# CSE 431/531: Algorithm Analysis and Design (Spring 2022)

# NP-Completeness


Lecturer: Shi Li

*Department of Computer Science and Engineering*
*University at Buffalo*

# NP-Completeness Theory

- The topics we discussed so far are positive results: how to design efficient algorithms for solving a given problem.
- NP-Completeness provides negative results: some problems can not be solved efficiently.

**Q:** Why do we study negative results?

# NP-Completeness Theory

- The topics we discussed so far are positive results: how to design efficient algorithms for solving a given problem.
- NP-Completeness provides negative results: some problems can not be solved efficiently.

**Q:** Why do we study negative results?

- A given problem $X$ cannot be solved in polynomial time.

# NP-Completeness Theory

- The topics we discussed so far are positive results: how to design efficient algorithms for solving a given problem.
- NP-Completeness provides negative results: some problems can not be solved efficiently.

**Q:** Why do we study negative results?

- A given problem $X$ cannot be solved in polynomial time.
- Without knowing it, you will have to keep trying to find polynomial time algorithm for solving $X$. All our efforts are doomed!

# Efficient = Polynomial Time

- Polynomial time: $O(n^k)$ for any constant $k > 0$

# Efficient = Polynomial Time

- Polynomial time: $O(n^k)$ for any constant $k > 0$
- Example: $O(n), O(n^2), O(n^{2.5}\log n), O(n^{100})$

# Efficient = Polynomial Time

- Polynomial time: $O(n^k)$ for any constant $k > 0$
- Example: $O(n), O(n^2), O(n^{2.5} \log n), O(n^{100})$
- Not polynomial time: $O(2^n), O(n^{\log n})$

# Efficient = Polynomial Time

- Polynomial time: $O(n^k)$ for any constant $k > 0$
- Example: $O(n), O(n^2), O(n^{2.5}\log n), O(n^{100})$
- Not polynomial time: $O(2^n), O(n^{\log n})$
- Almost all algorithms we learnt so far run in polynomial time

# Efficient = Polynomial Time

- Polynomial time: $O(n^k)$ for any constant $k > 0$
- Example: $O(n), O(n^2), O(n^{2.5}\log n), O(n^{100})$
- Not polynomial time: $O(2^n), O(n^{\log n})$
- Almost all algorithms we learnt so far run in polynomial time

## Reason for Efficient = Polynomial Time

# Efficient = Polynomial Time

- Polynomial time: $O(n^k)$ for any constant $k > 0$
- Example: $O(n), O(n^2), O(n^{2.5}\log n), O(n^{100})$
- Not polynomial time: $O(2^n), O(n^{\log n})$
- Almost all algorithms we learnt so far run in polynomial time

## Reason for Efficient = Polynomial Time

- For natural problems, if there is an $O(n^k)$-time algorithm, then $k$ is small, say 4

# Efficient = Polynomial Time

- Polynomial time: $O(n^k)$ for any constant $k > 0$
- Example: $O(n), O(n^2), O(n^{2.5}\log n), O(n^{100})$
- Not polynomial time: $O(2^n), O(n^{\log n})$
- Almost all algorithms we learnt so far run in polynomial time

## Reason for Efficient = Polynomial Time

- For natural problems, if there is an $O(n^k)$-time algorithm, then $k$ is small, say 4
- A good cut separating problems: for most natural problems, either we have a polynomial time algorithm, or the best algorithm runs in time $\Omega(2^{n^c})$ for some $c$

# Efficient = Polynomial Time

- Polynomial time: $O(n^k)$ for any constant $k > 0$
- Example: $O(n), O(n^2), O(n^{2.5}\log n), O(n^{100})$
- Not polynomial time: $O(2^n), O(n^{\log n})$
- Almost all algorithms we learnt so far run in polynomial time

## Reason for Efficient = Polynomial Time

- For natural problems, if there is an $O(n^k)$-time algorithm, then $k$ is small, say 4
- A good cut separating problems: for most natural problems, either we have a polynomial time algorithm, or the best algorithm runs in time $\Omega(2^{n^c})$ for some $c$
- Do not need to worry about the computational model

# Outline

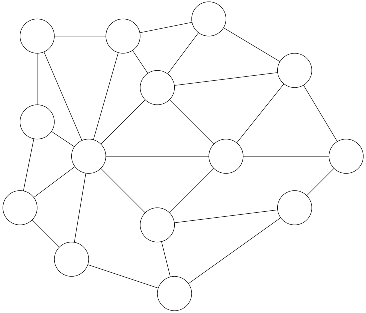
# Example: Hamiltonian Cycle Problem

**Def.** Let $G$ be an undirected graph. A Hamiltonian Cycle (HC) of $G$ is a cycle $C$ in $G$ that passes each vertex of $G$ exactly once.

## Hamiltonian Cycle (HC) Problem

**Input:** graph $G = (V, E)$

**Output:** whether $G$ contains a Hamiltonian cycle



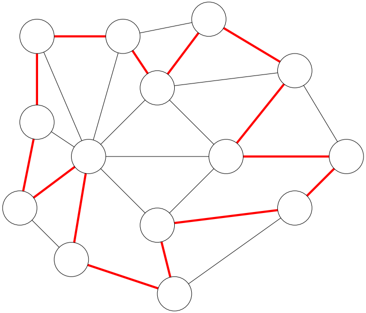
# Example: Hamiltonian Cycle Problem

**Def.** Let $G$ be an undirected graph. A Hamiltonian Cycle (HC) of $G$ is a cycle $C$ in $G$ that passes each vertex of $G$ exactly once.

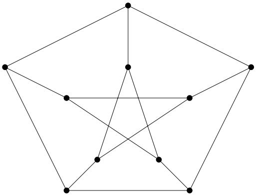
## Hamiltonian Cycle (HC) Problem

**Input:** graph $G = (V, E)$

**Output:** whether $G$ contains a Hamiltonian cycle



# Example: Hamiltonian Cycle Problem



- The graph is called the Petersen Graph. It has no HC.

# Example: Hamiltonian Cycle Problem

## Hamiltonian Cycle (HC) Problem

**Input:** graph $G = (V, E)$

**Output:** whether $G$ contains a Hamiltonian cycle

# Example: Hamiltonian Cycle Problem

### Hamiltonian Cycle (HC) Problem

**Input:** graph $G = (V, E)$

**Output:** whether $G$ contains a Hamiltonian cycle

Algorithm for Hamiltonian Cycle Problem:

- Enumerate all possible permutations, and check if it corresponds to a Hamiltonian Cycle

# Example: Hamiltonian Cycle Problem

**Hamiltonian Cycle (HC) Problem**

**Input:** graph $G = (V, E)$

**Output:** whether $G$ contains a Hamiltonian cycle

Algorithm for Hamiltonian Cycle Problem:

- Enumerate all possible permutations, and check if it corresponds to a Hamiltonian Cycle
- Running time: $O(n!m) = 2^{O(n \lg n)}$

# Example: Hamiltonian Cycle Problem

**Hamiltonian Cycle (HC) Problem**

**Input:** graph $G = (V, E)$

**Output:** whether $G$ contains a Hamiltonian cycle

Algorithm for Hamiltonian Cycle Problem:

- Enumerate all possible permutations, and check if it corresponds to a Hamiltonian Cycle
- Running time: $O(n!m) = 2^{O(n \lg n)}$
- Better algorithm: $2^{O(n)}$

# Example: Hamiltonian Cycle Problem

**Hamiltonian Cycle (HC) Problem**

**Input:** graph $G = (V, E)$

**Output:** whether $G$ contains a Hamiltonian cycle

Algorithm for Hamiltonian Cycle Problem:

- Enumerate all possible permutations, and check if it corresponds to a Hamiltonian Cycle
- Running time: $O(n!m) = 2^{O(n \lg n)}$
- Better algorithm: $2^{O(n)}$
- Far away from polynomial time

# Example: Hamiltonian Cycle Problem

**Hamiltonian Cycle (HC) Problem**

**Input:** graph $G = (V, E)$

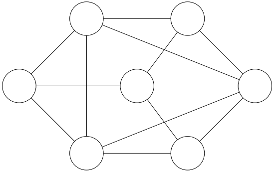
**Output:** whether $G$ contains a Hamiltonian cycle

Algorithm for Hamiltonian Cycle Problem:

- Enumerate all possible permutations, and check if it corresponds to a Hamiltonian Cycle
- Running time: $O(n!m) = 2^{O(n \lg n)}$
- Better algorithm: $2^{O(n)}$
- Far away from polynomial time
- HC is NP-hard: it is unlikely that it can be solved in polynomial time.

