Part I

Introduction to Reductions
Reductions

A reduction from Problem X to Problem Y means (informally) that if we have an algorithm for Problem Y, we can use it to find an algorithm for Problem X.
A reduction from Problem X to Problem Y means (informally) that if we have an algorithm for Problem Y, we can use it to find an algorithm for Problem X.

Using Reductions

- We use reductions to find algorithms to solve problems.
A reduction from Problem X to Problem Y means (informally) that if we have an algorithm for Problem Y, we can use it to find an algorithm for Problem X.

Using Reductions

- We use reductions to find algorithms to solve problems.
- We also use reductions to show that we can’t find algorithms for some problems. (We say that these problems are hard.)
A reduction from Problem X to Problem Y means (informally) that if we have an algorithm for Problem Y, we can use it to find an algorithm for Problem X.

**Using Reductions**

- We use reductions to find algorithms to solve problems.
- We also use reductions to show that we can’t find algorithms for some problems. (We say that these problems are **hard**.)

Also, the right reductions might win you a million dollars!
Example 1: Bipartite Matching and Flows

How do we solve the Bipartite Matching Problem?

Given a bipartite graph $G = (U \cup V, E)$ and number $k$, does $G$ have a matching of size $\geq k$?

Solution

Reduce it to Max-Flow. $G$ has a matching of size $\geq k$ iff there is a flow from $s$ to $t$ of value $\geq k$. 
Example 1: Bipartite Matching and Flows

How do we solve the Bipartite Matching Problem?

Given a bipartite graph $G = (U \cup V, E)$ and number $k$, does $G$ have a matching of size $\geq k$?
Example 1: Bipartite Matching and Flows

How do we solve the Bipartite Matching Problem?
Given a bipartite graph $G = (U \cup V, E)$ and number $k$, does $G$ have a matching of size $\geq k$?

Solution
Reduce it to Max-Flow. $G$ has a matching of size $\geq k$ iff there is a flow from $s$ to $t$ of value $\geq k$. 
Example 1: Bipartite Matching and Flows

How do we solve the **Bipartite Matching** Problem?

Given a bipartite graph $G = (U \cup V, E)$ and number $k$, does $G$ have a matching of size $\geq k$?

Solution

Reduce it to **Max-Flow**. $G$ has a matching of size $\geq k$ iff there is a flow from $s$ to $t$ of value $\geq k$. 
Example 1: Bipartite Matching and Flows

How do we solve the Bipartite Matching Problem?

Given a bipartite graph $G = (U \cup V, E)$ and number $k$, does $G$ have a matching of size $\geq k$?

Solution

Reduce it to Max-Flow. $G$ has a matching of size $\geq k$ iff there is a flow from $s$ to $t$ of value $\geq k$. 
Overview
Definitions

Types of Problems

Decision, Search, and Optimization

- Decision problems (example: given \( n \), is \( n \) prime?)

While using reductions and comparing problems, we typically work with the decision versions. Decision problems have Yes/No answers. This makes them easy to work with.
### Types of Problems

<table>
<thead>
<tr>
<th>Decision, Search, and Optimization</th>
</tr>
</thead>
<tbody>
<tr>
<td>• Decision problems (example: given ( n ), is ( n ) prime?)</td>
</tr>
<tr>
<td>• Search problems (example: given ( n ), find a factor of ( n ) if it exists)</td>
</tr>
</tbody>
</table>
Types of Problems

Decision, Search, and Optimization

- Decision problems (example: given $n$, is $n$ prime?)
- Search problems (example: given $n$, find a factor of $n$ if it exists)
- Optimization problems (example: find the smallest prime factor of $n$.)
Types of Problems

Decision, Search, and Optimization

- Decision problems (example: given $n$, is $n$ prime?)
- Search problems (example: given $n$, find a factor of $n$ if it exists)
- Optimization problems (example: find the smallest prime factor of $n$.)

For **Max-Flow**, the Optimization version is: Find the Maximum flow between $s$ and $t$. The Decision Version is: Given an integer $k$, is there a flow of value $\geq k$ between $s$ and $t$?
Types of Problems

Decision, Search, and Optimization

- Decision problems (example: given \( n \), is \( n \) prime?)
- Search problems (example: given \( n \), find a factor of \( n \) if it exists)
- Optimization problems (example: find the smallest prime factor of \( n \).)

For \text{Max-Flow}, the Optimization version is: Find the Maximum flow between \( s \) and \( t \). The Decision Version is: Given an integer \( k \), is there a flow of value \( \geq k \) between \( s \) and \( t \)?

While using reductions and comparing problems, we typically work with the decision versions. Decision problems have Yes/No answers. This makes them easy to work with.
Problems vs Instances

- A problem $\Pi$ consists of an infinite collection of inputs $\{I_1, I_2, \ldots, \}$. Each input is referred to as an instance.
- The size of an instance $I$ is the number of bits in its representation.
- For an instance $I$, $\text{sol}(I)$ is a set of feasible solutions to $I$.
- For optimization problems each solution $s \in \text{sol}(I)$ has an associated value.
An instance of **Bipartite Matching** is a bipartite graph, and an integer $k$.
Examples

An instance of **Bipartite Matching** is a bipartite graph, and an integer \( k \). The solution to this instance is “YES” if the graph has a matching of size \( \geq k \), and “NO” otherwise.
An instance of **Bipartite Matching** is a bipartite graph, and an integer $k$. The solution to this instance is “YES” if the graph has a matching of size $\geq k$, and “NO” otherwise.

An instance of **Max-Flow** is a graph $G$ with edge-capacities, two vertices $s, t$, and an integer $k$. The solution to this instance is “YES” if there is a flow from $s$ to $t$ of value $\geq k$, else ‘NO’.
An instance of Bipartite Matching is a bipartite graph, and an integer $k$. The solution to this instance is “YES” if the graph has a matching of size $\geq k$, and “NO” otherwise.

