Efficient Sequential Algorithms, Comp309

University of Liverpool

2010–2011
Module Organiser, Igor Potapov
Part 4: NP-Completeness

References: T. H. Cormen, C. E. Leiserson, R. L. Rivest


A decision problem is a computational problem for which the output is either yes or no.

The input to a computational problem is encoded as a finite binary string $s$ of length $|s|$.

For a decision problem $X$, $L(X)$ denotes the set of (binary) strings (inputs) for which the algorithm should output “yes”. We refer to $L(X)$ as a language. We say that an algorithm $A$ accepts a language $L(X)$ if $A$ outputs “yes” for each $s \in L(X)$ and outputs “no” for every other input.
A decision problem is a computational problem for which the output is either yes or no.

The input to a computational problem is encoded as a finite binary string $s$ of length $|s|$.

For a decision problem $X$, $L(X)$ denotes the set of (binary) strings (inputs) for which the algorithm should output “yes”. We refer to $L(X)$ as a language. We say that an algorithm $A$ accepts a language $L(X)$ if $A$ outputs “yes” for each $s \in L(X)$ and outputs “no” for every other input.
A decision problem is a computational problem for which the output is either yes or no.

The input to a computational problem is encoded as a finite binary string $s$ of length $|s|$.

For a decision problem $X$, $L(X)$ denotes the set of (binary) strings (inputs) for which the algorithm should output “yes”. We refer to $L(X)$ as a language. We say that an algorithm $A$ accepts a language $L(X)$ if $A$ outputs “yes” for each $s \in L(X)$ and outputs “no” for every other input.
The complexity class $P$ is the set of all decision problems $X$ (or languages $L(X)$) that can be solved in polynomial time.

That is, there is an algorithm $A$ that accepts language $L(X)$. The amount of time that algorithm $A$ takes on input $s$ is at most $p(|s|)$ where $p(n)$ is of the form $p(n) = n^k$ for a some constant $k$. ($p(n)$ is a \textit{polynomial} in $n$).
The complexity class $P$ is the set of all decision problems $X$ (or languages $L(X)$) that can be solved in polynomial time.

That is, there is an algorithm $A$ that accepts language $L(X)$. The amount of time that algorithm $A$ takes on input $s$ is at most $p(|s|)$ where $p(n)$ is of the form $p(n) = n^k$ for a some constant $k$. ($p(n)$ is a polynomial in $n$).
The complexity class **EXP** is the set of all decision problems $X$ (or languages $L(X)$) that can be solved in exponential time.

That is, there is an algorithm $A$ that accepts language $L(X)$. The amount of time that algorithm $A$ takes on input $s$ is at most $p(|s|)$ where $p(n)$ is a function of the form $p(n) = 2^{n^k}$ for some constant $k$. 
The complexity class **EXP** is the set of all decision problems $X$ (or languages $L(X)$) that can be solved in exponential time.

That is, there is an algorithm $A$ that accepts language $L(X)$. The amount of time that algorithm $A$ takes on input $s$ is at most $p(|s|)$ where $p(n)$ is a function of the form $p(n) = 2^{nk}$ for some constant $k$. 
Space complexity classes

PSPACE is the set of all decision problems \( X \) (or languages \( L(X) \)) that can be solved in polynomial space.

That is, there is an algorithm \( A \) that accepts language \( L(X) \). The amount of computer memory that algorithm \( A \) uses on input \( s \) is at most \( p(|s|) \) where \( p(n) \) is a polynomial in \( n \).
Space complexity classes

PSPACE is the set of all decision problems $X$ (or languages $L(X)$) that can be solved in polynomial space.

That is, there is an algorithm $A$ that accepts language $L(X)$. The amount of computer memory that algorithm $A$ uses on input $s$ is at most $p(|s|)$ where $p(n)$ is a polynomial in $n$. 
An algorithm that guesses some number of non-deterministic bits during its execution is called a non-deterministic algorithm.

We say that a non-deterministic algorithm $A$ accepts a string $s$ if there exists a choice of non-deterministic bits that causes algorithm $A$ to output “yes” with input $s$. Otherwise, we say that $A$ does not accept $s$.

We say that a non-deterministic algorithm $A$ accepts a language $L(X)$ if $A$ accepts every string $s \in L(X)$ and no other strings.
An algorithm that guesses some number of non-deterministic bits during its execution is called a non-deterministic algorithm.