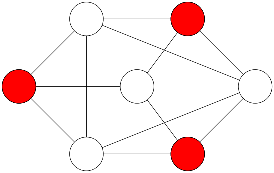
# Maximum Independent Set Problem

**Def.** An independent set of $G = (V, E)$ is a subset $I \subseteq V$ such that no two vertices in $I$ are adjacent in $G$.



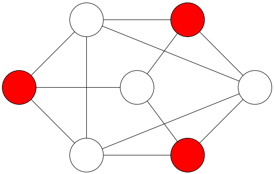
# Maximum Independent Set Problem

**Def.** An independent set of $G = (V, E)$ is a subset $I \subseteq V$ such that no two vertices in $I$ are adjacent in $G$.



# Maximum Independent Set Problem

**Def.** An independent set of $G = (V, E)$ is a subset $I \subseteq V$ such that no two vertices in $I$ are adjacent in $G$.



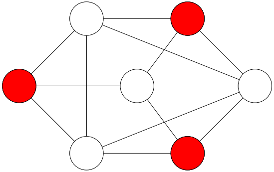
### Maximum Independent Set Problem

**Input:** graph $G = (V, E)$

**Output:** the size of the maximum independent set of $G$

# Maximum Independent Set Problem

**Def.** An independent set of $G = (V, E)$ is a subset $I \subseteq V$ such that no two vertices in $I$ are adjacent in $G$.



**Maximum Independent Set Problem**

**Input:** graph $G = (V, E)$

**Output:** the size of the maximum independent set of $G$

- Maximum Independent Set is NP-hard

# Formula Satisfiability

## Formula Satisfiability

**Input:** boolean formula with $n$ variables, with $\vee, \wedge, \neg$ operators.

**Output:** whether the boolean formula is satisfiable

- Example: $\neg((\neg x_1 \wedge x_2) \vee (\neg x_1 \wedge \neg x_3) \vee x_1 \vee (\neg x_2 \wedge x_3))$ is not satisfiable
- Trivial algorithm: enumerate all possible assignments, and check if each assignment satisfies the formula

# Formula Satisfiability

## Formula Satisfiability

**Input:** boolean formula with $n$ variables, with $\vee, \wedge, \neg$ operators.

**Output:** whether the boolean formula is satisfiable

- Example: $\neg((\neg x_1 \wedge x_2) \vee (\neg x_1 \wedge \neg x_3) \vee x_1 \vee (\neg x_2 \wedge x_3))$ is not satisfiable
- Trivial algorithm: enumerate all possible assignments, and check if each assignment satisfies the formula
- Formula Satisfiablity is NP-hard

# Outline

# Decision Problem Vs Optimization Problem

**Def.** A problem $X$ is called a decision problem if the output is either 0 or 1 (yes/no).

# Decision Problem Vs Optimization Problem

**Def.** A problem $X$ is called a decision problem if the output is either 0 or 1 (yes/no).

- When we define the P and NP, we only consider decision problems.

# Decision Problem Vs Optimization Problem

**Def.** A problem $X$ is called a decision problem if the output is either 0 or 1 (yes/no).

- When we define the P and NP, we only consider decision problems.

**Fact** For each optimization problem $X$, there is a decision version $X'$ of the problem. If we have a polynomial time algorithm for the decision version $X'$, we can solve the original problem $X$ in polynomial time.

# Optimization to Decision

**Shortest Path**

**Input:** graph $G = (V, E)$, weight $w, s, t$ and a bound $L$

**Output:** whether there is a path from $s$ to $t$ of length at most $L$

# Optimization to Decision

## Shortest Path

**Input:** graph $G = (V, E)$, weight $w, s, t$ and a bound $L$

**Output:** whether there is a path from $s$ to $t$ of length at most $L$

## Maximum Independent Set

**Input:** a graph $G$ and a bound $k$

**Output:** whether there is an independent set of size at least $k$

# Encoding

The input of a problem will be encoded as a binary string.

# Encoding

The input of a problem will be encoded as a binary string.

## Example: Sorting problem

# Encoding

The input of a problem will be encoded as a binary string.

## Example: Sorting problem

- Input: (3, 6, 100, 9, 60)

# Encoding

The input of a problem will be encoded as a binary string.

## Example: Sorting problem

- Input: (3, 6, 100, 9, 60)
- Binary: (11, 110, 1100100, 1001, 111100)

# Encoding

The input of a problem will be encoded as a binary string.

### Example: Sorting problem

- Input: (3, 6, 100, 9, 60)
- Binary: (11, 110, 1100100, 1001, 111100)
- String:

# Encoding

The input of a problem will be encoded as a binary string.

## Example: Sorting problem

- Input: (3, 6, 100, 9, 60)
- Binary: (11, 110, 1100100, 1001, 111100)
- String: 111101

# Encoding

The input of a problem will be encoded as a binary string.

## Example: Sorting problem

- Input: (3, 6, 100, 9, 60)
- Binary: (11, 110, 1100100, 1001, 111100)
- String: 11110111110001

# Encoding

The input of a problem will be encoded as a binary string.

## Example: Sorting problem

- Input: (3, 6, 100, 9, 60)
- Binary: (11, 110, 1100100, 1001, 111100)
- String: 111101111100011111000011000001

# Encoding

The input of a problem will be encoded as a binary string.

## Example: Sorting problem

- Input: (3, 6, 100, 9, 60)
- Binary: (11, 110, 1100100, 1001, 111100)
- String: 111101111100011111000011000001
  1100001101

# Encoding

The input of a problem will be encoded as a binary string.

## Example: Sorting problem

- Input: (3, 6, 100, 9, 60)
- Binary: (11, 110, 1100100, 1001, 111100)
- String: 1111011111000111110000011000001 110000110111111111000001

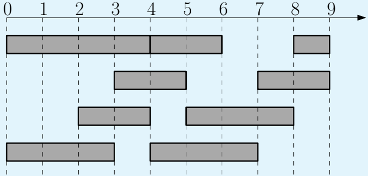
# Encoding

The input of an problem will be encoded as a binary string.

# Encoding

The input of an problem will be encoded as a binary string.

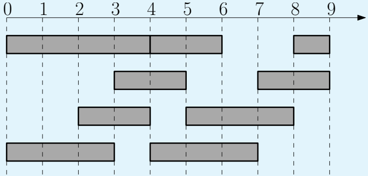
## Example: Interval Scheduling Problem



# Encoding

The input of an problem will be encoded as a binary string.

## Example: Interval Scheduling Problem

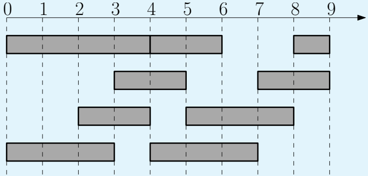



- $(0, 3, 0, 4, 2, 4, 3, 5, 4, 6, 4, 7, 5, 8, 7, 9, 8, 9)$

# Encoding

The input of an problem will be encoded as a binary string.

## Example: Interval Scheduling Problem




- $(0, 3, 0, 4, 2, 4, 3, 5, 4, 6, 4, 7, 5, 8, 7, 9, 8, 9)$
- Encode the sequence into a binary string as before

# Encoding

**Def.** The size of an input is the length of the encoded string $s$ for the input, denoted as $|s|$.

**Q:** Does it matter how we encode the input instances?

# Encoding

**Def.** The size of an input is the length of the encoded string $s$ for the input, denoted as $|s|$.

**Q:** Does it matter how we encode the input instances?

**A:** No! As long as we are using a "natural" encoding. We only care whether the running time is polynomial or not

# Define Problem as a Set

**Def.** A decision problem $X$ is the set of strings on which the output is yes. i.e, $s \in X$ if and only if the correct output for the input $s$ is 1 (yes).