An instance of Max-Flow is a graph $G$ with edge-capacities, two vertices $s, t$, and an integer $k$. The solution to this instance is “YES” if there is a flow from $s$ to $t$ of value $\geq k$, else ‘NO’.

What is an Algorithm for a decision Problem $X$? It takes as input an instance of $X$, and outputs either “YES” or “NO”.
Decision Problems and Languages

- A finite alphabet $\Sigma$. $\Sigma^*$ is set of all finite strings on $\Sigma$.
- A language $L$ is simply a subset of $\Sigma^*$; a set of strings.
Decision Problems and Languages

- A finite alphabet $\Sigma$. $\Sigma^*$ is set of all finite strings on $\Sigma$.
- A language $L$ is simply a subset of $\Sigma^*$; a set of strings.

For every language $L$ there is an associated decision problem $\Pi_L$ and conversely, for every decision problem $\Pi$ there is an associated language $L_\Pi$. 
Decision Problems and Languages

- A finite alphabet $\Sigma$. $\Sigma^*$ is set of all finite strings on $\Sigma$.
- A language $L$ is simply a subset of $\Sigma^*$; a set of strings.

For every language $L$ there is an associated decision problem $\Pi_L$ and conversely, for every decision problem $\Pi$ there is an associated language $L_\Pi$.

- Given $L$, $\Pi_L$ is the following problem: given $x \in \Sigma^*$, is $x \in L$?
  Each string in $\Sigma^*$ is an instance of $\Pi_L$ and $L$ is the set of instances for which the answer is YES.
- Given $\Pi$ the associated language
  $$L_\Pi = \{ l | l \text{ is an instance of } \Pi \text{ for which answer is YES} \}.$$  

Thus, decision problems and languages are used interchangeably.
For decision problems $X, Y$, a reduction from $X$ to $Y$ is:

- An algorithm . . .
For decision problems $X$, $Y$, a reduction from $X$ to $Y$ is:

- An algorithm . . .
- that takes $I_X$, an instance of $X$ as input . . .
Reductions, revised.

For decision problems $X, Y$, a reduction from $X$ to $Y$ is:

- An algorithm . . .
- that takes $I_X$, an instance of $X$ as input . . .
- and returns $I_Y$, an instance of $Y$ as output . . .
Reductions, revised.

For decision problems $X$, $Y$, a reduction from $X$ to $Y$ is:

- An algorithm . . .
- that takes $I_X$, an instance of $X$ as input . . .
- and returns $I_Y$, an instance of $Y$ as output . . .
- such that the solution (YES/NO) to $I_Y$ is the same as the solution to $I_X$. 

(Actually, this is only one type of reduction, but this is the one we'll use most often.)
Reductions, revised.

For decision problems $X, Y$, a reduction from $X$ to $Y$ is:

- An algorithm . . .
- that takes $I_X$, an instance of $X$ as input . . .
- and returns $I_Y$, an instance of $Y$ as output . . .
- such that the solution (YES/NO) to $I_Y$ is the same as the solution to $I_X$.

(Actually, this is only one type of reduction, but this is the one we’ll use most often.)
Using reductions to solve problems

Given a reduction $R$ from $X$ to $Y$, and an algorithm $A_Y$ for $Y$: 

1. Given an instance $I_X$ of $X$, use $R$ to produce an instance $I_Y$ of $Y$.
2. Use $A_Y$ to solve $I_Y$, and output the answer of $A_Y$.

In particular, if $R$ and $A_Y$ are polynomial-time algorithms, $A_X$ is also polynomial-time.
Using reductions to solve problems

Given a reduction $\mathcal{R}$ from $X$ to $Y$, and an algorithm $A_Y$ for $Y$: We have an algorithm $A_X$ for $X$! Here it is:
Using reductions to solve problems

Given a reduction $\mathcal{R}$ from $X$ to $Y$, and an algorithm $A_Y$ for $Y$: We have an algorithm $A_X$ for $X$! Here it is:

Given an instance $I_X$ of $X$, use $\mathcal{R}$ to produce an instance $I_Y$ of $Y$. Now, use $A_Y$ to solve $I_Y$, and output the answer of $A_Y$. 
Using reductions to solve problems

Given a reduction $\mathcal{R}$ from $X$ to $Y$, and an algorithm $A_Y$ for $Y$: We have an algorithm $A_X$ for $X$! Here it is:

Given an instance $I_X$ of $X$, use $\mathcal{R}$ to produce an instance $I_Y$ of $Y$. Now, use $A_Y$ to solve $I_Y$, and output the answer of $A_Y$.

\[ I_X \xrightarrow{\mathcal{R}} I_Y \xrightarrow{A_Y} \begin{cases} YES \\ NO \end{cases} \]

In particular, if $\mathcal{R}$ and $A_Y$ are polynomial-time algorithms, $A_X$ is also polynomial-time.
Using reductions to solve problems

Given a reduction $\mathcal{R}$ from $X$ to $Y$, and an algorithm $A_Y$ for $Y$: We have an algorithm $A_X$ for $X$! Here it is:

Given an instance $I_X$ of $X$, use $\mathcal{R}$ to produce an instance $I_Y$ of $Y$. Now, use $A_Y$ to solve $I_Y$, and output the answer of $A_Y$.

In particular, if $\mathcal{R}$ and $A_Y$ are polynomial-time algorithms, $A_X$ is also polynomial-time.
Comparing Problems

- Reductions allow us to formalize the notion of “Problem $X$ is no harder to solve than Problem $Y$.”
Comparing Problems

- Reductions allow us to formalize the notion of “Problem $X$ is no harder to solve than Problem $Y$”.
- If Problem $X$ reduces to Problem $Y$ (we write $X \leq Y$), then $X$ cannot be harder to solve than $Y$. 

