Intended Learning Outcomes

• Outline “The Halting Problem”
• Explain complexity class $P$ and $NP$ of decision problems
• Explain $NP$-Hard and $NP$-completeness
• Relate decision problems via polynomial-time “reduction” algorithm
• Analyze typical $NP$-complete problems
• Sketch proofs of $NP$-complete problems
A Formal Language Framework

• An **alphabet** $\Sigma$ is a finite set of symbols

• A **language** $L$ over $\Sigma$ is any set of strings made up of symbols from $\Sigma$
  - E.g. if $\Sigma = \{0,1\}$, the set $L = \{10, 11, 101, 111, 1011, 1101, 10001, \ldots\}$ (binary representation of prime numbers)

• The **empty string** is denoted by $\varepsilon$

• The **empty language** is denoted by $\emptyset$

• The **language of all strings** $\Sigma$ by $\Sigma^*$
  - E.g. if $\Sigma = \{0,1\}$ then $\Sigma^* = \{\varepsilon, 0, 1, 00, 01, 10, 11, 000, \ldots\}$ (set of all binary strings)
Operations on a Formal Language

• We can perform a variety of operations on languages
  - **Union** and **intersection**
  - The **complement** of \( L \) by \( L^+ = \Sigma - L \)
  - The **concatenation** \( L_1 L_2 \) of two languages \( L_1 \) and \( L_2 \) is the language \( L = \{x_1 x_2 : x_1 \in L_1 \text{ and } x_2 \in L_2\} \)
  - The set of instances for any **decision problem** \( Q \) is simply the set \( \Sigma^* \), where \( \Sigma = \{0, 1\} \)

![Diagram](attachment:image.png)

\( L(M) = \{0, 1\}^* \) that have an even number of 1’s
Decision Problem Using a Formal Language

- Since \( Q \) is entirely characterized by those \textbf{problem instances that produce a 1 (yes) answer}, we can view \( Q \) as a language \( L \) over \( \Sigma = \{0, 1\} \) where
  - \( L = \{ x \in \Sigma^* : Q(x) = 1 \} \quad \Sigma^* \) represents the language of all strings

- Example: the \textbf{decision problem PATH} has the corresponding language
  - \( \text{PATH} = \{ <G, u, v, k> : G = (V, E) \text{ is an undirected graph,} \)
    - \( u, v \in V, \)
    - \( k \geq 0 \text{ is an integer, and there exists a path from } u \text{ to } v \text{ in } G \text{ consisting of at most } k \text{ edges} \} \)
NP-Completeness

Definition 2: A language $L_1$ is polynomial-time reducible to a language $L_2 (L_1 \leq_p L_2)$, if there exists a polynomial-time computable function $f: \{0,1\}^* \rightarrow \{0,1\}^*$ such that for all $x \in \{0,1\}^*$, $x \in L_1$ iff $f(x) \in L_2$

- A language $L \subseteq \{0,1\}^*$ is NP-complete if:
  1. $L \in NP$  
     If a certificate can be verified in polynomial-time
  2. $L' \leq_p L$ for every $L' \in NP$  
     Every problem in NP is reducible to $L$ in polynomial-time

- If $L$ satisfies property 2, but not necessarily property 1, we say that $L$ is NP-hard
Circuit Satisfiability

• A Boolean formula contains
  ▪ **Variables** whose values are 0 or 1
  ▪ **Connectives**: $\land$ (AND), $\lor$ (OR), and $\neg$ (NOT)

<table>
<thead>
<tr>
<th>$x$</th>
<th>$\neg x$</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>$x$</th>
<th>$y$</th>
<th>$x \land y$</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>$x$</th>
<th>$y$</th>
<th>$x \lor y$</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

• A Boolean formula is **SAT** if there exists some assignment to its variables that evaluates it to 1
Circuit Satisfiability

• A Boolean combinational circuit consists of one or more Boolean combinational elements interconnected by wires

SAT: \(<x_1 = 1, x_2 = 1, x_3 = 0>\)
Exercise: Circuit Satisfiability

• Is this Boolean combinational circuit satisfiable?

\[ <x_1 = ?, x_2 = ?, x_3 = ?> \]
Circuit-Satisfiability Problem

- Given a **Boolean combinational circuit** of AND, OR, and NOT gates, is it **satisfiable**?