# Define Problem as a Set

**Def.** A decision problem $X$ is the set of strings on which the output is yes. i.e, $s \in X$ if and only if the correct output for the input $s$ is 1 (yes).

**Def.** An algorithm $A$ solves a problem $X$ if, $A(s) = 1$ if and only if $s \in X$.

# Define Problem as a Set

**Def.** A decision problem $X$ is the set of strings on which the output is yes. i.e, $s \in X$ if and only if the correct output for the input $s$ is 1 (yes).

**Def.** An algorithm $A$ solves a problem $X$ if, $A(s) = 1$ if and only if $s \in X$.

**Def.** $A$ has a polynomial running time if there is a polynomial function $p(\cdot)$ so that for every string $s$, the algorithm $A$ terminates on $s$ in at most $p(|s|)$ steps.

# Complexity Class P

**Def.** The complexity class P is the set of decision problems $X$ that can be solved in polynomial time.

# Complexity Class P

**Def.** The complexity class P is the set of decision problems $X$ that can be solved in polynomial time.

- The decision versions of interval scheduling, shortest path and minimum spanning tree all in P.

# Certifier for Hamiltonian Cycle (HC)

- Alice has a supercomputer, fast enough to run the $2^{O(n)}$ time algorithm for HC

# Certifier for Hamiltonian Cycle (HC)

- Alice has a supercomputer, fast enough to run the $2^{O(n)}$ time algorithm for HC
- Bob has a slow computer, which can only run an $O(n^3)$-time algorithm

# Certifier for Hamiltonian Cycle (HC)

- Alice has a supercomputer, fast enough to run the $2^{O(n)}$ time algorithm for HC
- Bob has a slow computer, which can only run an $O(n^3)$-time algorithm

**Q:** Given a graph $G = (V, E)$ with a HC, how can Alice convince Bob that $G$ contains a Hamiltonian cycle?

# Certifier for Hamiltonian Cycle (HC)

- Alice has a supercomputer, fast enough to run the $2^{O(n)}$ time algorithm for HC
- Bob has a slow computer, which can only run an $O(n^3)$-time algorithm

**Q:** Given a graph $G = (V, E)$ with a HC, how can Alice convince Bob that $G$ contains a Hamiltonian cycle?

**A:** Alice gives a Hamiltonian cycle to Bob, and Bob checks if it is really a Hamiltonian cycle of $G$

# Certifier for Hamiltonian Cycle (HC)

- Alice has a supercomputer, fast enough to run the $2^{O(n)}$ time algorithm for HC
- Bob has a slow computer, which can only run an $O(n^3)$-time algorithm

**Q:** Given a graph $G = (V, E)$ with a HC, how can Alice convince Bob that $G$ contains a Hamiltonian cycle?

**A:** Alice gives a Hamiltonian cycle to Bob, and Bob checks if it is really a Hamiltonian cycle of $G$

**Def.** The message Alice sends to Bob is called a certificate, and the algorithm Bob runs is called a certifier.

# Certifier for Independent Set (Ind-Set)

- Alice has a supercomputer, fast enough to run the $2^{O(n)}$ time algorithm for Ind-Set
- Bob has a slow computer, which can only run an $O(n^3)$-time algorithm

# Certifier for Independent Set (Ind-Set)

- Alice has a supercomputer, fast enough to run the $2^{O(n)}$ time algorithm for Ind-Set
- Bob has a slow computer, which can only run an $O(n^3)$-time algorithm

**Q:** Given graph $G = (V, E)$ and integer $k$, such that there is an independent set of size $k$ in $G$, how can Alice convince Bob that there is such a set?

# Certifier for Independent Set (Ind-Set)

- Alice has a supercomputer, fast enough to run the $2^{O(n)}$ time algorithm for Ind-Set
- Bob has a slow computer, which can only run an $O(n^3)$-time algorithm

**Q:** Given graph $G = (V, E)$ and integer $k$, such that there is an independent set of size $k$ in $G$, how can Alice convince Bob that there is such a set?

**A:** Alice gives a set of size $k$ to Bob and Bob checks if it is really a independent set in $G$.

# Certifier for Independent Set (Ind-Set)

- Alice has a supercomputer, fast enough to run the $2^{O(n)}$ time algorithm for Ind-Set
- Bob has a slow computer, which can only run an $O(n^3)$-time algorithm

**Q:** Given graph $G = (V, E)$ and integer $k$, such that there is an independent set of size $k$ in $G$, how can Alice convince Bob that there is such a set?

**A:** Alice gives a set of size $k$ to Bob and Bob checks if it is really a independent set in $G$.

- Certificate: a set of size $k$

# Certifier for Independent Set (Ind-Set)

- Alice has a supercomputer, fast enough to run the $2^{O(n)}$ time algorithm for Ind-Set
- Bob has a slow computer, which can only run an $O(n^3)$-time algorithm

**Q:** Given graph $G = (V, E)$ and integer $k$, such that there is an independent set of size $k$ in $G$, how can Alice convince Bob that there is such a set?

**A:** Alice gives a set of size $k$ to Bob and Bob checks if it is really a independent set in $G$.

- Certificate: a set of size $k$
- Certifier: check if the given set is really an independent set

# Graph Isomorphism

## Graph Isomorphism

**Input:** two graphs $G_1$ and $G_2$,

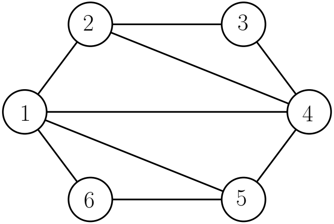
**Output:** whether two graphs are isomorphic to each other

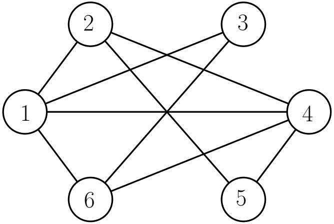
# Graph Isomorphism

## Graph Isomorphism

**Input:** two graphs $G_1$ and $G_2$,

**Output:** whether two graphs are isomorphic to each other



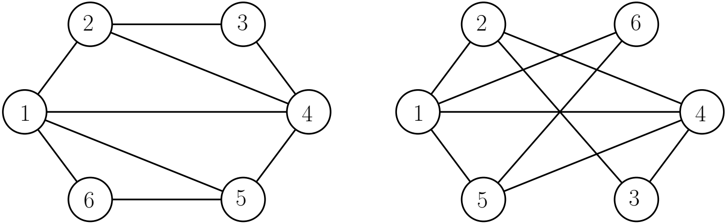



# Graph Isomorphism

## Graph Isomorphism

**Input:** two graphs $G_1$ and $G_2$,

**Output:** whether two graphs are isomorphic to each other

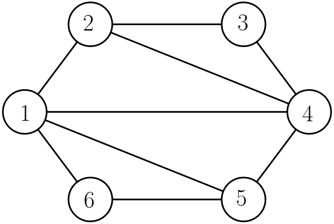


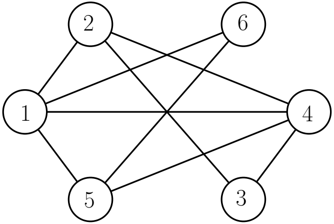
# Graph Isomorphism

## Graph Isomorphism

**Input:** two graphs $G_1$ and $G_2$,

**Output:** whether two graphs are isomorphic to each other



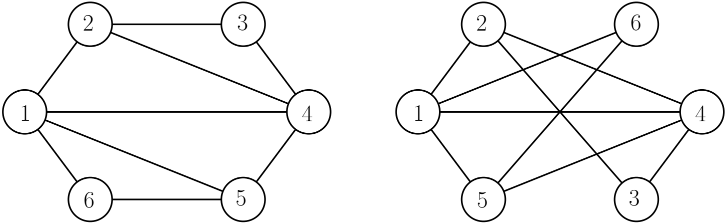

- What is the certificate?