Bipartite Matching $\leq$ Max-Flow. Therefore, Bipartite Matching cannot be harder than Max-Flow. Equivalently, Max-Flow is at least as hard as Bipartite Matching.
Comparing Problems

- Reductions allow us to formalize the notion of “Problem $X$ is no harder to solve than Problem $Y$”.
- If Problem $X$ reduces to Problem $Y$ (we write $X \leq Y$), then $X$ cannot be harder to solve than $Y$.
- **Bipartite Matching $\leq$ Max-Flow.** Therefore, **Bipartite Matching cannot be harder than Max-Flow.**
Comparing Problems

- Reductions allow us to formalize the notion of “Problem $X$ is no harder to solve than Problem $Y$”.
- If Problem $X$ reduces to Problem $Y$ (we write $X \leq Y$), then $X$ cannot be harder to solve than $Y$.
- **Bipartite Matching $\leq$ Max-Flow.** Therefore, **Bipartite Matching cannot be harder than Max-Flow.**
- Equivalently, **Max-Flow is at least as hard as Bipartite Matching.**
Comparing Problems

- Reductions allow us to formalize the notion of “Problem $X$ is no harder to solve than Problem $Y$”.
- If Problem $X$ reduces to Problem $Y$ (we write $X \leq Y$), then $X$ cannot be harder to solve than $Y$.
- **Bipartite Matching $\leq$ Max-Flow.** Therefore, **Bipartite Matching cannot be harder than Max-Flow.**
- Equivalently, **Max-Flow is at least as hard as Bipartite Matching.**
- More generally, if $X \leq Y$, we can say that $X$ is no harder than $Y$, or $Y$ is at least as hard as $X$. 
Part II

Examples of Reductions
Independent Sets and Cliques

Given a graph $G$, a set of vertices $V'$ is:
Independent Sets and Cliques

Given a graph $G$, a set of vertices $V'$ is:

- An **independent set** if no two vertices of $V'$ are connected by an edge of $G$. 
Independent Sets and Cliques

Given a graph $G$, a set of vertices $V'$ is:

- An **independent set** if no two vertices of $V'$ are connected by an edge of $G$.
- A **clique** if every pair of vertices in $V'$ is connected by an edge of $G$. 
Independent Sets and Cliques

Given a graph $G$, a set of vertices $V'$ is:

- An **independent set** if no two vertices of $V'$ are connected by an edge of $G$.
- A **clique** if every pair of vertices in $V'$ is connected by an edge of $G$. 

![Graph diagram]
Independent Sets and Clique

Given a graph $G$, a set of vertices $V'$ is:

- An **independent set** if no two vertices of $V'$ are connected by an edge of $G$.

- A **clique** if *every* pair of vertices in $V'$ is connected by an edge of $G$. 

![Graph Diagram]
Independent Sets and Clique

Given a graph $G$, a set of vertices $V'$ is:

- An **independent set** if no two vertices of $V'$ are connected by an edge of $G$.
- A **clique** if every pair of vertices in $V'$ is connected by an edge of $G$. 
The **INDEPENDENT SET** and **CLIQUE** Problems

**The INDEPENDENT SET Problem:**

**Input** A graph $G$ and an integer $k$.

**Goal** Decide whether $G$ has an independent set of size $\geq k$. 
The **Independent Set** and **Clique** Problems

**The Independent Set Problem:**

**Input** A graph $G$ and an integer $k$.

**Goal** Decide whether $G$ has an independent set of size $\geq k$.

**The Clique Problem:**

**Input** A graph $G$ and an integer $k$.

**Goal** Decide whether $G$ has a clique of size $\geq k$. 
Recall

For decision problems $X$, $Y$, a reduction from $X$ to $Y$ is:

- An algorithm . . .
Recall

For decision problems $X$, $Y$, a reduction from $X$ to $Y$ is:

- An algorithm …
- that takes $I_X$, an instance of $X$ as input …
Recall

For decision problems $X$, $Y$, a reduction from $X$ to $Y$ is:

- An algorithm 
- that takes $I_X$, an instance of $X$ as input 
- and returns $I_Y$, an instance of $Y$ as output 

Recall

For decision problems $X$, $Y$, a reduction from $X$ to $Y$ is:

- An algorithm . . .
- that takes $I_X$, an instance of $X$ as input . . .
- and returns $I_Y$, an instance of $Y$ as output . . .
- such that the solution (YES/NO) to $I_Y$ is the same as the solution to $I_X$. 
Reducing **Independent Set** to **Clique**

An instance of **Independent Set** is a graph $G$ and an integer $k$. Convert $G$ to $G'$, in which $\langle u, v \rangle$ is an edge iff $\langle u, v \rangle$ is not an edge of $G$. ($G'$ is the complement of $G$). We use $G'$ and $k$ as the instance of **Clique**.
Reducing **INDEPENDENT SET** to **CLIQUE**

An instance of **INDEPENDENT SET** is a graph $G$ and an integer $k$. 

A graphical representation is shown, illustrating the conversion of a graph to its complement for the purpose of reducing the Independent Set problem to the Clique problem.
Reducing **INDEPENDENT SET** to **CLIQUE**

An instance of **INDEPENDENT SET** is a graph $G$ and an integer $k$.

Convert $G$ to $\overline{G}$, in which $(u, v)$ is an edge iff $(u, v)$ is **not** an edge of $G$. ($\overline{G}$ is the *complement* of $G$.)

We use $\overline{G}$ and $k$ as the instance of **CLIQUE**.
Reducing **Independent Set** to **Clique**

An instance of **Independent Set** is a graph $G$ and an integer $k$.

Convert $G$ to $\overline{G}$, in which $(u, v)$ is an edge iff $(u, v)$ is not an edge of $G$. ($\overline{G}$ is the *complement* of $G$.)

