Unsolvable Problems

Part Two
Outline for Today

• Recap from Last Time
  • Where are we, again?

• A Different Perspective on RE
  • What exactly does “recognizability” mean?

• Verifiers
  • A new approach to problem-solving.

• Beyond RE
  • A beautiful example of an impossible problem.
Recap from Last Time
Self-Referential Programs

• **Claim:** Any program can be augmented to include a method called `mySource()` that returns a string representation of its source code.

• **Theorem:** It it possible to build Turing machines that get their own encodings and perform arbitrary computations on them.
What does this program do?

```cpp
bool willAccept(string program, string input) {
    /* ... some implementation ... */
}

int main() {
    string me = mySource();
    string input = getInput();

    if (willAccept(me, input)) {
        reject();
    } else {
        accept();
    }
}
```

What happens if...

... this program accepts its input?  
It rejects the input!

... this program doesn't accept its input?  
It accepts the input!
What does this program do?

```c
bool willHalt(string program, string input) {
    /* ... some implementation ... */
}

int main() {
    string me = mySource();
    string input = getInput();

    if (willHalt(me, input)) {
        while (true) {
            // loop infinitely
        }
    } else {
        accept();
    }
}
```

What happens if...

... this program halts on this input?
It loops on the input!

... this program loops on this input?
It halts on the input!
New Stuff!
So What?

• These problems might not seem all that exciting, so who cares if we can't solve them?

• Turns out, this same line of reasoning can be used to show that some very important problems are impossible to solve.
Secure Voting

• Suppose that you want to make a voting machine for use in an election between two parties.

• Let $\Sigma = \{r, d\}$. A string in $w$ corresponds to a series of votes for the candidates.

• Example: $rrdddrdd$ means “two people voted for $r$, then three people voted for $d$, then one more person voted for $r$, then one more person voted for $d$. ”
Secure Voting

- A voting machine is a program that takes as input a string of $r$'s and $d$'s, then reports whether person $r$ won the election.

- **Question:** Given a TM that someone claims is a secure voting machine, could we automatically check whether it actually is a secure voting machine?
A secure voting machine is a TM $M$ where
$L(M) = \{ w \in \Sigma^* | w \text{ has more } r\text{'s than } d\text{'s } \}$

```c
int main() {
    string input = getInput();
    int numRs = countRsIn(input);
    int numDs = countDsIn(input);
    if (numRs > numDs) accept();
    else reject();
}
```

A (simple) secure voting machine.

```c
int main() {
    string input = getInput();
    if (input[0] == 'r') accept();
    else reject();
}
```

A (simple) insecure voting machine.

```c
int main() {
    string input = getInput();
    int n = input.length();
    while (n > 1) {
        if (n % 2 == 0) n /= 2;
        else n = 3*n + 1;
    }
    int numRs = countRsIn(input);
    int numDs = countDsIn(input);
    if (numRs > numDs) accept();
    else reject();
}
```

No one knows!

An (evil) insecure voting machine.
Secure Voting

• A voting machine is a program that takes as input a string of \( r \)'s and \( d \)'s, then reports whether person \( r \) won the election.

• **Question:** Given a TM that someone claims is a secure voting machine, could we automatically check whether it actually is a secure voting machine?
Claim: A program that decides whether arbitrary input programs are secure voting machines is self-defeating. It therefore doesn’t exist.
A Decider for Secure Voting

• Let’s suppose that, somehow, we managed to build a decider for the secure voting problem.

• Schematically, that decider would look like this:

  ![Diagram of a decider for secure voting]

  - We could represent this decider in software as a method
    ```
    bool isSecureVotingMachine(string program);
    ```
  that takes as input a program, then returns whether that program is a secure voting machine.
What happens if...

... this program is a secure voting machine?  
then it's not a secure voting machine!

... this program is not a secure voting machine?  
then it's a secure voting machine!
**Theorem:** The secure voting problem is undecidable.

**Proof:** By contradiction; assume that the secure voting problem is decidable. Then there is some decider $D$ for the secure voting problem, which we can represent in software as a method `isSecureVotingMachine` that, given as input the source code of a program, returns true if the program is a secure voting machine and false otherwise.

