	0	0	…	0
x_3^{TRUE}	= 0	0	1	0	…	0	0	0	…	1
x_3^{FALSE}	= 0	0	1	0	…	0	1	1	…	0
…							…			
x_n^{TRUE}	= 0	0	0	0	…	1	0	0	…	0
x_n^{FALSE}	= 0	0	0	0	…	1	0	0	…	0
B	**= 1**	**1**	**1**	**1**	**…**	**1**	**?**	**?**	**…**	**?**

Which variables satisfy which clauses?

How do we ensure that each clause is satisfied?

3-SAT $\leq_{\text{poly}}$ SUBSET-SUM: clauses

3-SAT $\leq_{\text{poly}}$ SUBSET-SUM: clauses

- How do we design our target B so that each clause must be satisfied?

3-SAT $\leq_{\text{poly}}$ SUBSET-SUM: clauses

- How do we design our target B so that each clause must be satisfied?
- **Attempt 1:** Include a 1 digit for each clause

3-SAT $\leq_{\text{poly}}$ SUBSET-SUM: clauses

- How do we design our target B so that each clause must be satisfied?
- **Attempt 1:** Include a 1 digit for each clause
  - **Problem:** What if a clause has more than one TRUE variable?

3-SAT $\leq_{\text{poly}}$ SUBSET-SUM: clauses

- How do we design our target B so that each clause must be satisfied?
- **Attempt 1:** Include a 1 digit for each clause
  - **Problem:** What if a clause has more than one TRUE variable?
- **Attempt 2:** Include a 3 digit for each clause

3-SAT $\leq_{\text{poly}}$ SUBSET-SUM: clauses

- How do we design our target B so that each clause must be satisfied?
- **Attempt 1:** Include a 1 digit for each clause
  - **Problem:** What if a clause has more than one TRUE variable?
- **Attempt 2:** Include a 3 digit for each clause
  - **Problem:** A satisfied clause might have only 1 or 2 TRUE variables

3-SAT $\leq_{\text{poly}}$ SUBSET-SUM: clauses

- How do we design our target B so that each clause must be satisfied?
- **Attempt 1:** Include a 1 digit for each clause
  - **Problem:** What if a clause has more than one TRUE variable?
- **Attempt 2:** Include a 3 digit for each clause
  - **Problem:** A satisfied clause might have only 1 or 2 TRUE variables
- How do we represent “between 1 and 3” when subset sum requires an exact total?

3-SAT $\leq_{\text{poly}}$ SUBSET-SUM: clauses

- How do we design our target B so that each clause must be satisfied?
- **Attempt 1:** Include a 1 digit for each clause
  - **Problem:** What if a clause has more than one TRUE variable?
- **Attempt 2:** Include a 3 digit for each clause
  - **Problem:** A satisfied clause might have only 1 or 2 TRUE variables
- How do we represent "between 1 and 3" when subset sum requires an exact total?
- We will introduce **filler numbers**

3-SAT $\leq_{\text{poly}}$ SUBSET-SUM: fillers

3-SAT $\leq_{\text{poly}}$ SUBSET-SUM: fillers

- For each clause, introduce two *fillers*

3-SAT $\leq_{\text{poly}}$ SUBSET-SUM: fillers

- For each clause, introduce two *fillers*
- From a given clause, if at least one variable is TRUE, we can use up to two fillers to bring the total for that clause to 3

3-SAT $\leq_{\text{poly}}$ SUBSET-SUM: fillers

- For each clause, introduce two *fillers*
- From a given clause, if at least one variable is TRUE, we can use up to two fillers to bring the total for that clause to 3
- If all variables in a clause are FALSE, then that clause will never add up to 3 (even with the fillers)

3-SAT $\leq_{\text{poly}}$ SUBSET-SUM: fillers

$F = (x_1 \lor x_2 \lor \neg x_3) \land (\neg x_1 \lor x_2 \lor \neg x_3) \land \ldots \land (x_2 \lor x_3)$

x_1^{TRUE}	= 1	0	0	0	…	0	1	0	…	0
x_1^{FALSE}	= 1	0	0	0	…	0	0	1	…	0
x_2^{TRUE}	= 0	1	0	0	…	0	1	1	…	1
x_2^{FALSE}	= 0	1	0	0	…	0	0	0	…	0
x_3^{TRUE}	= 0	0	1	0	…	0	0	0	…	1
x_3^{FALSE}	= 0	0	1	0	…	0	1	1	…	0
…							…			
x_n^{TRUE}	= 0	0	0	0	…	1	0	0	…	0
x_n^{FALSE}	= 0	0	0	0	…	1	0	0	…	0
$fill_{11}$	= 0	0	0	0	…	0	1	0	…	0
$fill_{12}$	= 0	0	0	0	…	0	1	0	…	0
$fill_{21}$	= 0	0	0	0	…	0	0	1	…	0
$fill_{22}$	= 0	0	0	0	…	0	0	1	…	0
…										
$fill_{m1}$	= 0	0	0	0	…	0	0	0	…	1
$fill_{m2}$	= 0	0	0	0	…	0	0	0	…	1
B	**= 1**	**1**	**1**	**1**	**…**	**1**	**3**	**3**	**…**	**3**

