Textbook

• *Algorithm Design and Applications*, Goodrich and Tamassia (Chapter 17)
• *Introduction to Algorithms*, Cormen, Leiserson, Rivest, and Stein (Chapter 34)
• *Introduction to the Theory of Computation*, Michael Sipser (Chapters 4 and 7)
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
Automated Verification

• Given a computer **program** and a precise **specification** (e.g., sort a list of numbers)

• We need to **verify** that the **program** performs as specified

• Since both program and specification are **mathematical objects**, we could **automate the verification process** by a computer
Termination Analysis

- Given this C program and any unsigned integer input, will the program terminate or will it run forever?

```c
int main() {
  unsigned int n = __VERIFIER_nondet_uint();
  unsigned int x=n, y=0;
  while(x>0) {
    x--;
    y++;
  }
  __VERIFIER_assert(y==n);
}
```
Termination Analysis

• Given this C program and any integer input, will the program terminate or will it run forever?

```c
int main() {
    int y = __VERIFIER_nondet_int();
    while (y >= 0 && y <= 10) {
        y = (2*y + 1) / 2;
    }
    return 0;
}
```

Does not terminate for 0 <= x <= 10 due to rounding in integer division
Termination Analysis

• Given this C program and any integer input, will the program terminate or will it run forever?

```c
Void funcA(int *y1, int *y2) {
    while (*y1 != *y2) {
        if (*y1 > *y2) *y1 = *y1 - *y2;
        else *y2 = *y2 - *y1;
    }
}
int main() {
    int *y1 = alloca(sizeof(int)), *y2 = alloca(sizeof(int));
    *y1, *y2 = __VERIFIER_nondet_int();
    if (*y1 >= 0 && *y2 >= 0) funcA(y1, y2);
    return 0;
}
```

there is no verifier that can decide about termination for general enough programs
Turing’s “Halting Problem” (COMP11212)

• In 1936, Alan Turing proved that there exists no algorithm that
  - given a description of an arbitrary computer program and an input
  - decide whether that program will terminate or continue running forever for all possible inputs

• The general problem of program verification is not solvable by computers
  - Unbounded memory usage
Tractability

• What are **tractable** and **intractable** problems?
• Some problems are **intractable**: as inputs grow, we are unable to solve them in reasonable time
• What constitutes **reasonable time**?
  - A: **polynomial-time**
    - On an input of size $n$ the worst-case running time is $O(n^k)$ for some constant $k$
  - Polynomial-time: $O(n)$, $O(n^2)$, $O(n^3)$, $O(1)$, $O(n \log n)$
  - Not in polynomial-time: $O(2^n)$, $O(n^n)$, $O(n!)$
Growth Functions

"E" represents "times ten raised to the power of"
Exercise: Comparison of Running Times

For each function \( f(n) \) and time \( t \), determine the largest size \( n \) of a problem that can be solved in time \( t \)

- the algorithm to solve the problem takes \( f(n) \) microseconds

<table>
<thead>
<tr>
<th>Time (s)</th>
<th>( \log n )</th>
<th>( n^{1/2} )</th>
<th>( n )</th>
<th>( n^2 )</th>
<th>( n^3 )</th>
<th>( 2^n )</th>
<th>( n! )</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 second</td>
<td>2(^{1.1 \times 10^6})</td>
<td>1(x)10(^{12})</td>
<td>1(x)10(^6)</td>
<td>1000</td>
<td>100</td>
<td>19</td>
<td>9</td>
</tr>
</tbody>
</table>

\[
\log n = 1 \times 10^6 \quad \therefore \quad n = 2^{1.1 \times 10^6}
\]

\[
\sqrt{n} = 1 \times 10^6 \quad \therefore \quad \left(\sqrt{n}\right)^2 = (1 \times 10^6)^2 \quad \therefore \quad n = 1 \times 10^{12}
\]

\[
n^2 = 1 \times 10^6 \quad \therefore \quad n = \sqrt{1 \times 10^6} \quad \therefore \quad n = 1000
\]

\[
n^3 = 1 \times 10^6 \quad \therefore \quad n = \sqrt[3]{1 \times 10^6} \quad \therefore \quad n = 100
\]

\[
2^n = 1 \times 10^6 \quad \therefore \quad \log 2^n = \log 1 \times 10^6 \quad \therefore \quad n = 6 / \log 2 \approx 19
\]
Exercise: Comparison of Running Times

• For each function $f(n)$ and time $t$, determine the largest size $n$ of a problem that can be solved in time $t$
  