# Graph Isomorphism

## Graph Isomorphism

**Input:** two graphs $G_1$ and $G_2$,

**Output:** whether two graphs are isomorphic to each other




- What is the certificate?
- What is the certifier?

# The Complexity Class NP

**Def.** $B$ is an efficient certifier for a problem $X$ if

- $B$ is a polynomial-time algorithm that takes two input strings $s$ and $t$
- there is a polynomial function $p$ such that, $s \in X$ if and only if there is string $t$ such that $|t| \leq p(|s|)$ and $B(s,t) = 1$.

The string $t$ such that $B(s,t) = 1$ is called a certificate.

# The Complexity Class NP

**Def.** $B$ is an efficient certifier for a problem $X$ if

- $B$ is a polynomial-time algorithm that takes two input strings $s$ and $t$
- there is a polynomial function $p$ such that, $s \in X$ if and only if there is string $t$ such that $|t| \leq p(|s|)$ and $B(s,t) = 1$.

The string $t$ such that $B(s,t) = 1$ is called a certificate.

**Def.** The complexity class NP is the set of all problems for which there exists an efficient certifier.

# Hamiltonian Cycle $\in$ NP

- Input: Graph $G$

# Hamiltonian Cycle $\in$ NP

- Input: Graph $G$
- Certificate: a sequence $S$ of edges in $G$

# Hamiltonian Cycle $\in$ NP

- Input: Graph $G$
- Certificate: a sequence $S$ of edges in $G$
- $|\text{encoding}(S)| \leq p(|\text{encoding}(G)|)$ for some polynomial function $p$

# Hamiltonian Cycle $\in$ NP

- Input: Graph $G$
- Certificate: a sequence $S$ of edges in $G$
- $|\text{encoding}(S)| \le p(|\text{encoding}(G)|)$ for some polynomial function $p$
- Certifier $B$: $B(G, S) = 1$ if and only if $S$ is an HC in $G$

# Hamiltonian Cycle $\in$ NP

- Input: Graph $G$
- Certificate: a sequence $S$ of edges in $G$
- $|\text{encoding}(S)| \leq p(|\text{encoding}(G)|)$ for some polynomial function $p$
- Certifier $B$: $B(G, S) = 1$ if and only if $S$ is an HC in $G$
- Clearly, $B$ runs in polynomial time

# Hamiltonian Cycle $\in$ NP

- Input: Graph $G$
- Certificate: a sequence $S$ of edges in $G$
- $|\text{encoding}(S)| \leq p(|\text{encoding}(G)|)$ for some polynomial function $p$
- Certifier $B$: $B(G, S) = 1$ if and only if $S$ is an HC in $G$
- Clearly, $B$ runs in polynomial time

- $G \in$ HC $\quad\Longleftrightarrow\quad \exists S,\ B(G, S) = 1$

# Graph Isomorphism $\in$ NP

- Input: two graphs $G_1 = (V, E_1)$ and $G_2 = (V, E_2)$ on $V$

# Graph Isomorphism $\in$ NP

- Input: two graphs $G_1 = (V, E_1)$ and $G_2 = (V, E_2)$ on $V$
- Certificate: a 1-1 function $f : V \to V$

# Graph Isomorphism $\in$ NP

- Input: two graphs $G_1 = (V, E_1)$ and $G_2 = (V, E_2)$ on $V$
- Certificate: a 1-1 function $f : V \to V$
- $|\text{encoding}(f)| \leq p(|\text{encoding}(G_1, G_2)|)$ for some polynomial function $p$

# Graph Isomorphism $\in$ NP

- Input: two graphs $G_1 = (V, E_1)$ and $G_2 = (V, E_2)$ on $V$
- Certificate: a 1-1 function $f : V \to V$
- $|\text{encoding}(f)| \le p(|\text{encoding}(G_1, G_2)|)$ for some polynomial function $p$
- Certifier $B$: $B((G_1, G_2), f) = 1$ if and only if for every $u, v \in V$, we have $(u, v) \in E_1 \Leftrightarrow (f(u), f(v)) \in E_2$.

# Graph Isomorphism $\in$ NP

- Input: two graphs $G_1 = (V, E_1)$ and $G_2 = (V, E_2)$ on $V$
- Certificate: a 1-1 function $f : V \to V$
- $|\text{encoding}(f)| \le p(|\text{encoding}(G_1, G_2)|)$ for some polynomial function $p$
- Certifier $B$: $B((G_1, G_2), f) = 1$ if and only if for every $u, v \in V$, we have $(u, v) \in E_1 \Leftrightarrow (f(u), f(v)) \in E_2$.
- Clearly, $B$ runs in polynomial time

# Graph Isomorphism $\in$ NP

- Input: two graphs $G_1 = (V, E_1)$ and $G_2 = (V, E_2)$ on $V$
- Certificate: a 1-1 function $f : V \to V$
- $|\text{encoding}(f)| \le p(|\text{encoding}(G_1, G_2)|)$ for some polynomial function $p$
- Certifier $B$: $B((G_1, G_2), f) = 1$ if and only if for every $u, v \in V$, we have $(u, v) \in E_1 \Leftrightarrow (f(u), f(v)) \in E_2$.
- Clearly, $B$ runs in polynomial time

- $(G_1, G_2) \in \text{GI} \quad \Longleftrightarrow \quad \exists f,\ B((G_1, G_2), f) = 1$

# Maximum Independent Set $\in$ NP

- Input: graph $G = (V, E)$ and integer $k$

# Maximum Independent Set $\in$ NP

- Input: graph $G = (V, E)$ and integer $k$
- Certificate: a set $S \subseteq V$ of size $k$

# Maximum Independent Set $\in$ NP

- Input: graph $G = (V, E)$ and integer $k$
- Certificate: a set $S \subseteq V$ of size $k$
- $|\text{encoding}(S)| \le p(|\text{encoding}(G, k)|)$ for some polynomial function $p$

# Maximum Independent Set $\in$ NP

- Input: graph $G = (V, E)$ and integer $k$
- Certificate: a set $S \subseteq V$ of size $k$
- $|\text{encoding}(S)| \leq p(|\text{encoding}(G, k)|)$ for some polynomial function $p$
- Certifier $B$: $B((G, k), S) = 1$ if and only if $S$ is an independent set in $G$

# Maximum Independent Set $\in$ NP

- Input: graph $G = (V, E)$ and integer $k$
- Certificate: a set $S \subseteq V$ of size $k$
- $|\text{encoding}(S)| \leq p(|\text{encoding}(G, k)|)$ for some polynomial function $p$
- Certifier $B$: $B((G, k), S) = 1$ if and only if $S$ is an independent set in $G$
- Clearly, $B$ runs in polynomial time

# Maximum Independent Set $\in$ NP

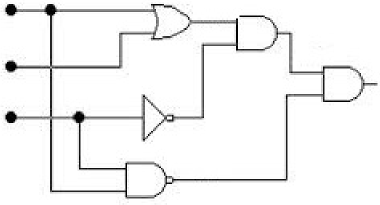
- Input: graph $G = (V, E)$ and integer $k$
- Certificate: a set $S \subseteq V$ of size $k$
- $|\text{encoding}(S)| \leq p(|\text{encoding}(G, k)|)$ for some polynomial function $p$
- Certifier $B$: $B((G, k), S) = 1$ if and only if $S$ is an independent set in $G$
- Clearly, $B$ runs in polynomial time

- $(G, k) \in \text{MIS} \quad \Longleftrightarrow \quad \exists S,\ B((G, k), S) = 1$

## Circuit Satisfiablity (Circuit-Sat) Problem

**Input:** a circuit with and/or/not gates

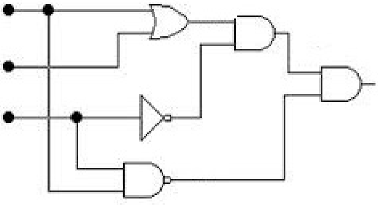
**Output:** whether there is an assignment such that the output is $1$?