Fillers don't affect variable truth assignments

Can use up to 2 fillers per clause

Need ≥ 1 TRUE variable + ≤ 2 fillers

3-SAT $\leq_{\text{poly}}$ SUBSET-SUM

$F = (x_1 \vee x_2 \vee \neg x_3) \wedge (\neg x_1 \vee x_2 \vee \neg x_3) \wedge \ldots \wedge (x_2 \vee x_3)$

x_1^{TRUE}	= 1	0	0	0	…	0	1	0	…	0
x_1^{FALSE}	= 1	0	0	0	…	0	0	1	…	0
x_2^{TRUE}	= 0	1	0	0	…	0	1	1	…	1
x_2^{FALSE}	= 0	1	0	0	…	0	0	0	…	0
x_3^{TRUE}	= 0	0	1	0	…	0	0	0	…	1
x_3^{FALSE}	= 0	0	1	0	…	0	1	1	…	0
…							…			
x_n^{TRUE}	= 0	0	0	0	…	1	0	0	…	0
x_n^{FALSE}	= 0	0	0	0	…	1	0	0	…	0
$fill_{11}$	= 0	0	0	0	…	0	1	0	…	0
$fill_{12}$	= 0	0	0	0	…	0	1	0	…	0
$fill_{21}$	= 0	0	0	0	…	0	0	1	…	0
$fill_{22}$	= 0	0	0	0	…	0	0	1	…	0
…										
$fill_{m1}$	= 0	0	0	0	…	0	0	0	…	1
$fill_{m2}$	= 0	0	0	0	…	0	0	0	…	1
B	**= 1**	**1**	**1**	**1**	**…**	**1**	**3**	**3**	**…**	**3**

3-SAT $\leq_{\text{poly}}$ SUBSET-SUM: poly-time

3-SAT $\leq_{\text{poly}}$ SUBSET-SUM: poly-time

- $O(n)$ "variable" numbers

3-SAT $\leq_{\text{poly}}$ SUBSET-SUM: poly-time

- $O(n)$ "variable" numbers
- $O(m)$ "filler" numbers

3-SAT $\leq_{\text{poly}}$ SUBSET-SUM: poly-time

- $O(n)$ "variable" numbers
- $O(m)$ "filler" numbers
- Each number has $O(n + m)$ base-10 digits

3-SAT $\leq_{\text{poly}}$ SUBSET-SUM: poly-time

- $O(n)$ "variable" numbers
- $O(m)$ "filler" numbers
- Each number has $O(n+m)$ base-10 digits
- $(O(n)+O(m)) \cdot O(n+m) =$ poly-time

3-SAT $\leq_{\text{poly}}$ SUBSET-SUM: poly-time

- $O(n)$ "variable" numbers
- $O(m)$ "filler" numbers
- Each number has $O(n+m)$ base-10 digits
- $(O(n) + O(m)) \cdot O(n+m) =$ poly-time
- **Note:** The length of the numbers would be exponential if we used a unary encoding

3-SAT $\leq_{\text{poly}}$ SUBSET-SUM: poly-time

- $O(n)$ "variable" numbers
- $O(m)$ "filler" numbers
- Each number has $O(n+m)$ base-10 digits
- $(O(n)+O(m)) \cdot O(n+m) =$ poly-time
- **Note:** The length of the numbers would be exponential if we used a unary encoding
  - If we could find a poly-time reduction that uses unary, we would have proven that $\mathrm{P} = \mathrm{NP}$

3-SAT $\leq_{\text{poly}}$ SUBSET-SUM: yes $\rightarrow$ yes

"YES maps to YES":

3-SAT $\leq_{\text{poly}}$ SUBSET-SUM: yes $\to$ yes

"YES maps to YES":

- Suppose F has a satisfying assignment

3-SAT $\leq_{\text{poly}}$ SUBSET-SUM: yes $\rightarrow$ yes

"YES maps to YES":