We use $\overline{G}$ and $k$ as the instance of **Clique**.
Reducing \textbf{INDEPENDENT SET} to \textbf{CLIQUE}

An instance of \textbf{INDEPENDENT SET} is a graph $G$ and an integer $k$.

Convert $G$ to $\overline{G}$, in which $(u, v)$ is an edge iff $(u, v)$ is \textbf{not} an edge of $G$. ($\overline{G}$ is the \textit{complement} of $G$.)

We use $\overline{G}$ and $k$ as the instance of \textbf{CLIQUE}.
Reducing **Independent Set** to **Clique**

An instance of **Independent Set** is a graph $G$ and an integer $k$.

Convert $G$ to $\overline{G}$, in which $(u, v)$ is an edge iff $(u, v)$ is **not** an edge of $G$. ($\overline{G}$ is the *complement* of $G$.) We use $\overline{G}$ and $k$ as the instance of **Clique**.
We showed that **Independent Set \( \leq \) Clique**.
What does this mean?
We showed that \textbf{Independent Set} \leq \textbf{Clique}.
What does this mean?

If we have an algorithm for \textbf{Clique}, we have an algorithm for \textbf{Independent Set}. 
We showed that $\text{Independent Set} \leq \text{Clique}$. What does this mean?

If we have an algorithm for $\text{Clique}$, we have an algorithm for $\text{Independent Set}$.

The $\text{Clique}$ Problem is at least as hard as the $\text{Independent Set}$ problem.
DFAs and NFAs

DFAs (Remember 273?) are automata that accept regular languages. NFAs are the same, except that they are non-deterministic, while DFAs are deterministic.
DFAs (Remember 273?) are automata that accept regular languages. NFAs are the same, except that they are non-deterministic, while DFAs are deterministic.

Every NFA can be converted to a DFA that accepts the same language using the subset construction.

(How long does this take?)
DFAs and NFAs

DFAs (Remember 273?) are automata that accept regular languages. NFAs are the same, except that they are non-deterministic, while DFAs are deterministic.

Every NFA can be converted to a DFA that accepts the same language using the subset construction.

(How long does this take?) The smallest DFA equivalent to an NFA with \( n \) states may have \( \approx 2^n \) states.
A DFA $M$ is said to be **universal** if it accepts every string. That is, $L(M) = \Sigma^*$, the set of all strings.
A DFA $M$ is said to be universal if it accepts every string. That is, $L(M) = \Sigma^*$, the set of all strings.

The DFA Universality Problem:

**Input** A DFA $M$

**Goal** Decide whether $M$ is universal.
A DFA $M$ is said to be universal if it accepts every string. That is, $L(M) = \Sigma^*$, the set of all strings.

**The DFA Universality Problem:**

**Input** A DFA $M$

**Goal** Decide whether $M$ is universal.

How do we solve DFA Universality?
DFA Universality

A DFA $M$ is said to be universal if it accepts every string. That is, $L(M) = \Sigma^*$, the set of all strings.

The DFA Universality Problem:

**Input** A DFA $M$

**Goal** Decide whether $M$ is universal.

How do we solve DFA Universality? We check if $M$ has any reachable non-final state. Alternatively, minimize $M$ to obtain $M'$ and see if $M'$ has a single state which is an accepting state.
An NFA $N$ is said to be **universal** if it accepts every string. That is, $L(N) = \Sigma^*$, the set of all strings.

**The NFA Universality Problem:**

**Input** An NFA $N$

**Goal** Decide whether $N$ is universal.

How do we solve **NFA Universality**?
An NFA $N$ is said to be **universal** if it accepts every string. That is, $L(N) = \Sigma^*$, the set of all strings.

**The NFA Universality Problem:**

**Input** An NFA $N$

**Goal** Decide whether $N$ is universal.

How do we solve **NFA Universality**?
Reduce it to **DFA Universality**?
An NFA $N$ is said to be \textit{universal} if it accepts every string. That is, $L(N) = \Sigma^*$, the set of all strings.

The \textbf{NFA Universality} Problem:

\textbf{Input}  An NFA $N$

\textbf{Goal}  Decide whether $N$ is universal.

How do we solve \textbf{NFA Universality}?
Reduce it to \textbf{DFA Universality}?
Given an NFA $N$, convert it to an equivalent DFA $M$, and use the \textbf{DFA Universality} Algorithm.
An NFA $N$ is said to be universal if it accepts every string. That is, $L(N) = \Sigma^*$, the set of all strings.

**The NFA Universality Problem:**

**Input** An NFA $N$

**Goal** Decide whether $N$ is universal.

How do we solve NFA Universality?
Reduce it to DFA Universality?
Given an NFA $N$, convert it to an equivalent DFA $M$, and use the DFA Universality Algorithm.

The reduction takes exponential time!
Polynomial-time reductions

We say that an algorithm is efficient if it runs in polynomial-time.
Polynomial-time reductions

We say that an algorithm is efficient if it runs in polynomial-time.

To find efficient algorithms for problems, we are only interested in polynomial-time reductions. Reductions that take longer are not useful.
Polynomial-time reductions

We say that an algorithm is efficient if it runs in polynomial-time.

To find efficient algorithms for problems, we are only interested in polynomial-time reductions. Reductions that take longer are not useful.

If we have a polynomial-time reduction from problem $X$ to problem $Y$ (we write $X \leq_P Y$), and a poly-time algorithm $A_Y$ for $Y$, we have a polynomial-time/efficient algorithm for $X$. 
Polynomial-time reductions

We say that an algorithm is **efficient** if it runs in polynomial-time.

To find efficient algorithms for problems, we are only interested in **polynomial-time** reductions. Reductions that take longer are not useful.