We say that a non-deterministic algorithm $A$ accepts a string $s$ if there exists a choice of non-deterministic bits that causes algorithm $A$ to output “yes” with input $s$. Otherwise, we say that $A$ does not accept $s$.

We say that a non-deterministic algorithm $A$ accepts a language $L(X)$ if $A$ accepts every string $s \in L(X)$ and no other strings.
An algorithm that guesses some number of non-deterministic bits during its execution is called a non-deterministic algorithm.

We say that a non-deterministic algorithm $A$ accepts a string $s$ if there exists a choice of non-deterministic bits that causes algorithm $A$ to output “yes” with input $s$. Otherwise, we say that $A$ does not accept $s$.

We say that a non-deterministic algorithm $A$ accepts a language $L(X)$ if $A$ accepts every string $s \in L(X)$ and no other strings.
The complexity class **NP** is the set of all decision problems $X$ (or languages $L(X)$) that can be non-deterministically accepted in polynomial time.

That is, there is a non-deterministic algorithm $A$ that accepts language $L(X)$. The amount of time that algorithm $A$ takes on input $s$ is at most $p(|s|)$ where $p(n)$ is a polynomial in $n$. 
The complexity class $NP$ is the set of all decision problems $X$ (or languages $L(X)$) that can be non-deterministically accepted in polynomial time.

That is, there is a non-deterministic algorithm $A$ that accepts language $L(X)$. The amount of time that algorithm $A$ takes on input $s$ is at most $p(|s|)$ where $p(n)$ is a polynomial in $n$. 
The complexity class NP is the set of all decision problems $X$ (or languages $L(X)$) that can be non-deterministically accepted in polynomial time.

That is, there is a non-deterministic algorithm $A$ that accepts language $L(X)$. The amount of time that algorithm $A$ takes on input $s$ is at most $p(|s|)$ where $p(n)$ is a polynomial in $n$. 
It is easy to see that $P \subseteq NP$

If $L(X)$ is accepted by a polynomial-time algorithm $A$ then it is also accepted by a non-deterministic algorithm in polynomial time.

The non-deterministic algorithm doesn’t have to make non-deterministic choices — it can just simulate algorithm $A$. 
We say that a language $L$, defining some decision problem, is **polynomial-time reducible** to a language $M$ (written $L \xrightarrow{\text{poly}} M$) if there is a polynomial-time-computable function $f$ that takes as input a binary string $s$ and outputs a binary string $f(s)$ so that $s \in L$ iff $f(s) \in M$.

As you saw in Comp202, If $L_1 \xrightarrow{\text{poly}} L_2$ and $L_2 \xrightarrow{\text{poly}} L_3$ then $L_1 \xrightarrow{\text{poly}} L_3$. 
Polynomial-time reducibility

We say that a language $L$, defining some decision problem, is **polynomial-time reducible** to a language $M$ (written $L \xrightarrow{\text{poly}} M$) if there is a polynomial-time-computable function $f$ that takes as input a binary string $s$ and outputs a binary string $f(s)$ so that $s \in L$ iff $f(s) \in M$.

As you saw in Comp202, If $L_1 \xrightarrow{\text{poly}} L_2$ and $L_2 \xrightarrow{\text{poly}} L_3$ then $L_1 \xrightarrow{\text{poly}} L_3$. 
NP-completeness

We say that a language $M$, defining some decision problem, is **NP-hard** if every language $L \in \text{NP}$ is polynomial-time reducible to $M$.

We say that a language $M$ is **NP-complete** if $M$ is in NP and $M$ is NP-hard.
The Cook-Levin Theorem is that the problem SAT is NP-complete.

**Name:** SAT  
**Instance:** A Boolean formula $F$  
**Question:** Does $F$ have a satisfying assignment?

Recall that a **Boolean formula** is an expression like

$$(x_{25} \land x_{12}) \lor \neg(x_{70} \lor (\neg x_3 \land x_{34})))$$

made up of the constants *true* and *false*, propositional variables $x_i$, parentheses and the connectives $\land$, $\lor$, $\neg$, $\Rightarrow$, $\Leftrightarrow$. An assignment of the *truth-values* true and false to the variables is **satisfying** if it makes the formula evaluate to true.
The Cook-Levin Theorem is that the problem SAT is NP-complete.

