Remember that if $L_1$ reduces to $L_2$, then we can decide membership in $L_1$ using an algorithm (TM) for membership in $L_2$.

In particular, if $L_2$ is Turing-decidable, and $L_1 \leq L_2$, then $L_1$ is Turing-decidable.

We did this by using a computable function $f$ such that

$$x \in L_1 \iff f(x) \in L_2.$$ 

Now we want to replace “Turing-decidable” with “decidable in polynomial time”.

Then we have to replace “computable function $f$” with a suitable related notion.
Polynomial-time reductions

What we want is that if $L_1$ reduces to $L_2$ via a polynomial-time reduction, then a polynomial-time algorithm for $L_2$ gives a polynomial-time algorithm for $L_1$.

The obvious analogy is to demand that $f$ itself be polynomial-time computable.

We say $L_1$ reduces in polynomial time to $L_2$, and write $L_1 \leq_P L_2$, if there is a polynomial-time computable function $f$ such that

$$x \in L_1 \iff f(x) \in L_2.$$
**Theorem.** If $L_2 \in P$ and $L_1 \leq_P L_2$, then $L_1 \in P$.

**Proof.** By picture. Let $M_2$ be a polynomial-time TM for $L_2$, and construct a polynomial-time TM $M_1$ for $L_1$ from it, using the reduction $f$:

(We see that $L_1 \in P$ using exactly the same reasoning as when we showed that the composition of two polynomial-time computable functions is computable in polynomial time.)
A similar result for NP

**Theorem.** If $L_2 \in \text{NP}$ and $L_1 \leq_P L_2$, then $L_1 \in \text{NP}$.

*Proof.* Exactly the same proof as for P, except now $M_1$ and $M_2$ are nondeterministic Turing machines.

*Exercise.* Find another proof of this result using the alternate definition of NP.
Theorem. If $A \leq_P B$ and $B \leq_P C$, then $A \leq_P C$.

Proof. Follows immediately from the result we proved for composition of polynomial-time computable functions in a previous lecture:

If $f$ is the polynomial-time computable function giving the reduction from $A$ to $B$, and $g$ is the polynomial-time computable function giving the reduction from $B$ to $C$, then the composition $g \circ f$ is the polynomial-time computable function giving the reduction from $A$ to $C$. 
The “hardest” Turing-recognizable language

If we go back to ordinary reductions for a second, then there is a way in which we can consider the language of the universal Turing machine $U$

$$A_{DTM} = \{ e(T)e(w) : T \text{ accepts } w \}$$

as the “hardest” Turing-recognizable language. Namely:

**Theorem.** Every Turing-recognizable language $L$ reduces to $A_{DTM}$.

**Proof.** Let $M$ be a DTM recognizing $L$. We need to construct $f$ such that

$$x \in L \iff f(x) \in A_{DTM}.$$ 

How can we do this?

Easy: define $f(x) = e(M)e(x)$.

This is actually very cool, because there is no notion of “hardest regular language”.
We can now define a notion of “hardest” language in NP:

(a) We say that a language $B$ is **NP-hard** if $A \leq_P B$ for all languages $A \in \text{NP}$.

(b) We say that a language $B$ is **NP-complete** if it is NP-hard and $B \in \text{NP}$.

Thus (a) is a *lower bound* on the complexity of $B$; it says it is as least as hard as every single language in NP.

And (b) is a *upper bound* on the complexity of $B$; it says it is in NP.
Now, before we get too excited, remember that these are just *definitions*.

Just because we make a definition doesn’t mean there necessarily exist languages that *satisfy* the definition.

However, it turns out that not only are there NP-complete and NP-hard languages, there are *hundreds* of them known.
Common pitfalls

Don’t make the common mistake of thinking that if a language is in NP, then it must be “hard”.

(NP stands for “nondeterministic polynomial time”, and does not stand for “not polynomial” as some people mistakenly think.)

Remember that $P \subseteq NP$, so there are lots of languages in NP that are “easy”.

It’s the NP-complete languages (NPC) and NP-hard languages (NPH) that are probably hard.

Many NP-complete languages are known (like SUBSUM and HAM), but so far nobody knows a polynomial-time algorithm for any of them.
Although we do not currently know for sure that $P \cap \text{NPC} = \emptyset$, most people believe this is the case.

And we can also prove the following:

**Theorem.** If any single NP-complete language is in $P$, then every language in NP is also in $P$.

**Proof.** Let $A$ be NP-complete and let $B$ be any language in NP. Then by definition of NP-completeness, we know $B \leq_P A$. Suppose $A \in P$. Then by a theorem we proved, we know $B \in P$.

So to prove $P = \text{NP}$, all we have to do is find a *single* NP-complete language that is in $P$.

No one has been able to do this so far.