• We will introduce the concept of polynomial time reduction to convert one problem to another.

• We will show that 3COL is polynomial time reducible to SAT.

• In fact, any NP problem is polynomial time reducible to SAT. This is Cook’s Theorem.

• Another way of saying this is that SAT is “NP-complete”. NP-complete problems are the key to deciding whether P=NP.
Polynomial time reduction

If we can reduce one problem A to another problem B, then any polynomial time algorithm for B will give us a polynomial time algorithm for A, won’t it?

It depends on how fast the reduction can be computed. If the translation algorithm runs in exponential time, the above is clearly not true. We will therefore introduce the concept of polynomial time reduction.
Polynomial time computable functions

Definition

A function $f : \Sigma^* \to \Delta^*$ is polynomial time computable if there exists a polynomial time deterministic Turing machine that halts on each input $w \in \Sigma^*$ with just $f(w)$ on its tape.
Polynomial time reduction

Let A and B be two decision problems.

Definition

A polynomial time reduction from A to B is a function $f$ which maps instances of A to instances of B, such that:

- $f$ is computable in polynomial time;
- for any instance $x$ of A, $x$ is a “yes” instance of A if and only if $f(x)$ is a “yes” instance of B.

We write $A \leq_P B$ if a polynomial time reduction from A to B exists.
Polynomial time reduction

If $f$ is a polynomial time reduction from A to B, and we can find a polynomial time algorithm for B, then this automatically gives us a polynomial time algorithm for A:

Given an instance $x$ of A, compute $f(x)$ and run the algorithm for B on it. This will tell us whether $x$ is a “yes” instance of A. The time taken to do all of this is polynomial.

**Theorem**

If $A \leq_P B$ and $B \in P$, then $A \in P$.

**Corollary**

If $A \leq_P B$ and $A \not\in P$, then $B \not\in P$. 
Reduction of 3COL to SAT

• As a classic example of polynomial time reduction, let us reduce 3COL to SAT.

• Recall that 3COL and SAT are both problems which we don’t have a polynomial time algorithm for.

• If we can do this reduction, then a polynomial time algorithm for SAT gives us a polynomial time algorithm for 3COL.
Reduction of 3COL to SAT

So our problem is this: given a graph $G$, find a transformation of $G$ to a boolean expression, such that the expression is satisfiable precisely when the graph is 3-colourable. The transformation must be computable in polynomial time.
Reduction of 3COL to SAT

- An assignment of 0’s and 1’s to the boolean variables in our expression must correspond to an assignment of colours to the vertices.

- So for each vertex $v_1, \ldots, v_m$ of $G$, we have three boolean variables, for a total of $3m$ variables:

  $r_i : 1$ precisely when $v_i$ is RED.

  $g_i : 1$ precisely when $v_i$ is GREEN.

  $b_i : 1$ precisely when $v_i$ is BLUE.
Reduction of 3COL to SAT

The logical relationships between the boolean variables must correspond to the colouring constraints on the graph.

These are of three kinds.

1. For each vertex $v_i$, the expression

$$ (r_i \lor g_i \lor b_i) $$

must be true (each vertex has at least one colour).

2. For each vertex $v_i$, the expression

$$ \neg(r_i \land g_i) \land \neg(r_i \land b_i) \land \neg(g_i \land b_i) $$

must be true (each vertex has at most one colour).
For each edge $e_{i,j}$ which is present in $G$, we need constraints:

$$
\neg(r_i \land r_j) \land \neg(g_i \land g_j) \land \neg(b_i \land b_j)
$$

(adjacent vertices cannot have the same colour).

We now combine all these formulae together into another boolean expression $f(G)$ by “and”ing them.

The expression $f(G)$ is satisfiable precisely when there exists an assignment of exactly one colour to each graph vertex, such that adjacent vertices have different colours.
An example

\[(r_1 \lor g_1 \lor b_1) \land (r_2 \lor g_2 \lor b_2) \land (r_3 \lor g_3 \lor b_3)\]

\[\land \neg (r_1 \land g_1) \land \neg (r_1 \land b_1) \land \neg (g_1 \land b_1)\]

\[\land \neg (r_2 \land g_2) \land \neg (r_2 \land b_2) \land \neg (g_2 \land b_2)\]

\[\land \neg (r_3 \land g_3) \land \neg (r_3 \land b_3) \land \neg (g_3 \land b_3)\]

\[\land \neg (r_1 \land r_2) \land \neg (g_1 \land g_2) \land \neg (b_1 \land b_2)\]

\[\land \neg (r_2 \land r_3) \land \neg (g_2 \land g_3) \land \neg (b_2 \land b_3)\]
So we have the required mapping $f$ from graphs to boolean expressions.

Can $f$ be computed in polynomial time?

The number of characters in the output expression $f(G)$ equals:

$$7m + 20m + 20l + (2m + l - 1),$$

where $m$=#vertices, $l$=#edges. This is certainly polynomial in the size of the input. Moreover, one pass through the input suffices to compute $f$. So $f$ can be computed in polynomial time.
Reduction of 3COL to SAT

- We have found a polynomial time reduction of 3COL to SAT.
- If we subsequently find a polynomial time algorithm for SAT, this gives us one for 3COL as well.
- Conversely, if we can prove that there is no polynomial time algorithm for 3COL, then there can’t be one for SAT either.
What is meant by a “hard” problem?