**Name:** SAT  
**Instance:** A Boolean formula $F$  
**Question:** Does $F$ have a satisfying assignment?

Recall that a **Boolean formula** is an expression like

$$(x_{25} \land x_{12}) \lor \neg(x_{70} \lor (\neg x_{3} \land x_{34}))$$

made up of the constants *true* and *false*, propositional variables $x_i$, parentheses and the connectives $\land$, $\lor$, $\neg$, $\Rightarrow$, $\Leftrightarrow$. An assignment of the **truth-values** true and false to the variables is **satisfying** if it makes the formula evaluate to true.
The Cook-Levin Theorem is that the problem SAT is NP-complete.

**Name: SAT**  
**Instance:** A Boolean formula $F$  
**Question:** Does $F$ have a satisfying assignment?

Recall that a **Boolean formula** is an expression like

$$(x_{25} \land x_{12}) \lor \neg (\neg x_{70} \lor (\neg x_{3} \land x_{34}))$$

made up of the constants *true* and *false*, propositional variables $x_i$, parentheses and the connectives $\land$, $\lor$, $\neg$, $\Rightarrow$, $\Leftrightarrow$. An assignment of the *truth-values* true and false to the variables is **satisfying** if it makes the formula evaluate to true.
If a language $M$ is NP-hard and $M \xrightarrow{\text{poly}} L$ then $L$ is NP-hard.

Thus, to show that a language $L$ is NP-complete, we do the following.

1. Show that $L$ is in NP, and
2. Take some NP-hard problem $M$ and find a polynomial-time reduction from $M$ to $L$.

Make sure you don’t go the wrong direction!

We will now show that some problems are NP-complete.
If a language $M$ is NP-hard and $M \xrightarrow{\text{poly}} L$ then $L$ is NP-hard.

Thus, to show that a language $L$ is NP-complete, we do the following.

1. Show that $L$ is in NP, and
2. Take some NP-hard problem $M$ and find a polynomial-time reduction from $M$ to $L$.

Make sure you don’t go the wrong direction!

We will now show that some problems are NP-complete.
If a language $M$ is NP-hard and $M \xrightarrow{\text{poly}} L$ then $L$ is NP-hard.

Thus, to show that a language $L$ is NP-complete, we do the following.

1. Show that $L$ is in NP, and

2. Take some NP-hard problem $M$ and find a polynomial-time reduction from $M$ to $L$.

Make sure you don’t go the wrong direction!

We will now show that some problems are NP-complete.
3-Conjunctive Normal Form Satisfiability (3-CNF)

- **Input:** A boolean formula $F$ expressed as an AND of clauses in which each clause is the OR of exactly three distinct literals.

- **Output:** Is there an assignment of boolean values to the variables which causes $F$ to evaluate to *true*?

$$F = (\neg y_1 \vee \neg x_1 \vee y_1) \land (\neg y_1 \vee x_1 \vee \neg y_2) \land (\neg y_1 \vee x_1 \vee y_2)$$

Note that $y_1$ and $\neg y_1$ are distinct literals.
3-Conjunctive Normal Form Satisfiability (3-CNF)

- **Input:** A boolean formula $F$ expressed as an AND of clauses in which each clause is the OR of exactly three distinct literals.

- **Output:** Is there an assignment of boolean values to the variables which causes $F$ to evaluate to *true*?

$$F = (\neg y_1 \lor \neg x_1 \lor y_1) \land (\neg y_1 \lor x_1 \lor \neg y_2) \land (\neg y_1 \lor x_1 \lor y_2)$$

Note that $y_1$ and $\neg y_1$ are distinct literals.
3-Conjunctive Normal Form Satisfiability (3-CNF)

- **Input:** A boolean formula $F$ expressed as an AND of clauses in which each clause is the OR of exactly three distinct literals.

- **Output:** Is there an assignment of boolean values to the variables which causes $F$ to evaluate to true?

$$F = (\neg y_1 \lor \neg x_1 \lor y_1) \land (\neg y_1 \lor x_1 \lor \neg y_2) \land (\neg y_1 \lor x_1 \lor y_2)$$

Note that $y_1$ and $\neg y_1$ are distinct literals.
3-CNF is in NP.

The non-deterministic algorithm “guesses” a satisfying assignment then checks in polynomial time that the guess is a satisfying assignment for $F$. 
3-CNF is in NP.

The non-deterministic algorithm “guesses” a satisfying assignment then checks in polynomial time that the guess is a satisfying assignment for $F$. 
To show that 3-CNF is NP-complete, we take some NP-complete problem, say SAT, and find a polynomial-time reduction from SAT to 3-CNF.