  ▪ the algorithm to solve the problem takes $f(n)$ microseconds

<table>
<thead>
<tr>
<th>$\lg n$</th>
<th>1 Second</th>
<th>1 Minute</th>
<th>1 Hour</th>
<th>1 Day</th>
<th>1 Month</th>
<th>1 Year</th>
<th>1 Century</th>
</tr>
</thead>
<tbody>
<tr>
<td>$2^{10^6}$</td>
<td>$2^{6\times10^7}$</td>
<td>$2^{3.6\times10^9}$</td>
<td>$2^{8.64\times10^{10}}$</td>
<td>$2^{2.592\times10^{12}}$</td>
<td>$2^{3.1536\times10^{13}}$</td>
<td>$2^{3.15576\times10^{15}}$</td>
<td></td>
</tr>
<tr>
<td>$\sqrt{n}$</td>
<td>$1 \times 10^{12}$</td>
<td>$3.6 \times 10^{15}$</td>
<td>$1.29 \times 10^{19}$</td>
<td>$7.46 \times 10^{21}$</td>
<td>$6.72 \times 10^{24}$</td>
<td>$9.95 \times 10^{26}$</td>
<td></td>
</tr>
<tr>
<td>$n$</td>
<td>$1 \times 10^{6}$</td>
<td>$6 \times 10^{7}$</td>
<td>$3.6 \times 10^{9}$</td>
<td>$8.64 \times 10^{10}$</td>
<td>$2.59 \times 10^{12}$</td>
<td>$3.15 \times 10^{13}$</td>
<td></td>
</tr>
</tbody>
</table>

| $n^2$ | 1000 | 7745 | 60000 | 293938 | 1609968 | 5615692 | 56176151 |
| $n^3$ | 100  | 391  | 1532  | 4420   | 13736   | 31593   | 146679   |
| $2^n$ | 19   | 25   | 31    | 36     | 41      | 44      | 51       |
| $n!$  | 9    | 11   | 12    | 13     | 15      | 16      | 17       |

Assume a 30 day month and 365 day year
Polynomial-Time Algorithms

• Are **some** problems solvable in polynomial time?
  - **Yes:** Every algorithm we have studied so far provides polynomial-time solution to some problem
  - We define $P$ to be the class of problems solvable in polynomial time (**tractable** or **easy**)

• Are **all** problems solvable in polynomial-time?
  - **No:** Turing’s “Halting Problem”
    - Verification of multi-threaded programs
  - Verification of bounded multi-threaded programs
    - **Intractable** (or hard), not in $P$
Summary of Tractability

- **Computational challenge**
- Polynomial time
- Not in Polynomial time
- Turing Halting Problem
- Decidable
- Undecidable

- Polynomial time
- Turing Halting Problem
What is P and NP?

- **P** is set of problems that can be solved in polynomial-time
  - problems that can be solved in $O(n^k)$ for some constant $k$, where $n$ is the size of the input

- **NP** (nondeterministic polynomial-time) is the set of problems that can be solved in polynomial-time by a **nondeterministic** computer
  - What does it mean?
Nondeterminism

- Think of a **non-deterministic computer** as a computer that magically “guesses” a solution, then has to **verify** that it is correct
  - If a solution exists, computer always guesses it
  - Imagine a **parallel computer** that can freely **spawn** an **unbounded number of processes**

- **So: NP = problems verifiable** in polynomial-time
NP-complete Problems

Definition 1: A decision problem is in the class NP-complete if it is in NP and is as “hard” as any problem in NP.

A decision problem has two possible outputs on any input.

- Circuit Satisfiability
**NP-complete Problems**

**Definition 1:** A decision problem is in the class NP-complete if it is in NP and is as “hard” as any problem in NP.

- Circuit Satisfiability
- Formula Satisfiability

\[ \Phi = ((x_1 \rightarrow x_2) \lor \neg((\neg x_1 \leftrightarrow x_3) \lor x_4)) \]

A decision problem has **two possible outputs** on any input.
NP-complete Problems

Definition 1: A decision problem is in the class NP-complete if it is in NP and is as “hard” as any problem in NP.

A decision problem has two possible outputs on any input.
NP-complete Problems

Definition 1: A decision problem is in the class NP-complete if it is in NP and is as “hard” as any problem in NP.

A decision problem has two possible outputs on any input.

