So far with Turing machines we have mostly used them as language acceptors.

In comes an input \( x \), and the “output” is accept or reject (or sometimes neither).

Now we want to generalize this a bit: we want Turing machines to be able to compute functions \( f \).

The basic idea is that \( w \) should be an input to the TM, and \( f(w) \) should be the result.
Formally, we say that a TM $M$ computes the function $f : \Sigma^* \rightarrow \Delta^*$ if

$$(q_0, \omega) x \xrightarrow{*} (q_{\text{acc}}, \omega) f(x)$$

for all $x \in \Sigma^*$.

Notice that we demand that $M$ halts after computing $f(x)$.
Computable functions

Here is an example of a TM computing a function. Here $\Sigma = \{a\}$ and $f(a^n) = a^{n+1}$.

We say a function is *computable* if there exists a TM computing it.
Relative hardness of problems

People have long wanted to classify computation problems according to how hard they are.

This is a really difficult problem in general, and leads to difficult questions such as “Is P equal to NP?”—still unsolved after many years.

However, there is a relatively simple technique that allows you, in some cases to say things like “problem A is as hard as (or harder than) problem B”.

This is the technique known as **reduction**.
The meaning of reductions

Roughly speaking, if we say “problem $A$ reduces to problem $B$”, this means “if we are given a program for problem $B$, we can use it as a subroutine to solve problem $A$”.

In this sense, problem $B$ is \textit{harder than} (or as hard as) problem $A$, because the \textit{capability of solving} $B$ also allows you to solve $A$ (but not necessarily the other way around).

As an example, consider the case where

\begin{align*}
A &= \text{respond to questions posed in English} \\
B &= \text{respond to questions posed in English or Chinese}
\end{align*}

Then clearly $A$ reduces to $B$, because an algorithm that allows you to do $B$ would also allow you to do $A$.

So $B$ is harder than (or as hard as) $A$. 
As another example consider

\[ A = \text{determine if } n \text{ is a prime number} \]
\[ B = \text{compute the prime factorization of } n \]

Then clearly \( A \) reduces to \( B \), because if you have an algorithm for \( B \) and you want to solve \( A \), all you have to do is run \( B \) on \( n \) and then look at the result to see if it consists of one number raised to the 1 power.

So \( B \) is harder than (or as hard as) \( A \).

In general, if \( A \) reduces to \( B \), we write \( A \leq B \).
A joke about reductions

An engineer and a computer scientist were shown into a kitchen, given an empty pan, and told to boil a pint of water. They both filled the pan with water, put it on the stove, and boiled it.

The next day they were shown into the kitchen again, given a pan full of water, and told to boil a pint of water.

The engineer took the pan, put it on the stove, and boiled it. The computer scientist took the pan and emptied it, and then said “now it reduces to the problem I already solved”.
The particular kinds of reductions we use

There are many different kinds of reductions, but in this course we are going to use one specific kind, called the “mapping” or Karp reduction.

Let \( \Sigma, \Delta \) be alphabets and let \( A \subseteq \Sigma^* \), \( B \subseteq \Delta^* \) be languages. We say that \( A \) reduces to \( B \) if there exists a computable function \( f : \Sigma^* \rightarrow \Delta^* \) such that \( w \in A \) iff \( f(w) \in B \) for all \( w \in \Sigma^* \).

In this case we write \( A \leq_m B \), or just \( A \leq B \).

Note that the condition is an if and only if.
Why is this a reduction?

To understand why this is called a reduction, let’s see that if $A \leq B$, then indeed we could solve $A$, given an algorithm for $B$.

Here’s a picture:

Note that $x \in A$ iff $f(x) \in B$. 
The particular kinds of reductions we use

We call such an $f$ a reduction \textit{from $A$ to $B$}.

Reductions always go \textit{from} the simpler problem \textit{to} the harder problem. That’s why we use the notation $\leq$.

Caution: this is a directional notion! Do not confuse “$A$ reduces to $B$” with “$B$ reduces to $A$”! All kinds of hurt will result from that.
The Karp reduction is named for Richard Karp (b. 1935), a theoretical computer scientist who is one of the founders of the theory of NP-completeness.

Karp won the Turing award (computer science’s highest award) for his contributions to computer science in 1985.

Another famous kind of reduction is the Turing reduction.
One fundamental property of reductions is that they obey the transitive property (just like $\leq$ does for real numbers):

**Theorem.** Let $A$, $B$, $C$ be languages. If $A \leq B$ and $B \leq C$, then $A \leq C$.

**Proof.** If $A \leq B$, then there is a computable function $f$ such that $x \in A$ iff $f(x) \in B$.

If $B \leq C$, then there is a computable function $g$ such that $y \in B$ iff $g(y) \in C$.

To prove $A \leq C$ we need to find a computable function $h$ such that $x \in A$ iff $h(x) \in C$.

What $h$ should we choose?
It’s $h = g \circ f$, which means $h(x) = g(f(x))$.

Why is $h$ computable?

Because if we can run a TM $M_1$ to convert $x$ to $f(x)$ on a tape, and a TM $M_2$ to convert $y$ to $g(y)$ on a tape, then by first running $M_1$ and then $M_2$ we will convert $x$ to $y := f(x)$ to $g(y) = g(f(x)) = h(x)$. 
Now we prove a very simple theorem that has wide-ranging consequences.