- Suppose F has a satisfying assignment
- If x_i is assigned TRUE, include x_i^{TRUE} in our subset. Otherwise, include x_i^{FALSE}

3-SAT $\leq_{\text{poly}}$ SUBSET-SUM: yes $\rightarrow$ yes

"YES maps to YES":

- Suppose F has a satisfying assignment
- If x_i is assigned TRUE, include x_i^{TRUE} in our subset. Otherwise, include x_i^{FALSE}
- A variable and its negation will never both be assigned TRUE, so we have a 1 in the first n positions of B

3-SAT $\leq_{\text{poly}}$ SUBSET-SUM: yes $\rightarrow$ yes

"YES maps to YES":

- Suppose F has a satisfying assignment
- If x_i is assigned TRUE, include x_i^{TRUE} in our subset. Otherwise, include x_i^{FALSE}
- A variable and its negation will never both be assigned TRUE, so we have a 1 in the first n positions of B
- Each clause is satisfied, so we have at least 1 in the last m positions of B

3-SAT $\leq_{\text{poly}}$ SUBSET-SUM: yes $\rightarrow$ yes

"YES maps to YES":

- Suppose F has a satisfying assignment
- If x_i is assigned TRUE, include x_i^{TRUE} in our subset. Otherwise, include x_i^{FALSE}
- A variable and its negation will never both be assigned TRUE, so we have a 1 in the first n positions of B
- Each clause is satisfied, so we have at least 1 in the last m positions of B
- Can use up to 2 fillers to get a 3 in the last m positions of B

3-SAT $\leq_{\text{poly}}$ SUBSET-SUM: no $\rightarrow$ no

"NO maps to NO":

3-SAT $\leq_{\text{poly}}$ SUBSET-SUM: no $\rightarrow$ no

"NO maps to NO":

- Suppose F is unsatisfiable

3-SAT $\leq_{\text{poly}}$ SUBSET-SUM: no $\rightarrow$ no

"NO maps to NO":

- Suppose F is unsatisfiable
- Every satisfying assignment will leave at least one clause unsatisfied

3-SAT $\leq_{\text{poly}}$ SUBSET-SUM: no $\rightarrow$ no

"NO maps to NO":

- Suppose F is unsatisfiable
- Every satisfying assignment will leave at least one clause unsatisfied
- One of the last m digits of our subset will add up to at most 2

3-SAT $\leq_{\text{poly}}$ SUBSET-SUM: no $\rightarrow$ no

"<u>NO maps to NO</u>":

- Suppose F is unsatisfiable
- Every satisfying assignment will leave at least one clause unsatisfied
- One of the last m digits of our subset will add up to at most 2
  - Without at least one TRUE variable, we don't have enough fillers to make that clause add up to 3

3-SAT $\leq_{\text{poly}}$ SUBSET-SUM: no $\rightarrow$ no

Alternately, we can prove the contrapositive: yes $\leftarrow$ yes

3-SAT $\leq_{\text{poly}}$ SUBSET-SUM: no $\rightarrow$ no

Alternately, we can prove the contrapositive: yes $\leftarrow$ yes

- Suppose there exists a subset that adds up to B

3-SAT $\leq_{\text{poly}}$ SUBSET-SUM: no $\rightarrow$ no

Alternately, we can prove the contrapositive: yes $\leftarrow$ yes

- Suppose there exists a subset that adds up to B
- Assign all of the variables that are part of the subset to be TRUE

3-SAT $\leq_{\text{poly}}$ SUBSET-SUM: no $\rightarrow$ no

Alternately, we can prove the contrapositive: yes $\leftarrow$ yes

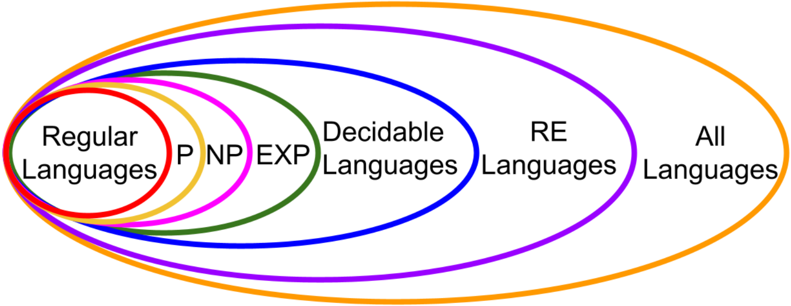
- Suppose there exists a subset that adds up to B
- Assign all of the variables that are part of the subset to be TRUE
- Because the first n digits of B are 1, we won't have a variable and its negation both be TRUE