Given this, we could then construct the following program $P$:

```java
int main() {
    string me = mySource();
    string input = getInput();

    bool answer = (countRs(input) > countDs(input));
    if (isSecureVotingMachine(me)) answer = !answer;

    if (answer) accept();
    else reject();
}
```

Now, either $P$ is a secure voting machine or it isn’t. If $P$ is a secure voting machine, then `isSecureVotingMachine(me)` will return true. Therefore, when $P$ is run, it will determine whether $w$ has more $r$’s than $d$’s, flip the result, and accept strings with at least as many $d$’s as $r$’s and reject strings with more $r$’s than $d$’s. Thus, $P$ is not a secure voting machine. On the other hand, if $P$ is not a secure voting machine, then `isSecureVotingMachine(me)` will return false. Therefore, when $P$ is run, it will accept all strings with at least as many $r$’s as $d$’s and reject all other strings, and so $P$ is a secure voting machine.

In both cases we reach a contradiction, so our assumption must have been wrong. Therefore, the secure voting problem is undecidable. ■
Interpreting this Result

• The previous argument tells us that there is no general algorithm that we can follow to determine whether a program is a secure voting machine. In other words, any general algorithm to check voting machines will always be wrong on at least one input.

• So what can we do?
  • Design algorithms that work in some, but not all cases. (This is often done in practice.)
  • Fall back on human verification of voting machines. (We do that too.)
  • Carry a healthy degree of skepticism about electronic voting machines. (Then again, did we even need the theoretical result for this?)
ASKING AIRCRAFT DESIGNERS ABOUT AIRPLANE SAFETY:
Nothing is ever foolproof, but modern airliners are incredibly resilient. Flying is the safest way to travel.

ASKING BUILDING ENGINEERS ABOUT ELEVATOR SAFETY:
Elevators are protected by multiple tried-and-tested failsafe mechanisms. They're nearly incapable of falling.

ASKING SOFTWARE ENGINEERS ABOUT COMPUTERIZED VOTING:
That's terrifying.

WAIT, REALLY?
Don't trust voting software and don't listen to anyone who tells you it's safe.

WHY?
I don't quite know how to put this, but our entire field is bad at what we do, and if you rely on us, everyone will die.

THEY SAY THEY'VE FIXED IT WITH SOMETHING CALLED "BLOCKCHAIN."
AAAAAA!!
Whatever they sold you, don't touch it.
Bury it in the desert.
Wear gloves.
Beyond R and RE
Beyond \textbf{R} and \textbf{RE}

- We've now seen how to use self-reference as a tool for showing undecidability (finding languages not in \textbf{R}).
- We still have not broken out of \textbf{RE} yet, though.
- To do so, we will need to build up a better intuition for the class \textbf{RE}.
What exactly is the class RE?
RE, Formally

• Recall that the class \textbf{RE} is the class of all recognizable languages:

\[ \text{RE} = \{ L \mid \text{there is a TM } M \text{ where } L(M) = L \} \]

• Since \( \mathbb{R} \neq \text{RE} \), there is no general way to “solve” problems in the class \textbf{RE}, if by “solve” you mean “make a computer program that can always tell you the correct answer.”

• So what exactly \textit{are} the sorts of languages in \textbf{RE}?
Does this graph contain a 4-clique?
Does this graph contain a 4-clique?
Does this graph contain a 4-clique?
Key Intuition:

A language $L$ is in $\text{RE}$ if, for any string $w$, if you are convinced that $w \in L$, there is some way you could prove that to someone else.
Does this Sudoku puzzle have a solution?
Does this graph have a Hamiltonian path (a simple path that passes through every node exactly once?)
Verification

11

Try running fourteen steps of the Hailstone sequence.

Does the hailstone sequence terminate for this number?
Does this Sudoku puzzle have a solution?
Does this graph have a Hamiltonian path (a simple path that passes through every node exactly once?)
Verification

11

Try running five steps of the Hailstone sequence.