## Circuit Satisfiablity (Circuit-Sat) Problem

**Input:** a circuit with and/or/not gates

**Output:** whether there is an assignment such that the output is $1$?



- Is Circuit-Sat $\in$ NP?

### $\overline{\text{HC}}$

**Input:** graph $G = (V, E)$

**Output:** whether $G$ does not contain a Hamiltonian cycle

## $\overline{\text{HC}}$

**Input:** graph $G = (V, E)$

**Output:** whether $G$ does not contain a Hamiltonian cycle

- Is $\overline{\text{HC}} \in \text{NP}$?

## $\overline{\text{HC}}$

**Input:** graph $G = (V, E)$

**Output:** whether $G$ does not contain a Hamiltonian cycle

- Is $\overline{\text{HC}} \in \text{NP}$?
- Can Alice convince Bob that $G$ is a yes-instance (i.e, $G$ does not contain a HC), if this is true.

## $\overline{\text{HC}}$

**Input:** graph $G = (V, E)$

**Output:** whether $G$ does not contain a Hamiltonian cycle

- Is $\overline{\text{HC}} \in \text{NP}$?
- Can Alice convince Bob that $G$ is a yes-instance (i.e, $G$ does not contain a HC), if this is true.
- Unlikely

### $\overline{\text{HC}}$

**Input:** graph $G = (V, E)$

**Output:** whether $G$ does not contain a Hamiltonian cycle

- Is $\overline{\text{HC}} \in \text{NP}$?
- Can Alice convince Bob that $G$ is a yes-instance (i.e, $G$ does not contain a HC), if this is true.
- Unlikely
- Alice can only convince Bob that $G$ is a no-instance

## $\overline{\text{HC}}$

**Input:** graph $G = (V, E)$

**Output:** whether $G$ does not contain a Hamiltonian cycle

- Is $\overline{\text{HC}} \in$ NP?
- Can Alice convince Bob that $G$ is a yes-instance (i.e, $G$ does not contain a HC), if this is true.
- Unlikely
- Alice can only convince Bob that $G$ is a no-instance
- $\overline{\text{HC}} \in$ Co-NP

# The Complexity Class Co-NP

**Def.** For a problem $X$, the problem $\overline{X}$ is the problem such that $s \in \overline{X}$ if and only if $s \notin X$.

**Def.** Co-NP is the set of decision problems $X$ such that $\overline{X} \in$ NP.

**Def.** A tautology is a boolean formula that always evaluates to 1.

## Tautology Problem

**Input:** a boolean formula

**Output:** whether the formula is a tautology

- e.g. $(\neg x_1 \wedge x_2) \vee (\neg x_1 \wedge \neg x_3) \vee x_1 \vee (\neg x_2 \wedge x_3)$ is a tautology

**Def.** A tautology is a boolean formula that always evaluates to 1.

## Tautology Problem

**Input:** a boolean formula

**Output:** whether the formula is a tautology

- e.g. $(\neg x_1 \wedge x_2) \vee (\neg x_1 \wedge \neg x_3) \vee x_1 \vee (\neg x_2 \wedge x_3)$ is a tautology
- Bob can certify that a formula is not a tautology

**Def.** A tautology is a boolean formula that always evaluates to 1.

## Tautology Problem

**Input:** a boolean formula

**Output:** whether the formula is a tautology

- e.g. $(\neg x_1 \wedge x_2) \vee (\neg x_1 \wedge \neg x_3) \vee x_1 \vee (\neg x_2 \wedge x_3)$ is a tautology
- Bob can certify that a formula is not a tautology
- Thus Tautology $\in$ Co-NP

**Def.** A tautology is a boolean formula that always evaluates to 1.

## Tautology Problem

**Input:** a boolean formula

**Output:** whether the formula is a tautology

- e.g. $(\neg x_1 \wedge x_2) \vee (\neg x_1 \wedge \neg x_3) \vee x_1 \vee (\neg x_2 \wedge x_3)$ is a tautology
- Bob can certify that a formula is not a tautology
- Thus Tautology $\in$ Co-NP
- Indeed, Tautology $= \overline{\text{Formula-Unsat}}$

# Prime

## Prime

**Input:** an integer $q \geq 2$

**Output:** whether $q$ is a prime

# Prime

## Prime

**Input:** an integer $q \geq 2$

**Output:** whether $q$ is a prime

- It is easy to certify that $q$ is not a prime

# Prime

**Prime**

**Input:** an integer $q \geq 2$

**Output:** whether $q$ is a prime

- It is easy to certify that $q$ is not a prime
- Prime $\in$ Co-NP

# Prime

**Prime**

**Input:** an integer $q \geq 2$

**Output:** whether $q$ is a prime

- It is easy to certify that $q$ is not a prime
- Prime $\in$ Co-NP
- [Pratt 1970] Prime $\in$ NP

# Prime

## Prime

**Input:** an integer $q \geq 2$

**Output:** whether $q$ is a prime

- It is easy to certify that $q$ is not a prime
- Prime $\in$ Co-NP
- [Pratt 1970] Prime $\in$ NP
- P $\subseteq$ NP $\cap$ Co-NP (see soon)

# Prime

## Prime

**Input:** an integer $q \geq 2$

**Output:** whether $q$ is a prime

- It is easy to certify that $q$ is not a prime
- Prime $\in$ Co-NP
- [Pratt 1970] Prime $\in$ NP
- P $\subseteq$ NP $\cap$ Co-NP (see soon)
- If a natural problem $X$ is in NP $\cap$ Co-NP, then it is likely that $X \in P$

# Prime

## Prime

**Input:** an integer $q \geq 2$

**Output:** whether $q$ is a prime

- It is easy to certify that $q$ is not a prime
- Prime $\in$ Co-NP
- [Pratt 1970] Prime $\in$ NP
- P $\subseteq$ NP $\cap$ Co-NP (see soon)
- If a natural problem $X$ is in NP $\cap$ Co-NP, then it is likely that $X \in P$
- [AKS 2002] Prime $\in$ P

# P $\subseteq$ NP

# P $\subseteq$ NP

- Let $X \in$ P and $s \in X$

**Q:** How can Alice convince Bob that $s$ is a yes instance?

# P ⊆ NP

- Let $X \in \mathsf{P}$ and $s \in X$

**Q:** How can Alice convince Bob that $s$ is a yes instance?

**A:** Since $X \in \mathsf{P}$, Bob can check whether $s \in X$ by himself, without Alice's help.

# $\mathsf{P} \subseteq \mathsf{NP}$

- Let $X \in \mathsf{P}$ and $s \in X$

**Q:** How can Alice convince Bob that $s$ is a yes instance?

**A:** Since $X \in \mathsf{P}$, Bob can check whether $s \in X$ by himself, without Alice's help.

- The certificate is an empty string

# P $\subseteq$ NP

- Let $X \in$ P and $s \in X$

**Q:** How can Alice convince Bob that $s$ is a yes instance?

**A:** Since $X \in$ P, Bob can check whether $s \in X$ by himself, without Alice's help.

- The certificate is an empty string
- Thus, $X \in$ NP and P $\subseteq$ NP

# P $\subseteq$ NP

- Let $X \in$ P and $s \in X$

**Q:** How can Alice convince Bob that $s$ is a yes instance?

**A:** Since $X \in$ P, Bob can check whether $s \in X$ by himself, without Alice's help.

- The certificate is an empty string
- Thus, $X \in$ NP and P $\subseteq$ NP
- Similarly, P $\subseteq$ Co-NP, thus P $\subseteq$ NP $\cap$ Co-NP

# Is $\mathsf{P} = \mathsf{NP}$?

## Is $\mathrm{P} = \mathrm{NP}$?

- A famous, big, and fundamental open problem in computer science

# Is P = NP?

- A famous, big, and fundamental open problem in computer science
- Little progress has been made