- Circuit Satisfiability
- Formula Satisfiability
- Equivalence checking
- $k$-Clique
NP-complete Problems

**Definition 1:** A decision problem is in the class NP-complete if it is in NP and is as “hard” as any problem in NP.

- The class NP consists of those (decision) problems that are “verifiable” in polynomial-time.

---

Certificate: \(<x_1 = 1, x_2 = 1, x_3 = 0>\)

Algorithm: Circuit Satisfiability

**Yes**

Satisfiable

Verify that the certificate is correct in time polynomial in the size of the input.
The NP-complete problems (or simply NPC) are an interesting class of problems:

- No polynomial-time algorithm has been discovered for any NP-Complete problem.

- $P \neq NP$ question? The biggest open problem in CS since it was first posed in 1971.

If you can establish a problem as NP-complete, then it is intractable unless $P = NP$:

- Do not search for a fast algorithm that solves the problem exactly!

- Approximation algorithm or tractable special case.
NP-complete problems are particularly tantalizing

• Several NP-complete problems seem similar to problems that are solved in polynomial-time
  ▪ Shortest vs. longest simple paths:
    o Find *shortest paths* from a single source in a directed graph $G=(V, E)$ is $O(VE)$ time
    o Find a *longest simple path* between two vertices is NPC
NP-complete problems are particularly tantalizing

- Euler tour vs. Hamiltonian cycle:
  - An Euler tour of a connected, directed graph $G=(V,E)$ is a cycle that traverses each edge of $G$ exactly once in $O(E)$
  - A Hamiltonian cycle of a directed graph $G=(V,E)$ is a simple cycle that contains each vertex in $V$ exactly once (NPC)

Euler tour:
$R-D-A-B-C-D-C-A-R$

Hamiltonian cycle shown by shadowed edges
Compute Hamiltonian Cycles

- The Hamiltonian-cycle problem: given a graph $G$, does it have a Hamiltonian cycle?
  - one possible decision algorithm lists all permutations of the vertices of $G$ and checks if each one is a Hamiltonian cycle
  - There are $m!$ possible permutations of the vertices
    - the running time is $O(m!)$, where $m$ depends on the graph encoding (e.g., adjacency matrix)
  - Verify that the cycle is Hamiltonian by checking if it is a permutation of the vertices of $V$
    - if each of the consecutive edges along the cycle exists in the graph
Example of a solution verified in polynomial-time

• In the Hamiltonian-cycle problem, given a directed graph $G=(V,E)$, a solution would be a sequence $\langle v_1, v_2, v_3, ..., v_{|V|} \rangle$ of $n$ vertices.

• We could check in polynomial-time that $(v_i, v_{i+1})$ in $E$ for $i = 1, 2, 3, ..., n-1$ and that $(v_n, v_1)$ in $E$ as well.

![Diagram of a Hamiltonian cycle](image)
Exercise: Formula Satisfiability

• Is this logical formula SAT? \((A \rightarrow B) \land A \land \neg B\)

Approach 1: Enumeration \((A \rightarrow B)\) is equivalent to \neg A \lor B

<p>| | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>B</td>
<td>A \rightarrow B</td>
<td>A</td>
<td>\neg B</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
</tbody>
</table>

→ Processor 1  → Processor 2  → Processor 3  → Processor 4

Approach 2: Deduction

\((A \rightarrow B)\) \quad A \quad B \quad \neg B \quad \text{Can we parallelise this?}

B
FALSE
Exercise: Equivalence Checking of Programs

• Are these two code fragments equivalent?

\[
\Phi_1 = \left( (\neg a \land \neg b) \land h \right) \lor \left( (a \land g) \lor (a \land f) \right)
\]
\[
\Phi_2 = \left( (a \land f) \lor (a \land (b \land g) \lor (b \land h)) \right)
\]

Are \( \Phi_1 \) and \( \Phi_2 \) equivalent?
Can we check a certificate in polynomial-time?
Summary of P and NP

• Summary so far:
  - **P** = problems that can be *solved* in polynomial time
  - **NP** = problems for which a *certificate* can be *verified* in *polynomial-time*
    - We can believe that **P ⊆ NP**

• Formula Satisfiability, Equivalence Checking and Hamiltonian-cycle problems are in **NP**:
  - Cannot solve in polynomial-time
  - Easy to verify a certificate in polynomial-time
Summary of NP-Complete Problems

• NP-Complete problems are the “hardest” problems in NP:
  ▪ If any one NP-Complete problem can be solved in polynomial-time…
  ▪ …then every NP-Complete problem can be solved in polynomial-time…
  ▪ …and in fact every problem in \( \text{NP} \) can be solved in polynomial-time (which would show \( \text{P} = \text{NP} \))
  ▪ Despite years of study, no polynomial-time algorithm has ever been discovered for any NP-complete problem
Reduction

