Today's learning goals

Sipser Ch 5.1

• Define and explain core examples of computational problems, include $A^{**}$, $E^{**}$, $EQ^{**}$, $HALT_{TM}$ (for ** either DFA or TM)
• Explain what it means for one problem to reduce to another
• Use reductions to prove undecidability (or decidability)

Announcements:  
Individ HW 6 due tomorrow  
Group HW 6 due Saturday
$A_{TM}$

$A_{TM} = \{ <M,w> \mid M \text{ is a TM and } w \text{ is in } L(M) \}$

Define the TM $N = "\text{On input } <M,w>:"

1. Simulate $M$ on $w$.
2. If $M$ accepts, accept. If $M$ rejects, reject."

$N$ is a Turing machine that recognizes $A_{TM}$.

No Turing machine decides $A_{TM}$.
$A_{TM}$

- Recognizable
- Not decidable

**Fact** (from discussion section): A language is decidable iff it and its complement are both recognizable.

**Corollary 4.23**: The complement of $A_{TM}$ is **unrecognizable**.

**Observation**: Complement of decidable set is also decidable.
<table>
<thead>
<tr>
<th>Decidable</th>
<th>Recognizable (and not decidable)</th>
<th>Co-recognizable (and not decidable)</th>
</tr>
</thead>
<tbody>
<tr>
<td>$A_{DFA}$</td>
<td>$A_{TM}$</td>
<td>$A_{TM}^c$</td>
</tr>
<tr>
<td>$E_{DFA}$</td>
<td></td>
<td></td>
</tr>
<tr>
<td>$EQ_{DFA}$</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Give algorithm!  

Diagonalization
Idea

If problem X is no harder than problem Y
...and if Y is easy
...then X must also be easy
If problem X is no harder than problem Y
...and if X is hard
...then Y must also be hard
Idea

If problem X is no harder than problem Y
...and if Y is \textit{decidable}
...then X must also be \textit{decidable}

If problem X is no harder than problem Y
...and if X is \textit{undecidable}
...then Y must also be \textit{undecidable}
Idea

If problem X is no harder than problem Y
...and if Y is **decidable**
...then X must also be **decidable**

If problem X is no harder than problem Y
...and if X is **undecidable**
...then Y must also be **undecidable**

"Problem X is no harder than problem Y" means
"Given information about Y, we could solve problem X".
The halting problem!

\[ \text{HALT}_\text{TM} = \{ <M,w> | M \text{ is a TM and } M \text{ halts on input } w \} \]

\[ A_{\text{TM}} = \{ <M,w> | M \text{ is a TM and } w \text{ is in } L(M) \} \]

How is \text{HALT}_\text{TM} related to \text{A}_{\text{TM}}?

A. They're the same set.
B. \text{HALT}_\text{TM} is a subset of \text{A}_{\text{TM}}
C. \text{A}_{\text{TM}} is a subset of \text{HALT}_\text{TM}
D. They have the same type of elements but no other relation.
E. I don't know.
The halting problem!

\[ \text{HALT}_{\text{TM}} = \{ <M,w> \mid M \text{ is a TM and } M \text{ halts on input } w \} \]

\[ \text{A}_{\text{TM}} = \{ <M,w> \mid M \text{ is a TM and } w \text{ is in } L(M) \} \]

But subset inclusion doesn't determine difficulty!

\[ \Sigma^* \text{ is "easier" than its subsets} \]
Claim: $A_{TM}$ is no harder than $HALT_{TM}$

In other words: we could use $HALT_{TM}$ to solve $A_{TM}$

Given: Turing machine $M$, string $w$, magic genie for $HALT_{TM}$

Goal: Accept if $w$ is in $L(M)$; Reject if $w$ is not in $L(M)$. 
Claim: $A_{TM}$ is no harder than $HALT_{TM}$

In other words: we could use $HALT_{TM}$ to solve $A_{TM}$

Given: Turing machine $M$, string $w$, magic genie for $HALT_{TM}$

We can ask the magic genie "does a certain TM halt on certain input string?" Genie will **magically** give correct yes/no answer.

We can ask the genie as many of these questions as we'd like, about *any* TM and *any* string.

Goal: Accept if $w$ is in $L(M)$; Reject if $w$ is not in $L(M)$. 
Claim: $A_{TM}$ is no harder than $HALT_{TM}$

In other words: we could use $HALT_{TM}$ to solve $A_{TM}$

Given: Turing machine $M$, string $w$, magic genie for $HALT_{TM}$

"On input $<M, w>$

1. Ask Genie about $M$ and $w$.
2. If Genie says no, then reject; if Genie says yes, run $M$ on $w$.
   a. If this computation accepts, accept.
   b. If this computation rejects, reject."