We will show that there is a polynomial-time computable function $f$ that takes as input an input $F$ of SAT and outputs an input $f(F)$ of 3-CNF so that $f(F)$ is a “yes” instance of 3-CNF iff $F$ is a “yes” instance of SAT.
To show that 3-CNF is NP-complete, we take some NP-complete problem, say SAT, and find a polynomial-time reduction from SAT to 3-CNF.

We will show that there is a polynomial-time computable function $f$ that takes as input an input $F$ of SAT and outputs an input $f(F)$ of 3-CNF so that $f(F)$ is a “yes” instance of 3-CNF iff $F$ is a “yes” instance of SAT.
The transformation from $F$ to $f(F)$

**Step 1:** Transform $F$ into a formula $F'$ which is the AND of clauses, each of which has at most 3 literals.

First, parse $F$

$$F = x_1 \land (\neg x_2 \Leftrightarrow (x_3 \lor x_4 \lor x_5)) \land \neg x_4$$
\[ F = x_1 \land (\neg x_2 \iff (x_3 \lor x_4 \lor x_5)) \land \neg x_4 \]
Now use the associativity of $\land$ and $\lor$ to form an equivalent tree in which every node has at most 2 children.
Now label the parent-edge out of every internal node (on the previous slide) by a new variable.

Rewrite the formula as an equation.

\[ F' = y_1 \land (y_1 \Leftrightarrow (x_1 \land y_2)) \land (y_2 \Leftrightarrow (y_3 \land \neg x_4)) \land (y_3 \Leftrightarrow (\neg x_2 \Leftrightarrow y_4)) \land (y_4 \Leftrightarrow (x_3 \lor y_5)) \land (y_5 \Leftrightarrow (x_4 \lor x_5)) \]

We have now transformed \( F \) into a formula \( F' \) which is the AND of clauses, each of which has at most 3 literals. \( F' \) is satisfiable iff \( F \) is.
Now label the parent-edge out of every internal node (on the previous slide) by a new variable.

Rewrite the formula as an equation.

\[ F' = y_1 \land (y_1 \leftrightarrow (x_1 \land y_2)) \land (y_2 \leftrightarrow (y_3 \land \neg x_4)) \land (y_3 \leftrightarrow (\neg x_2 \leftrightarrow y_4)) \land (y_4 \leftrightarrow (x_3 \lor y_5)) \land (y_5 \leftrightarrow (x_4 \lor x_5)) \]

We have now transformed \( F \) into a formula \( F' \) which is the AND of clauses, each of which has at most 3 literals. \( F' \) is satisfiable iff \( F \) is.
Now label the parent-edge out of every internal node (on the previous slide) by a new variable.

Rewrite the formula as an equation.

\[ F' = y_1 \land (y_1 \iff (x_1 \land y_2)) \land (y_2 \iff (y_3 \land \neg x_4)) \land (y_3 \iff (\neg x_2 \iff y_4)) \land (y_4 \iff (x_3 \lor y_5)) \land (y_5 \iff (x_4 \lor x_5)) \]

We have now transformed \( F \) into a formula \( F' \) which is the AND of clauses, each of which has at most 3 literals. \( F' \) is satisfiable iff \( F \) is.
Note that the transformation from $F$ to $F'$ can be implemented in polynomial time. Each connective in $F$ introduces at most one variable and one clause to $F'$ to $|F'|$ is at most a polynomial in $|F|$. 
The transformation from \( F \) to \( f(F) \)

**Step 2:** Transform \( F' \) into a formula \( F'' \) which is the AND of clauses, each of which is the OR of at most 3 literals.

We will use a truth table to transform each clause of \( F' \) to the AND of at most 8 clauses which are algebraically equivalent.
Step 2: Transform $F'$ into a formula $F''$ which is the AND of clauses, each of which is the OR of at most 3 literals.

