CSE 431/531: Algorithm Analysis and Design (Fall 2021)

NP-Completeness

Lecturer: Shi Li

Department of Computer Science and Engineering
University at Buffalo
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 cannot be solved efficiently.

Q: Why do we study negative results?
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 cannot be solved efficiently.

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

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

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

**A:**

- 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: $O(n), O(n^2), O(n^{2.5} \log n), O(n^{100})$
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 $\Omega(2^{nc})$ 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 $\Omega(2^{nc})$ for some $c$
- Do not need to worry about the computational model
Outline

1. Some Hard Problems
2. P, NP and Co-NP
3. Polynomial Time Reductions and NP-Completeness
4. NP-Complete Problems
5. Summary
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
**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.
Hamiltonian Cycle (HC) Problem

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

**Output:** whether $G$ contains a Hamiltonian cycle
**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
Hamiltonian Cycle 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: \( O(n!m) = 2^{O(n \lg n)} \)

Far away from polynomial time

HC is NP-hard: it is unlikely that it can be solved in polynomial time.
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.
**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
Def. An independent set of $G = (V, E)$ is a subset $I \subseteq V$ such that no two vertices in $I$ are adjacent in $G$. 
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

Input: boolean formula with $n$ variables, with $\lor$, $\land$, $\neg$ operators.

Output: whether the boolean formula is satisfiable

- Example: $\neg((\neg x_1 \land x_2) \lor (\neg x_1 \land \neg x_3) \lor x_1 \lor (\neg x_2 \land x_3))$ is not satisfiable

- Trivial algorithm: enumerate all possible assignments, and check if each assignment satisfies the formula
Formula Satisfiability

**Input:** boolean formula with $n$ variables, with $\lor$, $\land$, $\neg$ operators.

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

- Example: $\neg((\neg x_1 \land x_2) \lor (\neg x_1 \land \neg x_3) \lor x_1 \lor (\neg x_2 \land x_3))$ is not satisfiable

- Trivial algorithm: enumerate all possible assignments, and check if each assignment satisfies the formula

- Formula Satisfiability is NP-hard
Outline

1 Some Hard Problems

2 P, NP and Co-NP

3 Polynomial Time Reductions and NP-Completeness

4 NP-Complete Problems

5 Summary
Def. A problem $X$ is called a **decision problem** if the output is either 0 or 1 (yes/no).
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.
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 $k$
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
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)
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:
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
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: 1111011110001111100011000001
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: 111101111100001111000011000001
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
  11111111000001
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: 1111011111100011111110000111000001
  11000011011111111000001
The input of a problem will be encoded as a binary string.
The input of an problem will be encoded as a binary string.

Example: Interval Scheduling Problem
The input of a problem will be encoded as a binary string.

Example: Interval Scheduling Problem

\[(0, 3, 0, 4, 2, 4, 3, 5, 4, 6, 4, 7, 5, 8, 7, 9, 8, 9)\]
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
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.
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 $s \in X$. 
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.
**Def.** The *complexity class P* is the set of decision problems $X$ that can be solved in polynomial time.
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 \( P \).
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.

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 $G$. The message Alice sends to Bob is called a certificate, and the algorithm Bob runs is called a certifier.
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?
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.
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 $G$. 

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

- 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

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

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

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

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

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

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

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

1. $B$ is a polynomial-time algorithm that takes two input strings $s$ and $t$
2. there is a polynomial function $p$ such that, $s \in X$ if and only if there is a 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 $G$

Clearly, $B$ runs in polynomial time $G \in HC \iff \exists S, B(G, S) = 1$
Hamiltonian Cycle $\in$ NP

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

Clearly, $B$ runs in polynomial time.

$G \in \text{HC} \iff \exists S, B(G, S) = 1$
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 $p$

Clearly, $B$ runs in polynomial time $G \in \text{HC} \iff \exists S, B(G, S) = 1$
Hamiltonian Cycle $\in \text{NP}$

- **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 $G$
Hamiltonian Cycle ∈ 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

$G \in \text{HC} \iff \exists S, B(G, S) = 1$
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 $f : V \rightarrow V$

Clearly, $B$ runs in polynomial time if and only if $G_1 \in GI \iff \exists f, B((G_1, G_2), f) = 1$.
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 $f : V \rightarrow V$
- $|\text{encoding}(f)| \leq p(|\text{encoding}(G_1, G_2)|)$ 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 $(u, v) \in E_1 \iff (f(u), f(v)) \in E_2$.