If we have a polynomial-time reduction from problem $X$ to problem $Y$ (we write $X \leq_P Y$), and a poly-time algorithm $A_Y$ for $Y$, we have a polynomial-time/efficient algorithm for $X$.
Polynomial-time Reduction

A polynomial time reduction from a decision problem $X$ to a decision problem $Y$ is an algorithm $A$ that has the following properties:

- given an instance $I_X$ of $X$, $A$ produces an instance $I_Y$ of $Y$
- $A$ runs in time polynomial in $|I_X|$.
- Answer to $I_X$ YES iff answer to $I_Y$ is YES.

**Proposition**

If $X \leq_P Y$ then a polynomial time algorithm for $Y$ implies a polynomial time algorithm for $X$.

Such a reduction is called a Karp reduction. Most reductions we will need are Karp reductions.
For decision problems $X$ and $Y$, if $X \leq_p Y$, and $Y$ has an efficient algorithm, $X$ has an efficient algorithm.
Polynomial-time reductions and hardness

For decision problems $X$ and $Y$, if $X \leq_P Y$, and $Y$ has an efficient algorithm, $X$ has an efficient algorithm.

If you believe that \textsc{Independent Set} does not have an efficient algorithm, why should you believe the same of \textsc{Clique}?
Polynomial-time reductions and hardness

For decision problems $X$ and $Y$, if $X \leq_P Y$, and $Y$ has an efficient algorithm, $X$ has an efficient algorithm.

If you believe that INDEPENDENT SET does not have an efficient algorithm, why should you believe the same of CLIQUE?

Because we showed INDEPENDENT SET $\leq_P$ CLIQUE. If CLIQUE had an efficient algorithm, so would INDEPENDENT SET!
Polynomial-time reductions and hardness

For decision problems $X$ and $Y$, if $X \leq_P Y$, and $Y$ has an efficient algorithm, $X$ has an efficient algorithm.

If you believe that $\text{Independent Set}$ does not have an efficient algorithm, why should you believe the same of $\text{Clique}$?

Because we showed $\text{Independent Set} \leq_P \text{Clique}$. If $\text{Clique}$ had an efficient algorithm, so would $\text{Independent Set}$!

If $X \leq_P Y$ and $X$ does not have an efficient algorithm, $Y$ cannot have an efficient algorithm!
Polynomial-time reductions and instance sizes

**Proposition**

Let \( \mathcal{R} \) be a polynomial-time reduction from \( X \) to \( Y \). Then for any instance \( I_X \) of \( X \), the size of the instance \( I_Y \) of \( Y \) produced from \( I_X \) by \( \mathcal{R} \) is polynomial in the size of \( I_X \).
Polynomial-time reductions and instance sizes

Proposition

Let \( \mathcal{R} \) be a polynomial-time reduction from \( X \) to \( Y \). Then for any instance \( I_X \) of \( X \), the size of the instance \( I_Y \) of \( Y \) produced from \( I_X \) by \( \mathcal{R} \) is polynomial in the size of \( I_X \).

Proof.

\( \mathcal{R} \) is a polynomial-time algorithm and hence on input \( I_X \) of size \( |I_X| \) it runs in time \( p(|I_X|) \) for some polynomial \( p() \). 
\( I_Y \) is the output of \( \mathcal{R} \) on input \( I_X \) 
\( \mathcal{R} \) can write at most \( p(|I_X|) \) bits and hence \( |I_Y| \leq p(|I_X|) \). \( \square \)
Proposition

Let \( \mathcal{R} \) be a polynomial-time reduction from \( X \) to \( Y \). Then for any instance \( I_X \) of \( X \), the size of the instance \( I_Y \) of \( Y \) produced from \( I_X \) by \( \mathcal{R} \) is polynomial in the size of \( I_X \).

Proof.

\( \mathcal{R} \) is a polynomial-time algorithm and hence on input \( I_X \) of size \( |I_X| \) it runs in time \( p(|I_X|) \) for some polynomial \( p() \). 

\( I_Y \) is the output of \( \mathcal{R} \) on input \( I_X \).

\( \mathcal{R} \) can write at most \( p(|I_X|) \) bits and hence \( |I_Y| \leq p(|I_X|) \).

Note: Converse is not true. A reduction need not be polynomial-time even if output of reduction is of size polynomial in its input.
Polynomial-time Reduction

A polynomial time reduction from a decision problem $X$ to a decision problem $Y$ is an algorithm $A$ that has the following properties:

- given an instance $I_X$ of $X$, $A$ produces an instance $I_Y$ of $Y$
- $A$ runs in time polynomial in $|I_X|$. This implies that $|I_Y|$ (size of $I_Y$) is polynomial in $|I_X|$.
- Answer to $I_X$ YES iff answer to $I_Y$ is YES.

**Proposition**

If $X \leq_P Y$ then a polynomial time algorithm for $Y$ implies a polynomial time algorithm for $X$.

Such a reduction is called a Karp reduction. Most reductions we will need are Karp reductions.
Transitivity of Reductions

Proposition

\( X \leq_P Y \) and \( Y \leq_P Z \) implies that \( X \leq_P Z \).

Note: \( X \leq_P Y \) does not imply that \( Y \leq_P X \) and hence it is very important to know the FROM and TO in a reduction.

To prove \( X \leq_P Y \) you need to show a reduction FROM \( X \) TO \( Y \) In other words show that an algorithm for \( Y \) implies an algorithm for \( X \).
Vertex Cover

Given a graph $G = (V, E)$, a set of vertices $S$ is:
Vertex Cover

Given a graph $G = (V, E)$, a set of vertices $S$ is:

- A vertex cover if every $e \in E$ has at least one endpoint in $S$. 
Given a graph $G = (V, E)$, a set of vertices $S$ is:

- A **vertex cover** if every $e \in E$ has at least one endpoint in $S$. 
Given a graph $G = (V, E)$, a set of vertices $S$ is:

1. A **vertex cover** if every $e \in E$ has at least one endpoint in $S$. 
Given a graph $G = (V, E)$, a set of vertices $S$ is:

- A **vertex cover** if every $e \in E$ has at least one endpoint in $S$. 