Does the hailstone sequence terminate for this number?
Verification

- In each of the preceding cases, we were given some problem and some evidence supporting the claim that the answer is “yes.”
- Given the correct evidence, we can be certain that the answer is indeed “yes.”
- Given incorrect evidence, we aren't sure whether the answer is “yes.”
  - Maybe there's no evidence saying that the answer is “yes,” or maybe there is some evidence, but just not the evidence we were given.
- Let's formalize this idea.
Verifiers

- A **verifier** for a language $L$ is a TM $V$ with the following two properties:
  
  $V$ halts on all inputs.

  $\forall w \in \Sigma^*. (w \in L \iff \exists c \in \Sigma^*. V \text{ accepts } \langle w, c \rangle)$

- Intuitively, what does this mean?
Deciders and Verifiers

Decider $M$ for $L$

- **Input string** $(w)$
- **Certificate** $(c)$

$M$ halts on all inputs. $w \in L \iff M$ accepts $w$

Verifier $V$ for $L$

- **Input string** $(w)$
- **Certificate** $(c)$

$V$ halts on all inputs. $w \in L \iff \exists c \in \Sigma^*. V$ accepts $(w, c)$

"Solve the problem"

- **Yes!**
  - If $M$ accepts, then $w \in L$.
- **No!**
  - If $M$ rejects, then $w \notin L$.

"Check the answer"

- **Yes!**
  - If $V$ accepts $(w, c)$, then $w \in L$.
- **Not sure**
  - If $V$ rejects $(w, c)$, we don't know whether $w \in L$. 

If $M$ accepts, then $w \in L$.

If $M$ rejects, then $w \notin L$.

If $V$ accepts $(w, c)$, then $w \in L$.

If $V$ rejects $(w, c)$, we don't know whether $w \in L$. 

If $V$ accepts $(w, c)$, then $w \in L$.

If $V$ rejects $(w, c)$, we don't know whether $w \in L$. 

Verifiers

• A **verifier** for a language \( L \) is a TM \( V \) with the following properties:

\[
V \text{ halts on all inputs.}
\]

\[
\forall w \in \Sigma^*. \ (w \in L \iff \exists c \in \Sigma^*. \ V \text{ accepts } \langle w, c \rangle)
\]

• Some notes about \( V \):
  
  • If \( V \) accepts \( \langle w, c \rangle \), then we're guaranteed \( w \in L \).
  
  • If \( V \) rejects \( \langle w, c \rangle \), then either
    
    - \( w \in L \), but you gave the wrong \( c \), or
    
    - \( w \notin L \), so no possible \( c \) will work.
Verifiers

- A **verifier** for a language $L$ is a TM $V$ with the following properties:
  
  **$V$ halts on all inputs.**

  \[
  \forall w \in \Sigma^*. \ (w \in L \iff \exists c \in \Sigma^*. \ V \text{ accepts } (w, c))
  \]

- Some notes about $V$:
  
  - Notice that the certificate $c$ is existentially quantified. Any string $w \in L$ must have at least one $c$ that causes $V$ to accept, and possibly more.

  - $V$ is required to halt, so given any potential certificate $c$ for $w$, you can check whether the certificate is correct.
Verifiers

- A **verifier** for a language $L$ is a TM $V$ with the following properties:

  \[
  V \text{ halts on all inputs.}
  \]

  \[
  \forall w \in \Sigma^*. (w \in L \iff \exists c \in \Sigma^*. V \text{ accepts } \langle w, c \rangle)
  \]

- Some notes about $V$:
  - Notice that $\mathcal{L}(V) \neq L$. (*Good question: what is $\mathcal{L}(V)$?*)
  - The job of $V$ is just to check certificates, not to decide membership in $L$. 

Verifiers

- A **verifier** for a language $L$ is a TM $V$ with the following properties:

  $V$ halts on all inputs.

  $\forall w \in \Sigma^*. \ (w \in L \iff \exists c \in \Sigma^*. \ V \text{ accepts } \langle w, c \rangle)$

- Some notes about $V$:
  - Although this formal definition works with a string $c$, remember that $c$ can be an encoding of some other object.
  - In practice, $c$ will likely just be “some other auxiliary data that helps you out.”
Some Verifiers

• Let $L$ be the following language:

\[
L = \{ \langle n \rangle \mid n \in \mathbb{N} \text{ and the hailstone sequence terminates for } n \}\]

• Let's see how to build a verifier for $L$.