3-SAT $\leq_{\text{poly}}$ SUBSET-SUM: no $\rightarrow$ no

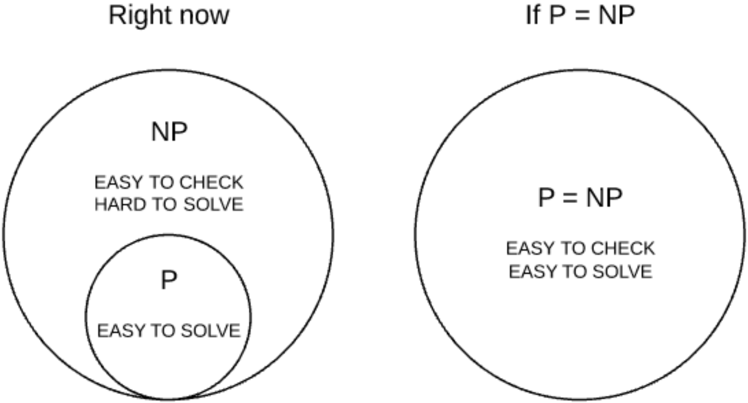
Alternately, we can prove the contrapositive: yes $\leftarrow$ yes

- Suppose there exists a subset that adds up to B
- Assign all of the variables that are part of the subset to be TRUE
- Because the first n digits of B are 1, we won't have a variable and its negation both be TRUE
- Because the last m digits of B are all 3, and there are only 2 fillers per clause, at least one variable is TRUE in each clause

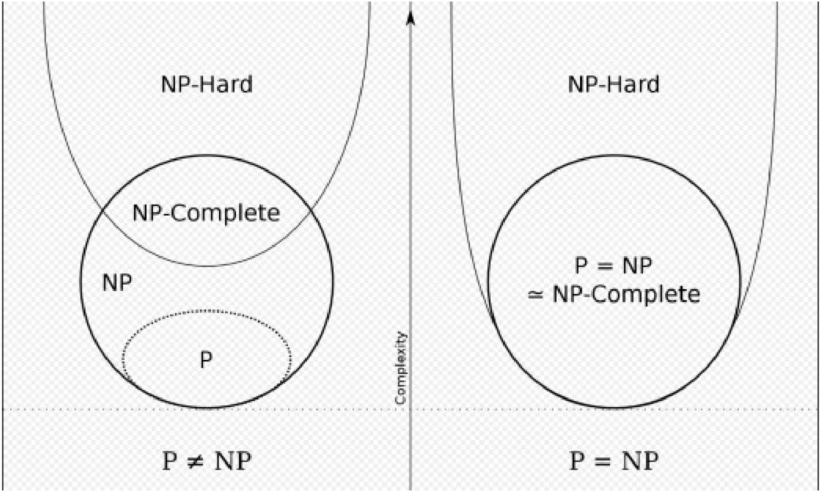
P vs. NP


Regular
Languages
P
NP
EXP
Decidable
Languages
RE
Languages
All
Languages

P vs. NP


Right now
NP
EASY TO CHECK
HARD TO SOLVE
P
EASY TO SOLVE
If P = NP
P = NP
EASY TO CHECK
EASY TO SOLVE

P vs. NP


NP-Hard
NP-Complete
NP
P
Complexity
P ≠ NP
NP-Hard
P = NP
≃ NP-Complete
P = NP

The million dollar question

The million dollar question

Can you design an *efficient* algorithm to find the biggest clique on Facebook?

The million dollar question

Can you design an *efficient* algorithm to find the biggest clique on Facebook?

- If you can do this, then $\mathrm{P} = \mathrm{NP}$

The million dollar question

Can you design an *efficient* algorithm to find the biggest clique on Facebook?

- If you can do this, then $\mathrm{P} = \mathrm{NP}$
- If you believe that $\mathrm{P} \neq \mathrm{NP}$, then this task is impossible

The million dollar question

Can you design an *efficient* algorithm to find the biggest clique on Facebook?

- If you can do this, then $\mathrm{P} = \mathrm{NP}$
- If you believe that $\mathrm{P} \neq \mathrm{NP}$, then this task is impossible

There is a million dollar bounty on the answer to this question!