# Is P = NP?

- A famous, big, and fundamental open problem in computer science
- Little progress has been made
- Most researchers believe P $\neq$ NP

# Is P = NP?

- A famous, big, and fundamental open problem in computer science
- Little progress has been made
- Most researchers believe P $\neq$ NP
- It would be too amazing if P = NP: if one can check a solution efficiently, then one can find a solution efficiently

# Is P = NP?

- A famous, big, and fundamental open problem in computer science
- Little progress has been made
- Most researchers believe P $\neq$ NP
- It would be too amazing if P = NP: if one can check a solution efficiently, then one can find a solution efficiently
- Complexity assumption: P $\neq$ NP

# Is P = NP?

- A famous, big, and fundamental open problem in computer science
- Little progress has been made
- Most researchers believe P $\neq$ NP
- It would be too amazing if P = NP: if one can check a solution efficiently, then one can find a solution efficiently
- Complexity assumption: P $\neq$ NP
- We said it is unlikely that Hamiltonian Cycle can be solved in polynomial time:

# Is $\mathsf{P} = \mathsf{NP}$?

- A famous, big, and fundamental open problem in computer science
- Little progress has been made
- Most researchers believe $\mathsf{P} \neq \mathsf{NP}$
- It would be too amazing if $\mathsf{P} = \mathsf{NP}$: if one can check a solution efficiently, then one can find a solution efficiently

- Complexity assumption: $\mathsf{P} \neq \mathsf{NP}$
- We said it is unlikely that Hamiltonian Cycle can be solved in polynomial time:
  - if $\mathsf{P} \neq \mathsf{NP}$, then $\mathrm{HC} \notin \mathsf{P}$

# Is P = NP?

- A famous, big, and fundamental open problem in computer science
- Little progress has been made
- Most researchers believe P $\neq$ NP
- It would be too amazing if P = NP: if one can check a solution efficiently, then one can find a solution efficiently
- Complexity assumption: P $\neq$ NP
- We said it is unlikely that Hamiltonian Cycle can be solved in polynomial time:
  - if P $\neq$ NP, then HC $\notin$ P
  - HC $\notin$ P, unless P = NP

# Is NP $=$ Co-NP?

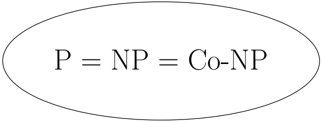
- Again, a big open problem

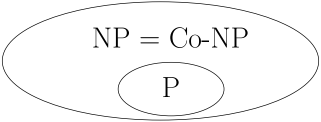
# Is NP $=$ Co-NP?

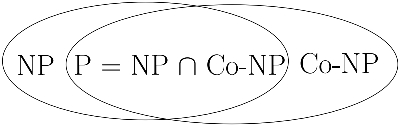
- Again, a big open problem
- Most researchers believe NP $\neq$ Co-NP.

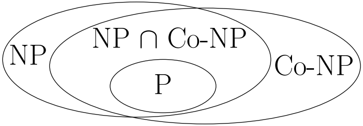
# 4 Possibilities of Relationships

Notice that $X \in$ NP $\iff \overline{X} \in$ Co-NP and P $\subseteq$ NP $\cap$ Co-NP




- People commonly believe: we are in the 4th scenario

# Outline

# Polynomial-Time Reducations

**Def.** Given a black box algorithm $A$ that solves a problem $X$, if any instance of a problem $Y$ can be solved using a polynomial number of standard computational steps, plus a polynomial number of calls to $A$, then we say $Y$ is polynomial-time reducible to $X$, denoted as $Y \leq_P X$.

# Polynomial-Time Reducations

**Def.** Given a black box algorithm $A$ that solves a problem $X$, if any instance of a problem $Y$ can be solved using a polynomial number of standard computational steps, plus a polynomial number of calls to $A$, then we say $Y$ is polynomial-time reducible to $X$, denoted as $Y \leq_P X$.

To prove positive results:

Suppose $Y \leq_P X$. If $X$ can be solved in polynomial time, then $Y$ can be solved in polynomial time.

# Polynomial-Time Reducations

**Def.** Given a black box algorithm $A$ that solves a problem $X$, if any instance of a problem $Y$ can be solved using a polynomial number of standard computational steps, plus a polynomial number of calls to $A$, then we say $Y$ is polynomial-time reducible to $X$, denoted as $Y \leq_P X$.

To prove positive results:

Suppose $Y \leq_P X$. If $X$ can be solved in polynomial time, then $Y$ can be solved in polynomial time.

To prove negative results:

Suppose $Y \leq_P X$. If $Y$ cannot be solved in polynomial time, then $X$ cannot be solved in polynomial time.

# Polynomial-Time Reduction: Example

## Hamiltonian-Path (HP) problem

**Input:** $G = (V, E)$ and $s, t \in V$

**Output:** whether there is a Hamiltonian path from $s$ to $t$ in $G$

# Polynomial-Time Reduction: Example

## Hamiltonian-Path (HP) problem

**Input:** $G = (V, E)$ and $s, t \in V$

**Output:** whether there is a Hamiltonian path from $s$ to $t$ in $G$

**Lemma** HP $\leq_{\mathsf{P}}$ HC.

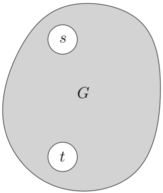
# Polynomial-Time Reduction: Example

## Hamiltonian-Path (HP) problem

**Input:** $G = (V, E)$ and $s, t \in V$

**Output:** whether there is a Hamiltonian path from $s$ to $t$ in $G$

**Lemma** HP $\leq_{\mathsf{P}}$ HC.



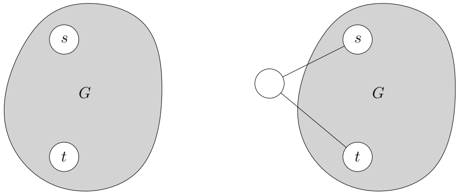
# Polynomial-Time Reduction: Example

## Hamiltonian-Path (HP) problem

**Input:** $G = (V, E)$ and $s, t \in V$

**Output:** whether there is a Hamiltonian path from $s$ to $t$ in $G$

**Lemma** HP $\leq_P$ HC.



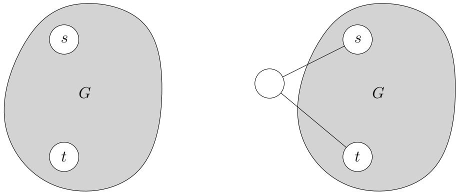
# Polynomial-Time Reduction: Example

## Hamiltonian-Path (HP) problem

**Input:** $G = (V, E)$ and $s, t \in V$

**Output:** whether there is a Hamiltonian path from $s$ to $t$ in $G$

**Lemma** HP $\leq_{\mathrm{P}}$ HC.




**Obs.** $G$ has a HP from $s$ to $t$ if and only if graph on right side has a HC.

# NP-Completeness

**Def.** A problem $X$ is called NP-complete if

1. $X \in \mathrm{NP}$, and
2. $Y \leq_{\mathrm{P}} X$ for every $Y \in \mathrm{NP}$.

# NP-Completeness

**Def.** A problem $X$ is called NP-hard if

2. $Y \leq_{\mathsf{P}} X$ for every $Y \in \mathsf{NP}$.

# NP-Completeness

**Def.** A problem $X$ is called NP-complete if

1. $X \in \mathsf{NP}$, and
2. $Y \leq_{\mathsf{P}} X$ for every $Y \in \mathsf{NP}$.

# NP-Completeness

**Def.** A problem $X$ is called NP-complete if

1. $X \in \mathsf{NP}$, and
2. $Y \leq_{\mathsf{P}} X$ for every $Y \in \mathsf{NP}$.

**Theorem** If $X$ is NP-complete and $X \in \mathsf{P}$, then $\mathsf{P} = \mathsf{NP}$.

# NP-Completeness

**Def.** A problem $X$ is called NP-complete if

1. $X \in \mathsf{NP}$, and
2. $Y \leq_{\mathsf{P}} X$ for every $Y \in \mathsf{NP}$.