**Theorem.** Let $A, B$ be languages. Suppose $A \leq B$. Then

(a) If $B$ is Turing-recognizable then so is $A$.
(b) If $B$ is Turing-decidable then so is $A$.

*Proof of (a).* Since $A \leq B$, that means there is a computable function $f$ such that $x \in A$ iff $f(x) \in B$. Let $f$ be computed by a TM $T_f$.

If $B$ is Turing-recognizable, then $B$ is recognized by some Turing machine $T_B$. We can now use $T_B$ and $T_f$ as subroutines to construct a Turing machine $T_A$ recognizing $A$.

The TM $T_A$ is given on the next slide.
The use of reductions

Here is the construction of $T_A$.

Why does it work? We know $x \in A$ iff $f(x) \in B$.

The Turing machine $T_B$ accepts $y$ iff $y \in B$.

Since $y = f(x)$ we have $T_A$ accepts $x$ iff $x \in A$. 
Proof of (b)

The proof of (b) is exactly the same, except now we take $T_B$ to decide $B$ (instead of just recognizing it), and we produce a decider $T_A$ for $A$. 
Prove that if $A \leq B$, then $\overline{A} \leq \overline{B}$. 
More applications of reductions

**Theorem.**
Suppose $A \leq B$. Then

(a) If $A$ is not Turing-decidable, then $B$ is not Turing-decidable.

(b) If $A$ is not Turing-recognizable, then $B$ is not Turing-recognizable.

**Proof.**

(a) Suppose $A$ is not Turing-decidable. If $B$ were Turing-decidable, then since $A \leq B$ we would have $A$ Turing-decidable, a contradiction.

(b) Suppose $A$ is not Turing-recognizable. If $B$ were Turing-recognizable, then since $A \leq B$ we would have $A$ Turing-recognizable, a contradiction.
The most important use of reductions (for us right now, that is) is to make proofs that a language is not recognizable (or not decidable) much easier.

For example, recall the two languages

\[ A_{\text{DTM}} = \{ e(M)e(w) : M \text{ accepts } w \} \]
\[ \text{HALT} = \{ e(M)e(w) : M \text{ halts on input } w \}. \]

We proved that \( A_{\text{DTM}} \) is not Turing-decidable, and then we used a similar argument to show that \( \text{HALT} \) is not Turing-decidable.
Reductions let us prove more problems undecidable

Instead, we can get the result about \( \text{HALT} \) right away, using reductions and our theorem. We’ll show that \( A_{\text{DTM}} \leq \text{HALT} \). To do that we construct a computable function \( f \) such that \( x \in A_{\text{DTM}} \) iff \( f(x) \in \text{HALT} \).

Here’s how \( f \) is defined:

\[
f(x) = \begin{cases} 
  e(M')e(w), & \text{if } x = e(M)e(w); \\
  x, & \text{otherwise},
\end{cases}
\]

where \( M' \) is exactly like \( M \), except that if \( M \) enters \( q_{\text{rej}} \), then \( M' \) runs forever instead.

Check that \( f \) is computable...

Check that \( x \in A_{\text{DTM}} \) iff \( f(x) \in \text{HALT} \).
Define

\[ \text{Accepts-} \epsilon = \{ e(M) : M \text{ is a DTM that accepts } \epsilon \}. \]

Let us prove that \( \text{Accepts-} \epsilon \) is not decidable.

To do so we will reduce from \( A_{\text{DTM}} \).
We need to create a function $f$ such that $x \in A_{DTM}$ iff $f(x) \in \text{Accepts-}\epsilon$.

Here is how we do it:

$$f(x) = \begin{cases} 
  e(M'), & \text{if } x = e(M)e(w); \\
  e(M_0), & \text{otherwise},
\end{cases}$$

where

- $M'$ is defined as follows: $M'$ erases its input tape, writes $w$ on its tape, and then runs $M$.

- $M_0$ is a TM that rejects every input.

Check that it works!
Let’s define the language

Accepts-Something := \{ e(M) : M is a DTM such that L(M) \neq \emptyset \}.

Let’s show that Accepts-Something is not Turing-decidable.

To do this, we should reduce from some known Turing-undecidable problem.

Let’s reduce from Accepts-\epsilon.
Then we have to construct a computable function $f$ such that $x \in \text{Accepts}-\epsilon$ iff $f(x) \in \text{Accepts-Something}$.

In other words, if $x = e(M)$, we want to construct a TM $M'$ such that $M$ accepts $\epsilon$ iff $M'$ accepts something.

How can we do that?

The easiest way is just to have $M'$ ignore its input and always run $M$ on $\epsilon$, *no matter what its input was*. 
One more example

So here is the construction of $f$:

$$f(x) = \begin{cases} e(M'), & \text{if } x = e(M) \\ x, & \text{otherwise,} \end{cases}$$

where $M'$ is a DTM that first erases its input tape and then calls $M$ on the result.

So by our construction of $M'$ we have $\epsilon \in L(M)$ iff $L(M') = \Sigma^*$. Thus $\epsilon \in L(M)$ iff $M'$ accepts something.

This is what we wanted. This reduction proves that Accepts-Something is not Turing-decidable, either.

By the way, here is an incorrect solution to this problem.

Suppose we define $f(x) = x$ for all $x$. What goes wrong?

(Hint: what if $x = e(M)$, where $M$ accepts the string 0 but no other strings?)