• The crux of NP-Completeness is reducibility
  
  ▪ A problem $A$ can be reduced to another problem $B$ if any instance $\alpha$ of $A$ can be transformed into some instance of $\beta$ of $B$:
    
    o The transformation takes polynomial-time
  
    o The answer for $\alpha$ is “yes” iff the answer for $\beta$ is also “yes”

  ▪ If $A$ reduces to $B$, $A$ is “no harder to solve” than $B$

  ▪ We are trying to prove that no efficient algorithm is likely to exist
Polynomial-time Reduction

1) Given an instance $\alpha$ of problem $A$, use a polynomial-time reduction algorithm
2) Transform it to an instance $\beta$ of problem $B$
3) Run the polynomial-time decision algorithm for $B$ on the instance $\beta$
4) Use the answer for $\beta$ as the answer for $\alpha$
Reducibility Example: SAT to 3-CNF-SAT

- **A**: A Boolean formula that contains
  - variables whose values are 0 or 1
  - connectives such as $\land$ (AND), $\lor$ (OR), and $\neg$ (NOT)

- **B**: A Boolean formula that is in 3-CNF
  - AND of clauses, each clause has exactly 3 distinct literals

\[ A: \Phi = ((x_1 \rightarrow x_2) \lor \neg((\neg x_1 \leftrightarrow x_3) \lor x_4)) \land \neg x_2 \]

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

\[ B: \Phi'' = (\neg y_1 \lor \neg y_2 \lor \neg y_3) \land (\neg y_1 \lor y_2 \lor x_2) \land (\neg y_1 \lor y_2 \lor x_2) \land (y_1 \lor \neg y_2 \lor x_2) \]
Exercise: SAT to 3-CNF-SAT

• Convert the following Boolean formula to 3-CNF:

\[ x_1 \rightarrow (x_2 \land x_3) \]

\[ y_1 \leftrightarrow (x_1 \rightarrow y_2) \]
\[ y_2 \leftrightarrow (x_2 \land x_3) \]
Exercise: SAT to 3-CNF-SAT

\( \Phi'_1 = y_1 \leftrightarrow (x_1 \rightarrow y_2) \)

<table>
<thead>
<tr>
<th>( y_1 )</th>
<th>( x_1 )</th>
<th>( y_2 )</th>
<th>( (\neg x_1 \lor y_2) )</th>
<th>( y_1 \leftrightarrow (x_1 \rightarrow y_2) )</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
</tbody>
</table>

\( \neg \Phi'_1 \)

\( (y_1 \land x_1 \land \neg y_2) \lor \)
\( (\neg y_1 \land x_1 \land y_2) \lor \)
\( (\neg y_1 \land \neg x_1 \land y_2) \lor \)
\( (\neg y_1 \land \neg x_1 \land \neg y_2) \lor \)

\( \Phi''_1 \)

\( (\neg y_1 \lor \neg x_1 \lor y_2) \land \)
\( (y_1 \lor \neg x_1 \lor \neg y_2) \land \)
\( (y_1 \lor x_1 \lor \neg y_2) \land \)
\( (y_1 \lor x_1 \lor y_2) \land \)
Exercise: SAT to 3-CNF-SAT

\[ \Phi'_2 = y_2 \leftrightarrow (x_2 \land x_3) \]

<table>
<thead>
<tr>
<th>(y_2)</th>
<th>(x_2)</th>
<th>(x_3)</th>
<th>((x_2 \land x_3))</th>
<th>(y_2 \leftrightarrow (x_2 \land x_3))</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
</tbody>
</table>

\[ \neg \Phi'_2 \]
\[ (y_2 \land x_2 \land \neg x_3) \lor \]
\[ (y_2 \land \neg x_2 \land x_3) \lor \]
\[ (y_2 \land \neg x_2 \land \neg x_3) \lor \]
\[ (\neg y_2 \land x_2 \land x_3) \lor \]

\[ \Phi''_2 \]
\[ (\neg y_2 \lor \neg x_2 \lor x_3) \land \]
\[ (\neg y_2 \lor x_2 \lor \neg x_3) \land \]
\[ (\neg y_2 \lor x_2 \lor x_3) \land \]
\[ (y_2 \lor \neg x_2 \lor \neg x_3) \land \]
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