Goal: Accept if $w$ is in $L(M)$; Reject if $w$ is not in $L(M)$
Proof of correctness: Let $M$ be a TM, $w$ a string

Case (1) $w \in L(M)$. We show that $TM_{Gen\text{e}I\text{m}}$ accepts on input $\langle M, w \rangle$.

In step 1, $alg$ asks $Gen\text{e}i$ about $\langle M, w \rangle$ and it says "yes" because $w \in L(M)$ so $M$ halts on $w$. Then in step 2, $M$ runs on $w$, and will accept by Case assumption so in 2a $Gen\text{e}i_{TM\text{e}M}$ accepts.

Case (2) $w \notin L(M)$. Similar, with subcases for $TM_{Gen\text{e}I\text{m}}$ rejects.
Reduction

"Problem X reduces to problem Y" means
"Problem X is no harder than problem Y" means
"Given a genie for problem Y, we could solve problem X" means
"Given a solution for Y, we have a solution for X"

In the previous example, we used a genie for \( \text{HALT}_{TM} \) to solve \( \text{A}_{TM} \). Thus, \( \text{A}_{TM} \) reduces to \( \text{HALT}_{TM} \).

Which is not true?
A. \( \text{HALT}_{TM} \) reduces to \( \text{A}_{TM} \)
B. \( \Sigma^* \) reduces to \( \text{A}_{TM} \)
C. \( \text{A}_{TM} \) reduces to \( \emptyset \) (the empty set)
D. More than one of the above
E. None of the above
$A_Tm$ reduces to $HALT_{Tm}$

$\exists$

$HALT_{Tm}$ reduces to $A_Tm$

(see podcast for proof on board)
Using reduction to prove undecidability

**Claim:** Problem X is undecidable

**Proof strategy:** Show that $A_{TM}$ reduces to X.

**Alternate Proof strategy:** Show that $HALT_{TM}$ reduces to X.

etc.

In each of these, have access to Genie which can answer questions about X.
Scooping the Loop Snooper
A proof that the Halting Problem is undecidable
Geoffrey K. Pullum
(http://www.lel.ed.ac.uk/~gpullum/loopsnoop.html)

No general procedure for bug checks will do.
Now, I won’t just assert that, I’ll prove it to you.
I will prove that although you might work till you drop,
you cannot tell if computation will stop.

For imagine we have a procedure called P
that for specified input permits you to see
whether specified source code, with all of its faults,
defines a routine that eventually halts.

You feed in your program, with suitable data,
and P gets to work, and a little while later
(in finite compute time) correctly infers
whether infinite looping behavior occurs.

If there will be no looping, then P prints out ‘Good.’
That means work on this input will halt, as it should.
But if it detects an unstoppable loop,
then P reports ‘Bad!’ — which means you’re in the soup.

Well, the truth is that P cannot possibly be,
because if you wrote it and gave it to me,
I could use it to set up a logical bind
that would shatter your reason and scramble your mind.

Here’s the trick that I’ll use — and it’s simple to do.
I’ll define a procedure, which I will call Q,
that will use P’s predictions of halting success
to stir up a terrible logical mess.

For a specified program, say A, one supplies,
the first step of this program called Q I devise
is to find out from P what’s the right thing to say
of the looping behavior of A run on A.

If P’s answer is ‘Bad!,’ Q will suddenly stop.
But otherwise, Q will go back to the top,
and start off again, looping endlessly back,
till the universe dies and turns frozen and black.

And this program called Q wouldn’t stay on the shelf;
I would ask it to forecast its run on itself.
When it reads its own source code, just what will it do?
What’s the looping behavior of Q run on Q?

If P warns of infinite loops, Q will quit;
yet P is supposed to speak truly of it!
And if Q’s going to quit, then P should say ‘Good.’
Which makes Q start to loop! (P denied that it would.)

No matter how P might perform, Q will scoop it:
Q uses P’s output to make P look stupid.
Whatever P says, it cannot predict Q:
P is right when it’s wrong, and is false when it’s true!

I’ve created a paradox, neat as can be —
and simply by using your putative P.
When you posited P you stepped into a snare;
Your assumption has led you right into my lair.

So where can this argument possibly go?
I don’t have to tell you; I’m sure you must know.
A reductio: There cannot possibly be
a procedure that acts like the mythical P.

You can never find general mechanical means
for predicting the acts of computing machines;
it’s something that cannot be done. So we users
must find our own bugs. Our computers are losers!
Next time

Pre-class reading for Wednesday Theorem 5.2