**CIRCUIT-SAT** = \{<C> : C is a satisfiable Boolean combinational circuit\}

- **Size**: number of **Boolean combinational elements** plus the number of wires
  - if the circuit has **k inputs**, then we would have to check up to \(2^k\) possible assignments

- When the **size of C** is polynomial in \(k\), checking each one takes \(\Omega(2^k)\)
  - Super-polynomial in the size of \(k\)
CIRCUIT-SAT Belongs to NP

Lemma 1: The circuit-satisfiability problem belongs to the class NP

• Proof:
  - Provide a two-input, polynomial-time algorithm $A$ that can verify CIRCUIT-SAT:
    - A Boolean combinational circuit $C$
    - A certificate corresponding to an assignment of Boolean values to the wires in $C$

$x_1 = 1, x_2 = 1, x_3 = 0$
For each logic gate, we check the value provided by the certificate

- the output wire is correctly computed as a function of the values on the input wires

If the output of the entire circuit is 1, the algorithm outputs 1

- the values assigned to the inputs of C provide a satisfying assignment

Otherwise, outputs 0
CIRCUIT-SAT Belongs to NP

- Whenever a **satisfiable circuit** is input to algorithm $A$, there exists a **certificate** whose length is polynomial in the size of $C$ and that causes $A$ to output $1$

- Whenever an **unsatisfiable circuit** is input, no certificate can fool $A$ into believing that the circuit is satisfiable

- Alg. $A$ runs in polynomial-time: CIRCUIT-SAT $\in$ NP
Lemma 2: The circuit-satisfiability problem is NP hard

- Let $L$ be any language in $NP$
- Describe a polynomial-time algorithm $F$ computing a reduction function $f$ that maps every binary string $x$ to a circuit $C = f(x)$
  - $x \in L$ iff $C \in$ CIRCUIT-SAT

$x \in L$ iff $C \in$ CIRCUIT-SAT
Reducibility Assumptions

The algorithm $F$ uses the two-input algorithm $A$ to compute the reduction function $f$

- Let $T(n)$ denote the **worst-case running time** of algorithm $A$ on length-$n$ input strings
- Let $k \geq 1$ be a constant such that $T(n) = O(n^k)$ and the **certificate length** is $O(n^k)$

The basic idea of the proof is to represent the computation of $A$ as a sequence of configurations
Computer Architecture

- Consists of combinational circuit, program counter (PC), auxiliary machine state, and working storage.
The sequence of configurations produced by an algorithm $A$ running on an input $x$ and certificate $y$

$M$ includes the combinational circuit to verify CIRCUIT-SAT
We need to prove two properties:

1. We must show that $F$ correctly computes a reduction function $f$
   - Show that $C$ is satisfiable \textit{iff} there exists a certificate $y$ such that $A(x,y) = 1$

2. We must show that $F$ runs in \textit{polynomial-time}
NP-hard Properties for CIRCUIT-SAT

• Property 1:
  
  ▪ Assume that there exists a certificate $y$ of length $O(n^k)$ such that $A(x, y) = 1$
  
  ▪ If we apply the bits of $y$ to the inputs of $C$, the output of $C$ is $C(y) = A(x, y) = 1$
    
      ○ If a certificate exists, then $C$ is satisfiable

Certificate $y$:
$x_1 = 1, x_2 = 1, x_3 = 0$  \[ C(y) = 1 \quad A(x,y) = 1 \]
NP-hard Properties for CIRCUIT-SAT

• Property 1:
  • For the other direction, suppose that \( C \) is satisfiable
  • Hence, there exists an input \( y \) to \( C \) such that \( C(y) = 1 \), from which we conclude that \( A(x,y) = 1 \)

Thus, \( F \) correctly computes a reduction function
NP-hard Properties for CIRCUIT-SAT