Beyond P vs. NP: the class $\mathrm{co\text{-}NP}$

Beyond P vs. NP: the class $\mathrm{co\text{-}NP}$

Def: The class $\mathrm{co\text{-}NP}$ is the set of languages whose complement is in NP

Beyond P vs. NP: the class $\mathrm{co\text{-}NP}$

Def: The class $\mathrm{co\text{-}NP}$ is the set of languages whose complement is in NP

- $L \in \mathrm{co\text{-}NP} \Leftrightarrow L^c \in \mathrm{NP}$

Beyond P vs. NP: the class $\mathrm{co\text{-}NP}$

Def: The class $\mathrm{co\text{-}NP}$ is the set of languages whose complement is in NP

- $L \in \mathrm{co\text{-}NP} \Leftrightarrow L^c \in \mathrm{NP}$
- It is easy to verify if $w \notin L$

Beyond P vs. NP: the class $\mathrm{co\text{-}NP}$

Def: The class $\mathrm{co\text{-}NP}$ is the set of languages whose complement is in NP

- $L \in \mathrm{co\text{-}NP} \Leftrightarrow L^c \in \mathrm{NP}$
- It is easy to verify if $w \notin L$
- **Example:** it is very easy to prove that a number is *not* prime (but harder to prove that it is prime)

Beyond P vs. NP: the class co-NP

Def: The class co-NP is the set of languages whose complement is in NP

- $L \in \text{co-NP} \Leftrightarrow L^c \in \mathrm{NP}$
- It is easy to verify if $w \notin L$
- **Example:** it is very easy to prove that a number is *not* prime (but harder to prove that it is prime)
- Some open questions:

Beyond P vs. NP: the class $\mathrm{co\text{-}NP}$

Def: The class $\mathrm{co\text{-}NP}$ is the set of languages whose complement is in NP

- $L \in \mathrm{co\text{-}NP} \Leftrightarrow L^c \in \mathrm{NP}$
- It is easy to verify if $w \notin L$
- **Example:** it is very easy to prove that a number is *not* prime (but harder to prove that it is prime)
- Some open questions:
  - Does $\mathrm{NP} = \mathrm{co\text{-}NP}$?

Beyond P vs. NP: the class $\mathrm{co\text{-}NP}$

Def: The class $\mathrm{co\text{-}NP}$ is the set of languages whose complement is in NP

- $L \in \mathrm{co\text{-}NP} \Leftrightarrow L^c \in \mathrm{NP}$
- It is easy to verify if $w \notin L$
- **Example:** it is very easy to prove that a number is *not* prime (but harder to prove that it is prime)
- Some open questions:
  - Does $\mathrm{NP} = \mathrm{co\text{-}NP}$?
  - Does $P = \mathrm{NP} \cap \mathrm{co\text{-}NP}$ (similar to how decidable $=$ RE $\cap$ co-RE)?

Beyond P vs. NP: the class PSPACE

Beyond P vs. NP: the class PSPACE

Def: The class PSPACE is the set of languages that can be decided using polynomial space/memory

Beyond P vs. NP: the class PSPACE

Def: The class PSPACE is the set of languages that can be decided using polynomial space/memory

- Space complexity is calculated based on how many extra tape squares are needed to process the input

Beyond P vs. NP: the class PSPACE

Def: The class PSPACE is the set of languages that can be decided using polynomial space/memory

- Space complexity is calculated based on how many extra tape squares are needed to process the input
- **Key insight:** Unlike time, space can be *reused*

Beyond P vs. NP: the class PSPACE

Def: The class PSPACE is the set of languages that can be decided using polynomial space/memory

- Space complexity is calculated based on how many extra tape squares are needed to process the input
- **Key insight:** Unlike time, space can be *reused*
- Some open questions:

Beyond P vs. NP: the class PSPACE

Def: The class PSPACE is the set of languages that can be decided using polynomial space/memory

- Space complexity is calculated based on how many extra tape squares are needed to process the input
- **Key insight:** Unlike time, space can be *reused*
- Some open questions:
  - Does $\mathrm{P} = \mathrm{PSPACE}$?

Beyond P vs. NP: the class PSPACE

Def: The class PSPACE is the set of languages that can be decided using polynomial space/memory

- Space complexity is calculated based on how many extra tape squares are needed to process the input
- **Key insight:** Unlike time, space can be *reused*
- Some open questions:
  - Does $\mathrm{P} = \mathrm{PSPACE}$?
  - Does $\mathrm{NP} = \mathrm{PSPACE}$?

Beyond P vs. NP: the class PSPACE

Def: The class PSPACE is the set of languages that can be decided using polynomial space/memory

- Space complexity is calculated based on how many extra tape squares are needed to process the input
- **Key insight:** Unlike time, space can be *reused*
- Some open questions:
  - Does P = PSPACE?
  - Does NP = PSPACE?
  - Does PSPACE = EXP?