![Graph with vertex cover highlighted]
The **Vertex Cover** Problem:

**Input**  A graph $G$ and integer $k$

**Goal**  Decide whether there is a vertex cover of size $\leq k$
The Vertex Cover Problem

The Vertex Cover Problem:

Input  A graph $G$ and integer $k$

Goal  Decide whether there is a vertex cover of size $\leq k$

Can we relate Independent Set and Vertex Cover?
Relationship between Vertex Cover and Independent Set

Proposition

Let $G = (V, E)$ be a graph. $S$ is an independent set if and only if $V \setminus S$ is a vertex cover.

Proof.

$(\Rightarrow)$ Let $S$ be an independent set. Consider any edge $(u, v) \in E$. Since $S$ is an independent set, either $u \not\in S$ or $v \not\in S$. Thus, either $u \in V \setminus S$ or $v \in V \setminus S$. $V \setminus S$ is a vertex cover.

$(\Leftarrow)$ Let $V \setminus S$ be some vertex cover. Consider $u, v \in S$. $(u, v)$ is not an edge, as otherwise $V \setminus S$ does not cover $(u, v)$. $S$ is thus an independent set.
Proposition

Let \( G = (V, E) \) be a graph. \( S \) is an independent set if and only if \( V \setminus S \) is a vertex cover.

Proof.

(⇒) Let \( S \) be an independent set. Consider any edge \((u, v)\) ∈ \( E \). Since \( S \) is an independent set, either \( u \notin S \) or \( v \notin S \). Thus, either \( u \in V \setminus S \) or \( v \in V \setminus S \). Hence, \( V \setminus S \) is a vertex cover.

(⇐) Let \( V \setminus S \) be some vertex cover. Consider \((u, v)\) ∈ \( S \). Since \((u, v)\) is not an edge, as otherwise \( V \setminus S \) does not cover \((u, v)\). Thus, \( S \) is an independent set.
Proposition

Let $G = (V, E)$ be a graph. $S$ is an independent set if and only if $V \setminus S$ is a vertex cover.

Proof.

$(\Rightarrow)$ Let $S$ be an independent set.
Relationship between Vertex Cover and Independent Set

**Proposition**

Let $G = (V, E)$ be a graph. $S$ is an independent set if and only if $V \setminus S$ is a vertex cover.

**Proof.**

$(\Rightarrow)$ Let $S$ be an independent set

- Consider any edge $(u, v) \in E$
Proposition

Let $G = (V, E)$ be a graph. $S$ is an independent set if and only if $V \setminus S$ is a vertex cover.

Proof.

$(\Rightarrow)$ Let $S$ be an independent set.
- Consider any edge $(u, v) \in E$.
- Since $S$ is an independent set, either $u \notin S$ or $v \notin S$. 
Proposition

Let \( G = (V, E) \) be a graph. \( S \) is an independent set if and only if \( V \setminus S \) is a vertex cover.

Proof.

(\( \Rightarrow \) ) Let \( S \) be an independent set

- Consider any edge \( (u, v) \in E \)
- Since \( S \) is an independent set, either \( u \notin S \) or \( v \notin S \)
- Thus, either \( u \in V \setminus S \) or \( v \in V \setminus S \)
Relationship between Vertex Cover and Independent Set

Proposition

Let $G = (V, E)$ be a graph. $S$ is an independent set if and only if $V \setminus S$ is a vertex cover.

Proof.

$(\Rightarrow)$ Let $S$ be an independent set

- Consider any edge $(u, v) \in E$
- Since $S$ is an independent set, either $u \notin S$ or $v \notin S$
- Thus, either $u \in V \setminus S$ or $v \in V \setminus S$
- $V \setminus S$ is a vertex cover

$(\Leftarrow)$ Let $V \setminus S$ be some vertex cover

Consider $u, v \in S$ $(u, v)$ is not edge, as otherwise $V \setminus S$ does not cover $(u, v)$

$S$ is thus an independent set
Relationship between Vertex Cover and Independent Set

Proposition

Let $G = (V, E)$ be a graph. $S$ is an independent set if and only if $V \setminus S$ is a vertex cover.

Proof.

(⇒) Let $S$ be an independent set
- Consider any edge $(u, v) \in E$
- Since $S$ is an independent set, either $u \not\in S$ or $v \not\in S$
- Thus, either $u \in V \setminus S$ or $v \in V \setminus S$
- $V \setminus S$ is a vertex cover

(⇐) Let $V \setminus S$ be some vertex cover
Proposition

Let $G = (V, E)$ be a graph. $S$ is an independent set if and only if $V \setminus S$ is a vertex cover

Proof.

($\Rightarrow$) Let $S$ be an independent set
- Consider any edge $(u, v) \in E$
- Since $S$ is an independent set, either $u \notin S$ or $v \notin S$
- Thus, either $u \in V \setminus S$ or $v \in V \setminus S$
- $V \setminus S$ is a vertex cover

($\Leftarrow$) Let $V \setminus S$ be some vertex cover
- Consider $u, v \in S$
Proposition

Let $G = (V, E)$ be a graph. $S$ is an independent set if and only if $V \setminus S$ is a vertex cover.

Proof.

($\Rightarrow$) Let $S$ be an independent set.
- Consider any edge $(u, v) \in E$.
- Since $S$ is an independent set, either $u \notin S$ or $v \notin S$.
- Thus, either $u \in V \setminus S$ or $v \in V \setminus S$.
- $V \setminus S$ is a vertex cover.

($\Leftarrow$) Let $V \setminus S$ be some vertex cover.
- Consider $u, v \in S$.
- $(u, v)$ is not edge, as otherwise $V \setminus S$ does not cover $(u, v)$. 
Relationship between Vertex Cover and Independent Set

**Proposition**

Let $G = (V, E)$ be a graph. $S$ is an independent set if and only if $V \setminus S$ is a vertex cover.