• Property 2:
  - We need show that $F$ runs in **polynomial-time** in $n = |x|$
  - The number of bits required to represent a configuration is polynomial in $n$
    - The program for $A$ itself has **constant size**, independent of the length of its input $x$
    - The length of the input $x$ is $n$, and the length of the certificate $y$ is $O(n^k)$
NP-hard Properties for CIRCUIT-SAT

• Property 2:
  - Since the algorithm $A$ runs for at most $O(n^k)$ steps, the amount of working storage required by $A$ is polynomial in $n$.
  - $M$ that implements the computer hardware has size polynomial in the length of a configuration.
    - The circuit $C$ consists of at most $t = O(n^k)$ copies of $M$, and hence it has size polynomial in $n$.

The reduction algorithm $F$ can construct $C$ from $x$ in polynomial-time, since each step of the construction takes polynomial-time.
Proving NP-Completeness

• Prove NP-completeness of a language $L$ by reduction consists of the following steps

1. Prove $L \in \text{NP}$
2. Select a known \textbf{NP-complete language $L'$}
3. Describe an algorithm that computes a \textbf{function $f$} mapping every instance $x \in \{0, 1\}^*$ of $L'$ to an instance $f(x)$ of $L$
4. Prove that the function $f$ satisfies $x \in L'$ iff $f(x) \in L$ for all $x \in \{0, 1\}^*$
5. Prove that the algorithm computing $f$ runs in polynomial-time
Formula Satisfiability (SAT)

- The SAT problem asks whether a given Boolean formula is satisfiable

\[ \text{SAT} = \{<\Phi>: \Phi \text{ is a satisfiable Boolean formula}\} \]

- Example:
  - \( \Phi = ((x_1 \rightarrow x_2) \lor \neg(((\neg x_1 \leftrightarrow x_3) \lor x_4)) \land \neg x_2 \)
  - Assignment: \(<x_1 = 0, x_2 = 0, x_3 = 1, x_4 = 1>\)
  - \( \Phi = ((0 \rightarrow 0) \lor \neg((\neg 0 \leftrightarrow 1) \lor 1)) \land \neg 0 \)
  - \( \Phi = (1 \lor \neg(1 \lor 1)) \land 1 \)
  - \( \Phi = (1 \lor 0) \land 1 \)
  - \( \Phi = 1 \)
Theorem 1: Satisfiability of Boolean formulas is NP-complete

1: SAT $\in \mathcal{NP}$: The verifying algorithm replaces each variable in $\Phi$ with its corresponding value
   - If the expression evaluates to 1, then $\Phi$ is SAT (polynomial-time);
   - otherwise it is UNSAT (polynomial-time)

2: SAT is NP-hard: Show that $\text{CIRCUIT-SAT} \leq_p \text{SAT}$
   - Show how to reduce any instance of circuit satisfiability to an instance of formula satisfiability in polynomial-time
CIRCUIT-SAT $\leq_p$ SAT

For each wire $x_i$ in the circuit $C$, the formula $\Phi$ has a variable $x_i$

\[
\phi = x_{10} \land (x_4 \leftrightarrow \neg x_3) \\
\land (x_5 \leftrightarrow (x_1 \lor x_2)) \\
\land (x_6 \leftrightarrow \neg x_4) \\
\land (x_7 \leftrightarrow (x_1 \land x_2 \land x_4)) \\
\land (x_8 \leftrightarrow (x_5 \lor x_6)) \\
\land (x_9 \leftrightarrow (x_6 \lor x_7)) \\
\land (x_{10} \leftrightarrow (x_7 \land x_8 \land x_9))
\]
CIRCUIT-SAT $\leq_p$ SAT

- Why is the circuit $C$ satisfiable exactly when the formula $\Phi$ is satisfiable?
  - If $C$ has a satisfying assignment, then each wire of the circuit has a defined value, and the circuit output is 1
    - When we assign wire values to variables in $\Phi$, each clause of $\Phi$ evaluates to 1, and the conjunction of all evaluates to 1
  - If some assignment causes $\Phi$ to evaluate to 1, the circuit $C$ is satisfiable by an analogous argument