Clearly, $B$ runs in polynomial time ($G_1 \in \text{GI} \iff \exists f, B((G_1, G_2), f) = 1$)
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 $p$
- **Certifier $B$:** $B((G_1, G_2), f) = 1$ if and only if for every $u, v \in V$, we have $(u, v) \in E_1 \iff (f(u), f(v)) \in E_2$. 

Clearly, $B$ runs in polynomial time ($G_1, G_2) \in GI \iff \exists f, B((G_1, G_2), f) = 1$
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 $f : V \rightarrow V$
- $|\text{encoding}(f)| \leq p(|\text{encoding}(G_1, G_2)|)$ 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 $(u, v) \in E_1 \iff (f(u), f(v)) \in E_2$.
- 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 $f : V \to V$
- $|\text{encoding}(f)| \leq p(|\text{encoding}(G_1, G_2)|)$ 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 $(u, v) \in E_1 \iff (f(u), f(v)) \in E_2$.
- Clearly, $B$ runs in polynomial time

$(G_1, G_2) \in \text{GI} \iff \exists f, B((G_1, G_2), f) = 1$
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 $\in \text{NP}$

- 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 $p$
Maximum Independent Set $\in \text{NP}$

- **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 $\in\text{NP}$

- **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

$(G, k) \in \text{MIS} \iff \exists S, B((G, k), S) = 1$
Circuit Satisfiability (Circuit-Sat) Problem

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

**Output:** whether there is an assignment such that the output is 1?
Circuit Satisfiability (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?

Input: graph $G = (V, E)$
Output: whether $G$ does not contain a Hamiltonian cycle
Input: graph $G = (V, E)$

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

Is $\overline{HC} \in NP$?
**Input:** graph $G = (V, E)$

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

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

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

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

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

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

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

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

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

- Is $\overline{HC} \in NP$?
- 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
- $\overline{HC} \in \text{Co-NP}$
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 $\overline{X} \in \text{NP}$.
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. \((\neg x_1 \land x_2) \lor (\neg x_1 \land \neg x_3) \lor x_1 \lor (\neg x_2 \land x_3)\) 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. \((\neg x_1 \land x_2) \lor (\neg x_1 \land \neg x_3) \lor x_1 \lor (\neg x_2 \land x_3)\) 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. \((\neg x_1 \land x_2) \lor (\neg x_1 \land \neg x_3) \lor x_1 \lor (\neg x_2 \land x_3)\) 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. \((\neg x_1 \land x_2) \lor (\neg x_1 \land \neg x_3) \lor x_1 \lor (\neg x_2 \land x_3)\) is a tautology
- Bob can certify that a formula is not a tautology
- Thus Tautology \(\in\) Co-NP
- Indeed, Tautology = Formula-Unsat
Problem:

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

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

| Input: | an integer $q \geq 2$ |
| Output: | whether $q$ is a prime |

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

[Pratt 1970] $\text{Prime} \in \text{NP}$

$\text{P} \subseteq \text{NP} \cap \text{Co-NP}$ (see soon)

[AKS 2002] $\text{Prime} \in \text{P}$
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 $\in$ Co-NP

$P \subseteq NP \cap \text{Co-NP}$ (see soon)

If a natural problem $X$ is in $NP \cap \text{Co-NP}$, then it is likely that $X \in P$

Prime $\in P$
**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 \( \in \text{Co-NP} \)
- [Pratt 1970] Prime \( \in \text{NP} \)
- \( \text{P} \subseteq \text{NP} \cap \text{Co-NP} \) (see soon)
Prime

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

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

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

$P \subseteq NP$
Let $X \in P$ and $s \in X$

Q: How can Alice convince Bob that $s$ is a yes instance?
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.
Let $X \in \mathcal{P}$ and $s \in X$

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

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

- The certificate is an empty string
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 $P \subseteq NP$
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 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, unless P = NP.
Is $P = NP$?

- 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 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 $HC \not\in P$
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 $HC \not\in P$
  - $HC \not\in P$, unless $P = NP$
Is NP = Co-NP?

- Again, a big open problem
Is \( NP = \text{Co-NP} \)?

- Again, a big open problem
- Most researchers believe \( NP \neq \text{Co-NP} \).
Notice that $X \in \text{NP} \iff \overline{X} \in \text{Co-NP}$ and $P \subseteq \text{NP} \cap \text{Co-NP}$.

- **P = NP = Co-NP**
- **NP = Co-NP**

NP: $P = \text{NP} \cap \text{Co-NP}$

People commonly believe: we are in the 4th scenario.
Outline

1. Some Hard Problems
2. P, NP and Co-NP
3. Polynomial Time Reductions and NP-Completeness
4. NP-Complete Problems
5. 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 $Y \leq_P X$. 