We will use a truth table to transform each clause of $F'$ to the AND of at most 8 clauses which are algebraically equivalent.
For example, take this clause of $F'$: $y_1 \Leftrightarrow (x_1 \land y_2)$

<table>
<thead>
<tr>
<th>$y_1$</th>
<th>$x_1$</th>
<th>$y_2$</th>
<th>result</th>
</tr>
</thead>
<tbody>
<tr>
<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>
</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>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</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>

The first 0 in the result column of the truth table says you can’t have $y_1x_1\neg y_2$ so insert the first clause below.

\[
(\neg y_1 \lor \neg x_1 \lor y_2) \land (\neg y_1 \lor x_1 \lor \neg y_2) \land \\
(\neg y_1 \lor x_1 \lor y_2) \land (y_1 \lor \neg x_1 \lor \neg y_2)
\]
Having done steps 1 and 2 we have now shown how to transform \( F \) into a formula \( F'' \) which is the AND of clauses, each of which is the OR of at most 3 literals.

\( F'' \) is satisfiable iff \( F \) is.

The transformation can be accomplished in polynomial time.
Having done steps 1 and 2 we have now shown how to transform $F$ into a formula $F''$ which is the AND of clauses, each of which is the OR of at most 3 literals.

$F''$ is satisfiable iff $F$ is.

The transformation can be accomplished in polynomial time.
The transformation from $F$ to $f(F)$

Step 3: Transform $F''$ into a formula $F'''$ which is the AND of clauses, each of which is the OR of exactly 3 literals. Let $f(F) = F'''$.

Transform a 2-literal clause like this, using a new variable $p$.

$$(x \lor y) \Rightarrow (x \lor y \lor p) \land (x \lor y \lor \lnot p)$$

Transform a 1-literal clause like this, using new variables $p$ and $q$.

$$x \Rightarrow (x \lor p \lor q) \land (x \lor p \lor \lnot q) \land (x \lor \lnot p \lor q) \land (x \lor \lnot p \lor \lnot q)$$
We have shown that there is a polynomial-time computable function $f$ that takes as input an input $F$ of SAT and outputs an input $f(F)$ of 3-CNF so that $f(F)$ is a “yes” instance of 3-CNF iff $F$ is a “yes” instance of SAT.

This is a polynomial-time reduction from SAT to 3-CNF.

Since SAT is NP-hard, we conclude that 3-CNF is NP-hard.

We already showed that 3-CNF is in NP, so we conclude that 3-CNF is NP-complete.
We have shown that there is a polynomial-time computable function $f$ that takes as input an input $F$ of SAT and outputs an input $f(F)$ of 3-CNF so that $f(F)$ is a “yes” instance of 3-CNF iff $F$ is a “yes” instance of SAT.

This is a polynomial-time reduction from SAT to 3-CNF.

Since SAT is NP-hard, we conclude that 3-CNF is NP-hard.

We already showed that 3-CNF is in NP, so we conclude that 3-CNF is NP-complete.
We have shown that there is a polynomial-time computable function $f$ that takes as input an input $F$ of SAT and outputs an input $f(F)$ of 3-CNF so that $f(F)$ is a “yes” instance of 3-CNF iff $F$ is a “yes” instance of SAT.

This is a polynomial-time reduction from SAT to 3-CNF.

Since SAT is NP-hard, we conclude that 3-CNF is NP-hard.

We already showed that 3-CNF is in NP, so we conclude that 3-CNF is NP-complete.
We have shown that there is a polynomial-time computable function $f$ that takes as input an input $F$ of SAT and outputs an input $f(F)$ of 3-CNF so that $f(F)$ is a “yes” instance of 3-CNF iff $F$ is a “yes” instance of SAT.

This is a polynomial-time reduction from SAT to 3-CNF.

Since SAT is NP-hard, we conclude that 3-CNF is NP-hard.

We already showed that 3-CNF is in NP, so we conclude that 3-CNF is NP-complete.
Another computational problem

Clique

- **Input:** An undirected graph $G$ and an integer $j$
- **Output:** Is there a set of $j$ vertices of $G$, each pair of which is connected by an edge?
Clique is in NP

The non-deterministic algorithm “guesses” a set of $j$ vertices then checks in polynomial time to see whether each pair is connected by an edge.
3-CNF $\xrightarrow{\text{poly}}$ Clique

Let $F$ be an input to 3-CNF. We show how to transform it to into an input $(G, j)$ of Clique such that $G$ has a $j$-clique iff $F$ is satisfiable.