Thus, we have shown that CIRCUIT-SAT $\leq_p$ SAT which completes the proof
Exercise: CIRCUIT-SAT to SAT

• Convert the following Boolean circuit to SAT
Exercise: CIRCUIT-SAT to SAT

• Convert the following Boolean circuit to SAT

\[ \Phi = Q \land (D \leftrightarrow (\neg A \lor \neg B)) \]
\[ \land (E \leftrightarrow (\neg A \land \neg B)) \]
\[ \land (Q \leftrightarrow (D \land C \land E)) \]

\[ \begin{align*}
\text{<A=0, B=0, C=1>} & \\
\Phi &= Q \land (D \leftrightarrow (1 \lor 1)) \\
& \land (E \leftrightarrow (1 \land 1)) \\
& \land (Q \leftrightarrow (D \land 1 \land E)) \\
\end{align*} \]

\[ \begin{align*}
\text{<A=0, B=1, C=0>} & \\
\Phi &= Q \land (D \leftrightarrow (1 \lor 0)) \\
& \land (E \leftrightarrow (1 \land 0)) \\
& \land (Q \leftrightarrow (D \land 0 \land E)) \\
\end{align*} \]
Conjunctive Normal Form

- We define **conjunctive normal form (CNF)** as
  - **Literal**: an occurrence of a **Boolean** or its **negation**
  - A Boolean formula is in **CNF**, if it is an **AND of clauses**, consisting of an **OR of one or more literals**
    - Ex: \((x_1 \lor \neg x_2) \land (\neg x_1 \lor x_3 \lor x_4) \land (\neg x_5)\)
  - **3-CNF**: each clause has exactly 3 distinct literals
    - Ex: \((x_1 \lor \neg x_2 \lor \neg x_3) \land (\neg x_1 \lor x_3 \lor x_4) \land (\neg x_5 \lor x_3 \lor x_4)\)
    - Notice: **true** if at least one literal in each clause is **true**

- In 3-CNFSAT, we are asked whether a given Boolean formula \(\Phi\) in 3-CNFS is **satisfiable**
The 3-CNF Problem

1: **3-CNF-SAT** ∈ **NP**: Theorem 1 (SAT ∈ NP) applies equally well here to show that 3-CNF-SAT ∈ NP

2: **3-CNF is NP-hard**:

   2.1: construct a binary “parse” tree for the input formula Φ, with literals as leaves and connectives as internal nodes

\[
\phi = ((x_1 \rightarrow x_2) \lor \neg((\neg x_1 \leftrightarrow x_3) \lor x_4)) \land \neg x_2
\]
The 3-CNF Problem

2.2: Rewrite \( \Phi \) as the **AND of the root variable** and a **conjunction of clauses** describing the operation of each node:

\[
\phi' = y_1 \land (y_1 \leftrightarrow (y_2 \land \neg x_2)) \\
\land (y_2 \leftrightarrow (y_3 \lor y_4)) \\
\land (y_3 \leftrightarrow (x_1 \rightarrow x_2)) \\
\land (y_4 \leftrightarrow \neg y_5) \\
\land (y_5 \leftrightarrow (y_6 \lor x_4)) \\
\land (y_6 \leftrightarrow (\neg x_1 \leftrightarrow x_3)).
\]

The only requirement that we might fail to meet is that each clause has to be an OR of 3 literals.
The 3-CNF Problem

2.3: Convert each clause $\Phi'_i$ into CNF

**Step A:** Construct a truth table for $\Phi'_i$ by evaluating all possible assignments to its variables

**Step B:** Using the truth-table entries that evaluate to 0, build a formula in disjunctive normal form (or DNF) — an OR of ANDs — that is equivalent to $\neg \Phi'_i$

**Step C:** Negate $\neg \Phi'_i$ and convert it into a CNF formula $\Phi''_i$ by using DeMorgan’s law

\[ \neg (a \land b) = \neg a \lor \neg b \, , \]
\[ \neg (a \lor b) = \neg a \land \neg b \, , \]

*to complement all literals, change ORs into ANDs, and change ANDs into ORs*
Exercise: The 3-CNF Problem