**Proof.**

($\Rightarrow$) Let $S$ be an independent set
- Consider any edge $(u, v) \in E$
- Since $S$ is an independent set, either $u \notin S$ or $v \notin S$
- Thus, either $u \in V \setminus S$ or $v \in V \setminus S$
- $V \setminus S$ is a vertex cover

($\Leftarrow$) Let $V \setminus S$ be some vertex cover
- Consider $u, v \in S$
- $(u, v)$ is not edge, as otherwise $V \setminus S$ does not cover $(u, v)$
- $S$ is thus an independent set
**Independent Set \leq_p Vertex Cover**

Let $G$, a graph with $n$ vertices, and an integer $k$ be an instance of the **Independent Set** problem.
Independent Set $\leq_p$ Vertex Cover

Let $G$, a graph with $n$ vertices, and an integer $k$ be an instance of the Independent Set problem.

$G$ has an independent set of size $\geq k$ iff $G$ has a vertex cover of size $\leq n - k$
Let $G$, a graph with $n$ vertices, and an integer $k$ be an instance of the Independent Set problem.

$G$ has an independent set of size $\geq k$ iff $G$ has a vertex cover of size $\leq n - k$

$(G, k)$ is an instance of Independent Set, and $(G, n - k)$ is an instance of Vertex Cover with the same answer.
**Independent Set \leq_p Vertex Cover**

Let $G$, a graph with $n$ vertices, and an integer $k$ be an instance of the **Independent Set** problem.

$G$ has an independent set of size $\geq k$ iff $G$ has a vertex cover of size $\leq n - k$

$(G, k)$ is an instance of **Independent Set**, and $(G, n - k)$ is an instance of **Vertex Cover** with the same answer.

Therefore, **Independent Set \leq_p Vertex Cover**. Also **Vertex Cover \leq_p Independent Set**.
A problem of Languages

Suppose you work for the United Nations. Let $U$ be the set of all languages spoken by people across the world. The United Nations also has a set of translators, all of whom speak English, and some other languages from $U$. 

A problem of Languages

Suppose you work for the United Nations. Let $U$ be the set of all languages spoken by people across the world. The United Nations also has a set of translators, all of whom speak English, and some other languages from $U$.

Due to budget cuts, you can only afford to keep $k$ translators on your payroll. Can you do this, while still ensuring that there is someone who speaks every language in $U$?
A problem of Languages

Suppose you work for the United Nations. Let $U$ be the set of all languages spoken by people across the world. The United Nations also has a set of translators, all of whom speak English, and some other languages from $U$.

Due to budget cuts, you can only afford to keep $k$ translators on your payroll. Can you do this, while still ensuring that there is someone who speaks every language in $U$?

More General problem: Find/Hire a small group of people who can accomplish a large number of tasks.
The **Set Cover** Problem

**Input**
Given a set $U$ of $n$ elements, a collection $S_1, S_2, \ldots, S_m$ of subsets of $U$, and an integer $k$

**Goal**
Is there is a collection of at most $k$ of these sets $S_i$ whose union is equal to $U$?
The **Set Cover** Problem

**Input** Given a set $U$ of $n$ elements, a collection $S_1, S_2, \ldots S_m$ of subsets of $U$, and an integer $k$

**Goal** Is there is a collection of at most $k$ of these sets $S_i$ whose union is equal to $U$?

**Example**

Let $U = \{1, 2, 3, 4, 5, 6, 7\}$, $k = 2$ with

- $S_1 = \{3, 7\}$
- $S_2 = \{3, 4, 5\}$
- $S_3 = \{1\}$
- $S_4 = \{2, 4\}$
- $S_5 = \{5\}$
- $S_6 = \{1, 2, 6, 7\}$
The **Set Cover** Problem

**Input**  Given a set $U$ of $n$ elements, a collection $S_1, S_2, \ldots, S_m$ of subsets of $U$, and an integer $k$

**Goal**  Is there is a collection of at most $k$ of these sets $S_i$ whose union is equal to $U$?

**Example**

Let $U = \{1, 2, 3, 4, 5, 6, 7\}$, $k = 2$ with

- $S_1 = \{3, 7\}$
- $S_2 = \{3, 4, 5\}$
- $S_3 = \{1\}$
- $S_4 = \{2, 4\}$
- $S_5 = \{5\}$
- $S_6 = \{1, 2, 6, 7\}$

$\{S_2, S_6\}$ is a set cover
**Vertex Cover \( \leq_p \) Set Cover**

Given graph \( G = (V, E) \) and integer \( k \) as instance of **Vertex Cover**, construct an instance of **Set Cover** as follows:
**Vertex Cover \( \leq_p \) Set Cover**

Given graph \( G = (V, E) \) and integer \( k \) as instance of **Vertex Cover**, construct an instance of **Set Cover** as follows:

- Number \( k \) for the **Set Cover** instance is the same as the number \( k \) given for the **Vertex Cover** instance.
**Vertex Cover \( \leq_P \) Set Cover**

Given graph \( G = (V, E) \) and integer \( k \) as instance of **Vertex Cover**, construct an instance of **Set Cover** as follows:

- Number \( k \) for the **Set Cover** instance is the same as the number \( k \) given for the **Vertex Cover** instance.
- \( U = E \)
**Vertex Cover \( \leq_p \) Set Cover**

Given graph \( G = (V, E) \) and integer \( k \) as instance of **Vertex Cover**, construct an instance of **Set Cover** as follows:

- Number \( k \) for the **Set Cover** instance is the same as the number \( k \) given for the **Vertex Cover** instance.
- \( U = E \)
- We will have one set corresponding to each vertex; \( S_v = \{ e \mid e \text{ is incident on } v \} \)
**Vertex Cover $\leq_P$ Set Cover**