Let $j =$ number of clauses in $F$. For every clause $C_r = (x_1 \lor x_2 \lor \neg x_3)$, introduce vertices $x_{1,r}$, $x_{2,r}$ and $\neg x_{3,r}$. Introduce edges between vertices in different clauses, unless they are the negation of each other. For example...

$$(x_1 \lor x_2 \lor \neg x_3) \land (\neg x_1 \lor x_2 \lor x_4)$$

$\neg x_{3,1}$

$x_{4,2}$

$x_{2,1}$

$x_{2,2}$

$x_{1,1}$

$\neg x_{1,2}$
3-CNF poly $\rightarrow$ Clique

Let $F$ be an input to 3-CNF. We show how to transform it to into an input $(G, j)$ of Clique such that $G$ has a $j$-clique iff $F$ is satisfiable.

Let $j =$ number of clauses in $F$. For every clause $C_r = (x_1 \lor x_2 \lor \neg x_3)$, introduce vertices $x_{1,r}, x_{2,r}$ and $\neg x_{3,r}$.

Introduce edges between vertices in different clauses, unless they are the negation of each other. For example...

$$(x_1 \lor x_2 \lor \neg x_3) \land (\neg x_1 \lor x_2 \lor x_4)$$

\[
\begin{align*}
\neg x_{3,1} & \quad \neg x_{1,2} \\
\quad x_{4,2} & \\
\quad x_{2,2} & \\
\quad x_{2,1} & \\
\quad x_{1,1} &
\end{align*}
\]
Let $F$ be an input to 3-CNF. We show how to transform it to into an input $(G, j)$ of Clique such that $G$ has a $j$-clique iff $F$ is satisfiable.

Let $j = \text{number of clauses in } F$. For every clause $C_r = (x_1 \lor x_2 \lor \neg x_3)$, introduce vertices $x_{1,r}$, $x_{2,r}$ and $\neg x_{3,r}$. Introduce edges \textit{between} vertices in different clauses, \textbf{unless} they are the negation of each other. For example...

\[
(x_1 \lor x_2 \lor \neg x_3) \land (\neg x_1 \lor x_2 \lor x_4)
\]
Let $F$ be an input to 3-CNF. We show how to transform it to into an input $(G, j)$ of Clique such that $G$ has a $j$-clique iff $F$ is satisfiable.

Let $j = \text{number of clauses in } F$. For every clause $C_r = (x_1 \lor x_2 \lor \neg x_3)$, introduce vertices $x_{1,r}$, $x_{2,r}$ and $\neg x_{3,r}$. Introduce edges between vertices in different clauses, unless they are the negation of each other. For example...

$$(x_1 \lor x_2 \lor \neg x_3) \land (\neg x_1 \lor x_2 \lor x_4)$$
Suppose we have a satisfying assignment. We can choose one “true” literal from each of the $j$ clauses, and that gives us a clique.

Similarly, we can turn a clique into a satisfying assignment.

Note that the transformation takes polynomial time.

We have shown that Clique is NP-complete.
Suppose we have a satisfying assignment. We can choose one “true” literal from each of the $j$ clauses, and that gives us a clique.

Similarly, we can turn a clique into a satisfying assignment.

Note that the transformation takes polynomial time.

We have shown that Clique is NP-complete.
Suppose we have a satisfying assignment. We can choose one “true” literal from each of the $j$ clauses, and that gives us a clique.

Similarly, we can turn a clique into a satisfying assignment.

Note that the transformation takes polynomial time.

We have shown that Clique is NP-complete.
Suppose we have a satisfying assignment. We can choose one “true” literal from each of the $j$ clauses, and that gives us a clique.

Similarly, we can turn a clique into a satisfying assignment.

Note that the transformation takes polynomial time.

We have shown that Clique is NP-complete.
Another computational problem

**Vertex Cover**

- **Input:** An undirected graph $G$ and an integer $k$
- **Output:** Is there a set $U$ of $k$ vertices of $G$ such that for every edge $(u, v)$ of $G$, at least one of $u$ and $v$ is in $U$?
Vertex Cover is in NP

The non-deterministic algorithm “guesses” a set of $k$ vertices then checks in polynomial time to see whether every edge is covered.
Let $G = (V, E)$ and $j$ be an input to clique. We show how to transform it to into an input $(G', k)$ of Vertex Cover such that $G'$ has a vertex cover of size $k$ iff $G$ has a clique of size $j$.

Method: Let $\overline{E} = \{(u, v) \mid (u, v) \not\in E\}$ and $G' = (V, \overline{E})$ and $k = |V| - j$.

If $U$ is a clique then $V - U$ covers all non-edges (and vice-versa).