**Theorem** If $X$ is NP-complete and $X \in \mathsf{P}$, then $\mathsf{P} = \mathsf{NP}$.

- NP-complete problems are the hardest problems in NP

# NP-Completeness

**Def.** A problem $X$ is called NP-complete if

1. $X \in \text{NP}$, and
2. $Y \leq_{\text{P}} X$ for every $Y \in \text{NP}$.

**Theorem** If $X$ is NP-complete and $X \in \text{P}$, then $\text{P} = \text{NP}$.

- NP-complete problems are the hardest problems in NP
- NP-hard problems are at least as hard as NP-complete problems (a NP-hard problem is not required to be in NP)

# NP-Completeness

**Def.** A problem $X$ is called NP-complete if

1. $X \in \text{NP}$, and
2. $Y \leq_{\text{P}} X$ for every $Y \in \text{NP}$.

**Theorem** If $X$ is NP-complete and $X \in \text{P}$, then $\text{P} = \text{NP}$.

- NP-complete problems are the hardest problems in NP
- NP-hard problems are at least as hard as NP-complete problems (a NP-hard problem is not required to be in NP)
- To prove $\text{P} = \text{NP}$ (if you believe it), you only need to give an efficient algorithm for any NP-complete problem

# NP-Completeness

**Def.** A problem $X$ is called NP-complete if

1. $X \in \text{NP}$, and
2. $Y \leq_{\text{P}} X$ for every $Y \in \text{NP}$.

**Theorem** If $X$ is NP-complete and $X \in \text{P}$, then $\text{P} = \text{NP}$.

- NP-complete problems are the hardest problems in NP
- NP-hard problems are at least as hard as NP-complete problems (a NP-hard problem is not required to be in NP)
- To prove $\text{P} = \text{NP}$ (if you believe it), you only need to give an efficient algorithm for any NP-complete problem
- If you believe $\text{P} \neq \text{NP}$, and proved that a problem $X$ is NP-complete (or NP-hard), stop trying to design efficient algorithms for $X$

# Outline

**Def.** A problem $X$ is called NP-complete if

1. $X \in \mathsf{NP}$, and
2. $Y \leq_{\mathsf{P}} X$ for every $Y \in \mathsf{NP}$.

**Def.** A problem $X$ is called NP-complete if

1. $X \in \text{NP}$, and
2. $Y \leq_\text{P} X$ for every $Y \in \text{NP}$.

- How can we find a problem $X \in \text{NP}$ such that every problem $Y \in \text{NP}$ is polynomial time reducible to $X$? Are we asking for too much?

**Def.** A problem $X$ is called NP-complete if

1. $X \in \text{NP}$, and
2. $Y \leq_\text{P} X$ for every $Y \in \text{NP}$.

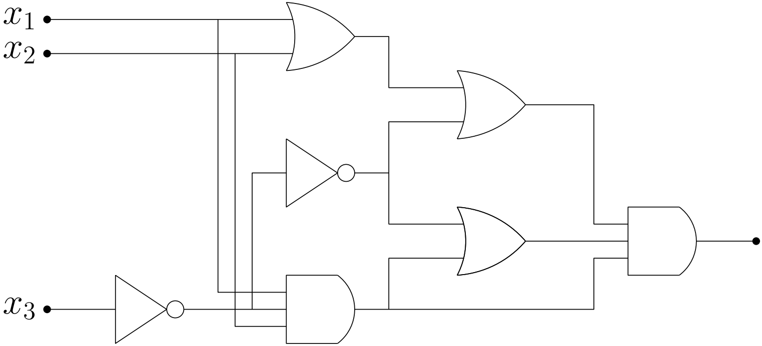
- How can we find a problem $X \in \text{NP}$ such that every problem $Y \in \text{NP}$ is polynomial time reducible to $X$? Are we asking for too much?
- No! There is indeed a large family of natural NP-complete problems

# The First NP-Complete Problem: Circuit-Sat

## Circuit Satisfiability (Circuit-Sat)

**Input:** a circuit

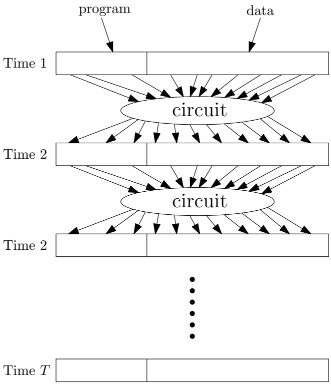
**Output:** whether the circuit is satisfiable



# Circuit-Sat is NP-Complete

- key fact: algorithms can be converted to circuits

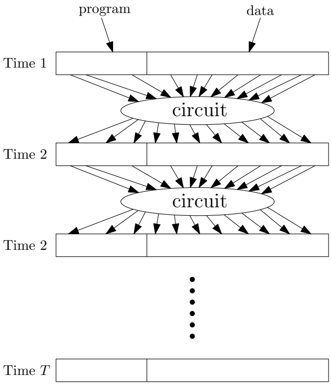
**Fact** Any algorithm that takes $n$ bits as input and outputs $0/1$ with running time $T(n)$ can be converted into a circuit of size $p(T(n))$ for some polynomial function $p(\cdot)$.



# Circuit-Sat is NP-Complete

- key fact: algorithms can be converted to circuits

**Fact** Any algorithm that takes $n$ bits as input and outputs $0/1$ with running time $T(n)$ can be converted into a circuit of size $p(T(n))$ for some polynomial function $p(\cdot)$.

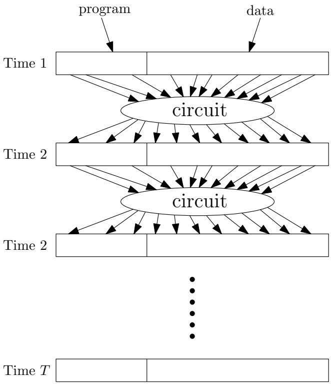



- Then, we can show that any problem $Y \in$ NP can be reduced to Circuit-Sat.

# Circuit-Sat is NP-Complete

- key fact: algorithms can be converted to circuits

**Fact** Any algorithm that takes $n$ bits as input and outputs $0/1$ with running time $T(n)$ can be converted into a circuit of size $p(T(n))$ for some polynomial function $p(\cdot)$.




- Then, we can show that any problem $Y \in$ NP can be reduced to Circuit-Sat.
- We prove HC $\leq_P$ Circuit-Sat as an example.

# HC $\leq_P$ Circuit-Sat

check-HC$(G, S)$

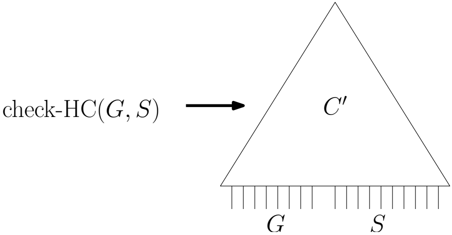
- Let check-HC$(G, S)$ be the certifier for the Hamiltonian cycle problem: check-HC$(G, S)$ returns 1 if $S$ is a Hamiltonian cycle is $G$ and $0$ otherwise.