Given graph $G = (V, E)$ and integer $k$ as instance of **Vertex Cover**, construct an instance of **Set Cover** as follows:

- Number $k$ for the **Set Cover** instance is the same as the number $k$ given for the **Vertex Cover** instance.
- $U = E$
- We will have one set corresponding to each vertex; $S_v = \{e \mid e \text{ is incident on } v\}$

Observe that $G$ has vertex cover of size $k$ if and only if $U, \{S_v\}_{v \in V}$ has a set cover of size $k$. (Exercise: Prove this.)
**Vertex Cover \( \leq_P \) Set Cover: Example**

\[
\begin{align*}
\text{Let } U &= \{ a, b, c, d, e, f, g \}, \quad k = 2 \\
S_1 &= \{ c, g \}, \\
S_2 &= \{ b, d \}, \\
S_3 &= \{ c, d, e \}, \\
S_4 &= \{ e, f \}, \\
S_5 &= \{ a \}, \\
S_6 &= \{ a, b, f, g \}.
\end{align*}
\]

\{S_3, S_6\} is a set cover.
**Vertex Cover \( \leq_p \text{Set Cover} \): Example**

Let \( U = \{a, b, c, d, e, f, g\} \),
\( k = 2 \) with

\[
S_1 = \{c, g\} \quad S_2 = \{b, d\} \\
S_3 = \{c, d, e\} \quad S_4 = \{e, f\} \\
S_5 = \{a\} \quad S_6 = \{a, b, f, g\}
\]
**Vertex Cover \( \leq_P \) Set Cover: Example**

Let \( U = \{a, b, c, d, e, f, g\} \), 
\( k = 2 \) with

\[
\begin{align*}
S_1 &= \{c, g\} & S_2 &= \{b, d\} \\
S_3 &= \{c, d, e\} & S_4 &= \{e, f\} \\
S_5 &= \{a\} & S_6 &= \{a, b, f, g\}
\end{align*}
\]

\( \{S_3, S_6\} \) is a set cover

\( \{3, 6\} \) is a vertex cover
To prove that $X \leq_P Y$ you need to give an algorithm $A$ that
- transforms an instance $I_X$ of $X$ into an instance $I_Y$ of $Y$
- satisfies the property that answer to $I_X$ is YES iff $I_Y$ is YES
  - typical easy direction to prove: answer to $I_Y$ is YES if answer to $I_X$ is YES
  - typical difficult direction to prove: answer to $I_X$ is YES if answer to $I_Y$ is YES (equivalently answer to $I_X$ is NO if answer to $I_Y$ is NO)
- runs in polynomial time
Try proving $\text{MATCHING} \leq_P \text{BIPARTITE MATCHING}$ via following reduction:

- Given graph $G = (V, E)$ obtain a bipartite graph $G' = (V', E')$ as follows.
  
  - Let $V_1 = \{u_1 \mid u \in V\}$ and $V_2 = \{u_2 \mid u \in V\}$. We set $V' = V_1 \cup V_2$ (that is, we make two copies of $V$).
  
  - $E' = \{(u_1, v_2) \mid u \neq v \text{ and } (u, v) \in E\}$

- Given $G$ and integer $k$ the reduction outputs $G'$ and $k$. 
Example
Claim

Reduction is a poly-time algorithm. If $G$ has a matching of size $k$ then $G'$ has a matching of size $k$. 

Incorrect! Why?

Vertex $u \in V$ has two copies $u_1$ and $u_2$ in $G'$. A matching in $G'$ may use both copies!
Claim

Reduction is a poly-time algorithm. If $G$ has a matching of size $k$ then $G'$ has a matching of size $k$.

Proof.

Exercise.
**Claim**

*Reduction is a poly-time algorithm. If G has a matching of size k then G' has a matching of size k.*

**Proof.**

Exercise.

**Claim**

*If G' has a matching of size k then G has a matching of size k.*
“Proof”

Claim

*Reduction is a poly-time algorithm. If $G$ has a matching of size $k$ then $G'$ has a matching of size $k$.*

Proof.

Exercise.

Claim

*If $G'$ has a matching of size $k$ then $G$ has a matching of size $k$.*

Incorrect! Why?
“Proof”

Claim

*Reduction is a poly-time algorithm. If $G$ has a matching of size $k$ then $G'$ has a matching of size $k$.**

Proof.
Exercise.

Claim

*If $G'$ has a matching of size $k$ then $G$ has a matching of size $k$.**

Incorrect! Why? Vertex $u \in V$ has two copies $u_1$ and $u_2$ in $G'$. A matching in $G'$ may use both copies!
Summary

We looked at polynomial-time reductions.
Summary

We looked at polynomial-time reductions.

Using polynomial-time reductions
Summary

We looked at polynomial-time reductions.

<table>
<thead>
<tr>
<th>Using polynomial-time reductions</th>
</tr>
</thead>
<tbody>
<tr>
<td>If $X \leq_P Y$, and we have an efficient algorithm for $Y$, we have an efficient algorithm for $X$.</td>
</tr>
</tbody>
</table>
Summary

We looked at polynomial-time reductions.

Using polynomial-time reductions

- If $X \leq_P Y$, and we have an efficient algorithm for $Y$, we have an efficient algorithm for $X$.
- If $X \leq_P Y$, and there is no efficient algorithm for $X$, there is no efficient algorithm for $Y$. 
Summary

We looked at polynomial-time reductions.

### Using polynomial-time reductions

- If $X \leq_P Y$, and we have an efficient algorithm for $Y$, we have an efficient algorithm for $X$.
- If $X \leq_P Y$, and there is no efficient algorithm for $X$, there is no efficient algorithm for $Y$.

We looked at some examples of reductions between **Independent Set**, **Clique**, **Vertex Cover**, and **Set Cover**.