**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.
**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.
**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 $G$
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** $\text{HP} \leq_P \text{HC}$. 

Polynomial-Time Reduction: Example
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** $\text{HP} \leq_{P} \text{HC}$. 
**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** \( \text{HP} \leq_P \text{HC} \).
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** $HP \leq_P HC$.

**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 \textbf{NP-complete} if

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

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

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

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 $P = \text{NP}$ (if you believe it), you only need to give an efficient algorithm for any NP-complete problem.

If you believe $P \neq \text{NP}$, and proved that a problem $X$ is NP-complete (or NP-hard), stop trying to design efficient algorithms for $X$. 
**Def.** A problem $X$ is called **NP-complete** if

1. $X \in \text{NP}$, and
2. $Y \leq_{P} X$ for every $Y \in \text{NP}$.
Def. A problem $X$ is called \textbf{NP-complete} if

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

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

1. $X \in \text{NP}$, and
2. $Y \leq_P X$ 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-Completeness

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

1. $X \in \text{NP}$, and
2. $Y \leq_P X$ 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. $Y \leq_P X$ 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 $X$
Def. A problem $X$ is called **NP-complete** if

1. $X \in \text{NP}$, and
2. $Y \leq_p X$ for every $Y \in \text{NP}$.
Def. A problem $X$ is called \textbf{NP-complete} if

1. $X \in \text{NP}$, and
2. $Y \leq_{P} X$ 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. \( Y \leq_P X \) 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 $Y \in \text{NP}$ 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 $Y \in \text{NP}$ can be reduced to Circuit-Sat.
- We prove $\text{HC} \leq_P \text{Circuit-Sat}$ as an example.
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.
Let $\text{check-HC}(G, S)$ be the certifier for the Hamiltonian cycle problem: $\text{check-HC}(G, S)$ returns 1 if $S$ is a Hamiltonian cycle in $G$ and 0 otherwise.

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

\( G \) is a yes-instance if and only if there is an \( S \) such that \( \text{check-HC}(G, S) \) returns 1.

Construct a circuit \( C' \) for the algorithm \( \text{check-HC} \).
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 in \(G\) 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 \(C\).
Let $\text{check-HC}(G, S)$ be the certifier for the Hamiltonian cycle problem: $\text{check-HC}(G, S)$ returns 1 if $S$ is a Hamiltonian cycle in $G$ and 0 otherwise.

$G$ is a yes-instance if and only if there is an $S$ such that $\text{check-HC}(G, S)$ returns 1.

Construct a circuit $C'$ for the algorithm $\text{check-HC}$.

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.
$Y \leq_P \text{Circuit-Sat, For Every } Y \in \text{NP}$

- 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
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.

\[ Y \leq_P \text{Circuit-Sat, For Every } Y \in \text{NP} \]

**Theorem**  Circuit-Sat is NP-complete.
Reductions of NP-Complete Problems

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

Diagram:

- Circuit-Sat → 3-Sat
- 3-Sat → Ind-Set, HC, 3D-Matching, Set-Cover, Subset-Sum, TSP, Knapsack
- Ind-Set → Vertex-Cover
- HC → TSP
- 3D-Matching → Subset-Sum
- Set-Cover → Knapsack
Outline

1. Some Hard Problems
2. P, NP and Co-NP
3. Polynomial Time Reductions and NP-Completeness
4. NP-Complete Problems
5. 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 \( \text{NP} \) is the set of problems for which Alice can convince Bob a yes instance is a yes instance.
### Summary

<table>
<thead>
<tr>
<th>Def.</th>
<th>$B$ is an <strong>efficient certifier</strong> for a problem $X$ if</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>$B$ is a polynomial-time algorithm that takes two input strings $s$ and $t$</td>
</tr>
<tr>
<td></td>
<td>there is a polynomial function $p$ such that, $s \in X$ if and only if there is string $t$ such that $</td>
</tr>
<tr>
<td></td>
<td>The string $t$ such that $B(s, t) = 1$ is called a <strong>certificate</strong>.</td>
</tr>
</tbody>
</table>

| 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. $X \in \text{NP}$, and
2. $Y \leq_P X$ for every $Y \in \text{NP}$.

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

- 3D-Matching
- Circuit-Sat
- 3-Sat
- Ind-Set
- Vertex-Cover
- HC
- Set-Cover
- 3-Coloring
- TSP
- Subset-Sum
- 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 $X \in \text{NP}$, 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