# HC $\leq_P$ Circuit-Sat

check-HC$(G, S)$

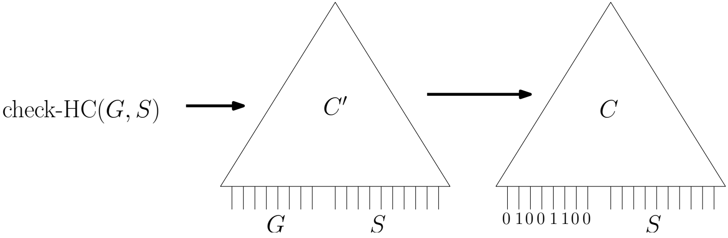
- Let check-HC$(G, S)$ be the certifier for the Hamiltonian cycle problem: check-HC$(G, S)$ returns 1 if $S$ is a Hamiltonian cycle is $G$ and $0$ otherwise.
- $G$ is a yes-instance if and only if there is an $S$ such that check-HC$(G, S)$ returns 1

# HC $\leq_P$ Circuit-Sat



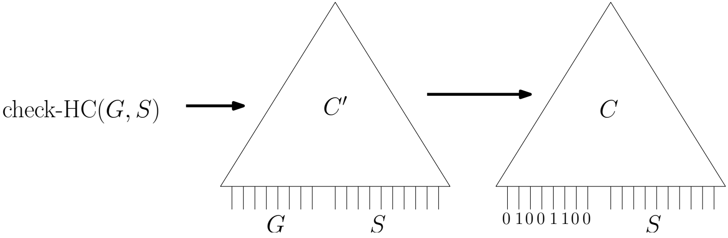

- Let check-HC$(G, S)$ be the certifier for the Hamiltonian cycle problem: check-HC$(G, S)$ returns 1 if $S$ is a Hamiltonian cycle is $G$ and $0$ otherwise.
- $G$ is a yes-instance if and only if there is an $S$ such that check-HC$(G, S)$ returns 1
- Construct a circuit $C'$ for the algorithm check-HC

# HC $\leq_P$ Circuit-Sat




- Let check-HC$(G, S)$ be the certifier for the Hamiltonian cycle problem: check-HC$(G, S)$ returns 1 if $S$ is a Hamiltonian cycle is $G$ and $0$ otherwise.
- $G$ is a yes-instance if and only if there is an $S$ such that check-HC$(G, S)$ returns 1
- Construct a circuit $C'$ for the algorithm check-HC
- hard-wire the instance $G$ to the circuit $C'$ to obtain the circuit $C$

# HC $\leq_P$ Circuit-Sat




- Let check-HC$(G, S)$ be the certifier for the Hamiltonian cycle problem: check-HC$(G, S)$ returns 1 if $S$ is a Hamiltonian cycle is $G$ and $0$ otherwise.
- $G$ is a yes-instance if and only if there is an $S$ such that check-HC$(G, S)$ returns 1
- Construct a circuit $C'$ for the algorithm check-HC
- hard-wire the instance $G$ to the circuit $C'$ to obtain the circuit $C$
- $G$ is a yes-instance if and only if $C$ is satisfiable

# $Y \leq_P$ Circuit-Sat, For Every $Y \in$NP

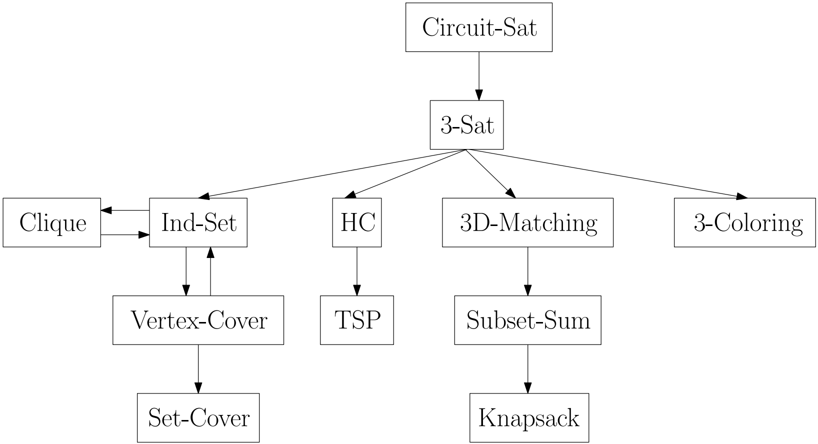
- Let check-Y$(s,t)$ be the certifier for problem $Y$: check-Y$(s,t)$ returns 1 if $t$ is a valid certificate for $s$.
- $s$ is a yes-instance if and only if there is a $t$ such that check-Y$(s,t)$ returns 1
- Construct a circuit $C'$ for the algorithm check-Y
- hard-wire the instance $s$ to the circuit $C'$ to obtain the circuit $C$
- $s$ is a yes-instance if and only if $C$ is satisfiable $\square$

# $Y \leq_P$ Circuit-Sat, For Every $Y \in$NP

- Let check-Y$(s,t)$ be the certifier for problem $Y$: check-Y$(s,t)$ returns 1 if $t$ is a valid certificate for $s$.
- $s$ is a yes-instance if and only if there is a $t$ such that check-Y$(s,t)$ returns 1
- Construct a circuit $C'$ for the algorithm check-Y
- hard-wire the instance $s$ to the circuit $C'$ to obtain the circuit $C$
- $s$ is a yes-instance if and only if $C$ is satisfiable $\square$

**Theorem** Circuit-Sat is NP-complete.

# Reductions of NP-Complete Problems



# Outline

# Summary

- We consider decision problems
- Inputs are encoded as $\{0,1\}$-strings

**Def.** The complexity class P is the set of decision problems $X$ that can be solved in polynomial time.

- Alice has a supercomputer, fast enough to run an exponential time algorithm
- Bob has a slow computer, which can only run a polynomial-time algorithm

**Def.** (Informal) The complexity class NP is the set of problems for which Alice can convince Bob a yes instance is a yes instance

# Summary

**Def.** $B$ is an efficient certifier for a problem $X$ if

- $B$ is a polynomial-time algorithm that takes two input strings $s$ and $t$
- there is a polynomial function $p$ such that, $s \in X$ if and only if there is string $t$ such that $|t| \leq p(|s|)$ and $B(s,t) = 1$.

The string $t$ such that $B(s,t) = 1$ is called a certificate.

**Def.** The complexity class NP is the set of all problems for which there exists an efficient certifier.

# Summary

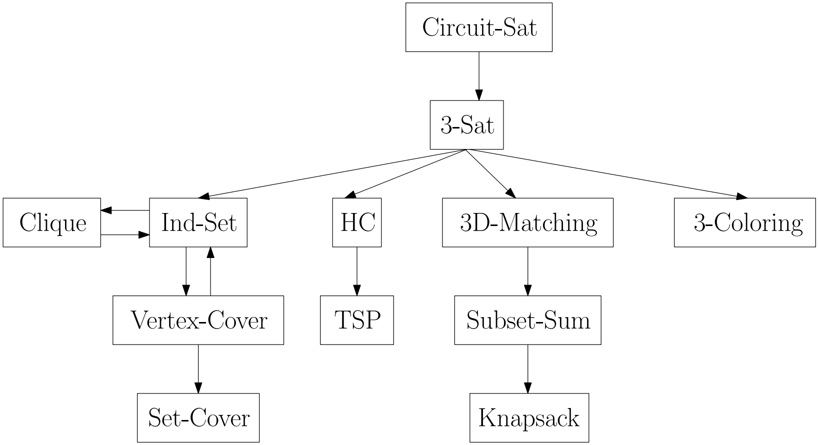
**Def.** Given a black box algorithm $A$ that solves a problem $X$, if any instance of a problem $Y$ can be solved using a polynomial number of standard computational steps, plus a polynomial number of calls to $A$, then we say $Y$ is polynomial-time reducible to $X$, denoted as $Y \leq_P X$.

**Def.** A problem $X$ is called NP-complete if

1. $X \in$ NP, and
2. $Y \leq_\text{P} X$ for every $Y \in$ NP.

- If any NP-complete problem can be solved in polynomial time, then $P = NP$
- Unless $P = NP$, a NP-complete problem can not be solved in polynomial time

# Summary



Circuit-Sat
3-Sat
Clique
Ind-Set
HC
3D-Matching
3-Coloring
Vertex-Cover
TSP
Subset-Sum
Set-Cover
Knapsack

# Summary

## Proof of NP-Completeness for Circuit-Sat

- Fact 1: a polynomial-time algorithm can be converted to a polynomial-size circuit
- Fact 2: for a problem in NP, there is a efficient certifier.
- Given a problem $X \in$ NP, let $B(s,t)$ be the certifier
- Convert $B(s,t)$ to a circuit and hard-wire $s$ to the input gates
- $s$ is a yes-instance if and only if the resulting circuit is satisfiable

- Proof of NP-Completeness for other problems by reductions