The idea is that if we can reduce all NP problems to a problem in polynomial time, then that problem must be “hard”, because a polynomial time algorithm for it leads to polynomial time algorithms for all NP problems.
**The Cook-Levin Theorem**

The **Cook-Levin Theorem** says that not only can **3COL** be reduced to **SAT**, but every other problem in **NP** can be reduced to **SAT** as well!

So **SAT** is at least as hard as any **NP** problem.

Moreover, if we can find a polynomial time algorithm for **SAT**, then we can find one for *any* **NP** problem, i.e.

\[ P = NP. \]

<table>
<thead>
<tr>
<th>Theorem (Cook-Levin)</th>
</tr>
</thead>
<tbody>
<tr>
<td>There is a polynomial time reduction from any <strong>NP</strong> problem to <strong>SAT</strong>.</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Corollary</th>
</tr>
</thead>
<tbody>
<tr>
<td>If <strong>SAT</strong> (\in\ P) then (P = NP).</td>
</tr>
</tbody>
</table>
Importance of the Cook-Levin theorem

- If we ever find a polynomial time algorithm for SAT, we will know that there is one for every NP problem, including nasty ones like 3COL and TSP(D).

- On the other hand, probably $P$ is not equal to $NP$. In this case SAT can not be in $P$. So it is worthwhile making a serious attempt to prove it and finally answer the question “Is $P=NP$?”.
A word about the proof ...

To prove the Cook-Levin theorem, we have to find a polynomial time reduction from every NP problem $X$ to $\text{SAT}$.

\[ X \rightarrow \text{SAT} \]

instance $\rightarrow$ boolean expression

“yes” $\rightarrow$ “yes” (satisfiable)

“no” $\rightarrow$ “no” (not satisfiable)
A word about the proof ... (not examinable!)

The proof proceeds as follows:

- There exists a non-deterministic TM which solves the problem $X$ in polynomial time.

- For any input $I$ to the TM, encode the execution of the machine as a boolean expression $\varphi_I$ which is satisfiable if and only if the machine accepts the input:
  
  - $\varphi_I$ contains some boolean variables which encode the non-deterministic choices the machine has in its execution.
  
  - $\varphi_I$ is satisfiable if and only if a combination of choices exists that leads to $I$ being accepted, i.e. if and only if $I$ is a “yes” instance of the problem.
  
  - $\varphi_I$ can be computed in polynomial time.

(see pp. 281-286 of Sipser for details)
NP-hard and NP-complete problems

So SAT has the property that we can reduce any NP problem to it in polynomial time.

**Definition**

A decision problem X is called **NP-hard** if every decision problem Y in NP is polynomial time reducible to X.

If X is also in NP, it is called **NP-complete**.

If any NP-hard or NP-complete problem is in P, then P=NP.
Cook and Levin were the first to show that there exists an NP-complete problem (SAT).

To prove that another problem, call it X, is NP-complete, we only have to show it is in NP and then find a polynomial time reduction to this problem from SAT (or any known NP-complete problem).

Then there must be a polynomial time reduction from any NP problem to X (via SAT).
NP-hard and NP-complete problems

If $X$ is an NP-complete problem and there exists a polynomial time reduction from $X$ to some problem $Y$, then $Y$ is NP-hard.

Using this rule, we can now find lots of NP-complete problems, including \textsc{3COL} and \textsc{TSP}(D).

So if we find a polynomial time algorithm for any of these, then there must be a polynomial time algorithm for every NP problem, i.e.

$$P = NP$$
The big picture

- \( P \)
- \( \text{NP} \)
- \( \text{NP-complete} \)
- \( \text{NP-hard} \)

increasing difficulty
Or it might just look like this:

Increasing difficulty
Suppose we have a problem $X$ which is $O(n^2)$, and a polynomial time reduction of 3COL to $X$. What can we deduce?

$X$ must be in P. But, the polynomial time reduction of 3COL to $X$ tells us that $X$ is NP-hard. Since $X$ is in P, which is in NP, $X$ is also NP-complete, and so $P = NP$.

Alternatively, we can see that the reduction gives us a polynomial time algorithm for 3COL, and therefore for any NP problem.
Suppose we have an NP problem $X$ and an algorithm for $X$ which is $\Theta(2^n)$. What can we deduce?

Not much. There could be a faster (polynomial) algorithm for $X$; all we know is that $X$ is $O(2^n)$.

If we knew that $X$ itself was $\Theta(2^n)$, then there could be no algorithm for $X$ faster than $\Theta(2^n)$, so no polynomial algorithm for $X$. This would prove $P \neq NP$. 
Recall: given a decision problem, we can think of the “yes” instances as forming a language. Solving the decision problem amounts to deciding this language.

Conversely, each language $L \subseteq \Sigma^*$ can be associated a decision problem:

- the inputs (problem instances) are the strings over $\Sigma^*$,
- the ”yes” instances are precisely those strings which belong to $L$.

**Definition**

A language $L$ is said to be in P (in NP) if the associated decision problem is in P (respectively NP).
Summary

- (N)P is the class of problems that can be solved in polynomial time by (non-)deterministic machines.

- We don’t know whether P=NP. NP problems which may or may not be in P include SAT, 3COL and TSP(D).

- Polynomial time reduction can help us work out which are the key problems to look at in order to answer this question.

- The Cook-Levin Theorem states that we can reduce any NP-problem to SAT. The class of problems with this property is called NP-hard.

- NP-complete := NP-hard ∩ NP

- If we find a polynomial time algorithm for a single NP-complete problem, we get one for all NP problems.
Beyond the class NP:

- the class PSPACE
- the class EXPTIME
- ...