This is a polynomial-time transformation, so we have shown that vertex cover is NP-complete.
Let $G = (V, E)$ and $j$ be an input to clique. We show how to transform it to into an input $(G', k)$ of Vertex Cover such that $G'$ has a vertex cover of size $k$ iff $G$ has a clique of size $j$.

Method: Let $\overline{E} = \{(u, v) \mid (u, v) \not\in E\}$ and $G' = (V, \overline{E})$ and $k = |V| - j$.

If $U$ is a clique then $V - U$ covers all non-edges (and vice-versa).

This is a polynomial-time transformation, so we have shown that vertex cover is NP-complete.
Clique $\xrightarrow{\text{poly}}$ Vertex Cover

Let $G = (V, E)$ and $j$ be an input to clique. We show how to transform it to into an input $(G', k)$ of Vertex Cover such that $G'$ has a vertex cover of size $k$ iff $G$ has a clique of size $j$.

Method: Let $\overline{E} = \{ (u, v) \mid (u, v) \notin E \}$ and $G' = (V, \overline{E})$ and $k = |V| - j$.

If $U$ is a clique then $V - U$ covers all non-edges (and vice-versa).

This is a polynomial-time transformation, so we have shown that vertex cover is NP-complete.
Let $G = (V, E)$ and $j$ be an input to clique. We show how to transform it to into an input $(G', k)$ of Vertex Cover such that $G'$ has a vertex cover of size $k$ iff $G$ has a clique of size $j$.

Method: Let $\overline{E} = \{(u, v) \mid (u, v) \notin E\}$ and $G' = (V, \overline{E})$ and $k = |V| - j$.

If $U$ is a clique then $V - U$ covers all non-edges (and vice-versa).

This is a polynomial-time transformation, so we have shown that vertex cover is NP-complete.
One last computational problem (this one is pretty tricky!)

Subset Sum

- **Input:** A set $S$ of non-negative integers and a non-negative integer $t$.
- **Output:** Is there a subset of $S$ whose elements sum to $t$?

Example: $S = \{1, 3, 5\}$. What about $t = 4$? What about $t = 2$?
Subset Sum is in NP

The non-deterministic algorithm “guesses” the subset and checks that its elements sum to \( t \).
Let $G = (V, E)$ and $k$ be an input to vertex cover. We show how to transform it to an input $S, t$ of subset sum such that $G$ has a vertex cover of size $k$ iff $S$ has a subset that sums to $t$.

Notation: Let $V = \{v_0, \ldots, v_{n-1}\}$. Let $E = \{e_0, \ldots, e_{m-1}\}$.
The (polynomial-time) transformation:

For \( i \leftarrow 0 \) to \( n - 1 \)
\[
x_i \leftarrow 4^m
\]
For \( j \leftarrow 0 \) to \( m - 1 \)
\[
\text{If } e_j \text{ is incident on } v_i
\]
\[
x_i \leftarrow x_i + 4^j
\]
For \( j \leftarrow 0 \) to \( m - 1 \)
\[
y_j \leftarrow 4^j
\]
\( S \leftarrow \{x_0, \ldots, x_{n-1}, y_0, \ldots, y_{m-1}\} \)
\( t \leftarrow k4^m + \sum_{j=0}^{m-1} 2 \cdot 4^j \)
Return \( S \) and \( t \)
We claim that if $G$ has a size-$k$ vertex cover then $S$ has a subset that sums to $t$.

- Start with a size-$k$ vertex cover.
- Let $S'$ contain $x_i$s for vertices in the cover and $y_j$s for edges incident once on cover.
- Sum of $x_i$s in $S'$ is $k4^m$.
- Edge incident twice on cover contributes $2 \cdot 4^i$ to $x$'s
- Edge incident once on cover contributes $4^i$ to $x$'s and $4^i$ to $y$'s.
- Elements in $S'$ sum to $t$. 
We claim that if $S$ has a subset that sums to $t$ then $G$ has a size-$k$ vertex cover.

- Start with $S'$ which sums to $t$.
- Each $e^i$ contributes at most $2 \cdot 4^i$ to $x$s and $4^i$ to $y$s.
- The $e^i$s do not contribute to the $k4^m$ in $t$.
- $S'$ has $k$ $x_i$s.
- These $k$ vertices are a vertex cover because each $e^j$ contributes exactly $2 \cdot 4^j$ to $t$ but only $4^j$ of this can come from $y_j$ so it must be adjacent to one of the vertices in $S'$.
We have shown that Subset Sum is NP-complete.