Converts each clause $\Phi_1'$ into CNF

$$\phi_1' = (y_1 \leftrightarrow (y_2 \land \neg x_2))$$

**Step A:** Build the truth-table of $\Phi_1'$

**Step B:** Convert $\Phi_1'$ to DNF formula (an OR of ANDs)

**Step C:** Negate and apply DeMorgan’s laws to obtain the CNF formula
Exercise: The 3-CNF Problem

Step A: Build the truth-table

<table>
<thead>
<tr>
<th>$y_1$</th>
<th>$y_2$</th>
<th>$x_2$</th>
<th>($y_1 \leftrightarrow (y_2 \land \neg x_2)$)</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
</tbody>
</table>

Step B: Convert $\Phi'_1$ to DNF formula (an OR of ANDs)

$$\neg \Phi'_1 = (y_1 \land y_2 \land y_3) \lor (y_1 \land \neg y_2 \land x_2) \lor (y_1 \land \neg y_2 \land \neg x_2) \lor (\neg y_1 \land y_2 \land \neg x_2)$$
Exercise: The 3-CNF Problem

Step C: Negate and apply DeMorgan’s laws to obtain the CNF formula:

\[ \neg \Phi'_1 = (y_1 \land y_2 \land y_3) \lor (y_1 \land \neg y_2 \land x_2) \lor (y_1 \land \neg y_2 \land \neg x_2) \lor (\neg y_1 \land y_2 \land \neg x_2) \]

\[ \Phi''_1 = (\neg y_1 \lor \neg y_2 \lor \neg y_3) \land (\neg y_1 \lor y_2 \lor x_2) \land (\neg y_1 \lor y_2 \lor x_2) \land (y_1 \lor \neg y_2 \lor x_2) \]

\[ \Phi''_1 \] is equivalent to the original clause \( \Phi'_1 \)
The 3-CNF Problem

We must also show that the reduction can be computed in polynomial-time.

• Constructing Φ’ from Φ introduces at most 1 variable and 1 clause per connective in Φ.
• Constructing Φ” from Φ’ can introduce at most 8 clauses into Φ” for each clause from Φ’.
  - since each clause of Φ’ has at most 3 variables, and the truth table for each clause has at most $2^3 = 8$ rows.

The size of the resulting formula Φ” is polynomial in the length of the original formula.
Examples of NP-Complete Problems

• Given one NP-Complete problem, we can prove many interesting problems NP-Complete
  ▪ Graph coloring (= register allocation)
Examples of NP-Complete Problems

• Given one NP-Complete problem, we can prove many interesting problems NP-Complete
  ▪ Graph coloring (= register allocation)
  ▪ Hamiltonian cycle
Examples of NP-Complete Problems

• Given one NP-Complete problem, we can prove many interesting problems NP-Complete
  • Graph coloring (= register allocation)
  • Hamiltonian cycle
  • Knapsack problem
Examples of NP-Complete Problems

- Given one NP-Complete problem, we can prove many interesting problems NP-Complete
  - Graph coloring (= register allocation)
  - Hamiltonian cycle
  - Knapsack problem
  - Traveling salesman
Examples of NP-Complete Problems

• Given one NP-Complete problem, we can prove many interesting problems NP-Complete
  - Graph coloring (= register allocation)
  - Hamiltonian cycle
  - Knapsack problem
  - Traveling salesman
  - Job scheduling
Examples of NP-Complete Problems

- Given one NP-Complete problem, we can prove many interesting problems NP-Complete
  - Graph coloring (= register allocation)
  - Hamiltonian cycle
  - Knapsack problem
  - Traveling salesman
  - Job scheduling
  - Equivalence checking

```plaintext
if(!a&&!b) h();
else if(!a) g();
else f();
if(a)f();
else if(b) g();
else h();
```
Summary

• No polynomial-time algorithm has yet been discovered for an NP-complete problem
  ▪ To become a good algorithm designer, you must understand the theory of NP-completeness

• Various problems have been shown to be NP-Complete
  ▪ Some reductions are profound, some are comparatively easy, many are easy once the key insight is given

• You can expect a simple NP-Completeness proof on the final