• This verifier will take as input
  • a natural number $n$, and
  • some certificate $c$.

• The certificate $c$ should be some evidence that suggests that the hailstone sequence terminates for $n$.

• What evidence could we provide?
Some Verifiers

• Let $L$ be the following language:

$$L = \{ \langle n \rangle | n \in \mathbb{N} \text{ and the hailstone sequence terminates for } n \}$$

```cpp
bool checkHailstone(int n, int c) {
    for (int i = 0; i < c; i++) {
        if (n % 2 == 0) n /= 2;
        else n = 3*n + 1;
    }
    return n == 1;
}
```

• Do you see why $\langle n \rangle \in L$ iff there is some $c$ such that checkHailstone($n$, $c$) returns true?

• Do you see why checkHailstone always halts?
Some Verifiers

• Let $L$ be the following language:
  
  $$L = \{ \langle G \rangle \mid G \text{ is a graph and } G \text{ has a Hamiltonian path} \}$$

• (Refresher: a Hamiltonian path is a simple path that visits every node in the graph.)

• Let's see how to build a verifier for $L$.

• Our verifier will take as input
  
  • a graph $G$, and
  • a certificate $c$.

• The certificate $c$ should be some evidence that suggests that $G$ has a Hamiltonian path.

• What information could we put into the certificate?
Some Verifiers

• Let $L$ be the following language:

\[ L = \{ \langle G \rangle \mid G \text{ is a graph with a Hamiltonian path} \} \]

```cpp
bool checkHamiltonian(Graph G, vector<Node> c) {
    if (c.size() != G.numNodes()) return false;
    if (containsDuplicate(c)) return false;
    for (size_t i = 0; i + 1 < c.size(); i++) {
        if (!G.hasEdge(c[i], c[i+1])) return false;
    }
    return true;
}
```

• Do you see why $\langle G \rangle \in L$ iff there is a $c$ where checkHamiltonian($G$, $c$) returns true?

• Do you see why checkHamiltonian always halts?
A Very Nifty Verifier

- Consider $A_{\text{TM}}$:
  \[ A_{\text{TM}} = \{ \langle M, w \rangle \mid M \text{ is a TM and } M \text{ accepts } w \} \].

- This is a *canonical* example of an undecidable language. There’s no way, in general, to tell whether a TM $M$ will accept a string $w$.

- Although this language is undecidable, it’s an \textbf{RE} language, and it’s possible to build a verifier for it!
A Very Nifty Verifier

• Consider $A_{\text{TM}}$:

$$A_{\text{TM}} = \{ \langle M, w \rangle \mid M \text{ is a TM and } M \text{ accepts } w \}.$$  

• We know that $U_{\text{TM}}$ is a recognizer for $A_{\text{TM}}$. It is also a verifier for $A_{\text{TM}}$?

• No, for two reasons:
  • $U_{\text{TM}}$ doesn’t always halt. (*Do you see why?*)
  • $U_{\text{TM}}$ takes as input a TM $M$ and a string $w$. A verifier also needs a certificate.
A Very Nifty Verifier

• Consider $A_{TM}$:

$$A_{TM} = \{ \langle M, w \rangle \mid M \text{ is a TM and } M \text{ accepts } w \}.$$  

• A verifier for $A_{TM}$ would take as input
  • A TM $M$,
  • a string $w$, and
  • a certificate $c$.

• The certificate $c$ should be some evidence that suggests that $M$ accepts $w$.

• What could our certificate be?
Run this TM for fifteen steps.
Some Verifiers

- Consider $A_{TM}$:

  $A_{TM} = \{ \langle M, w \rangle \mid M$ is a TM and $M$ accepts $w \}$.  

```cpp
bool checkWillAccept(TM M, string w, int c) {
    set up a simulation of M running on w;
    for (int i = 0; i < c; i++) {
        simulate the next step of M running on w;
    }
    return whether M is in an accepting state;
}
```

- Do you see why $M$ accepts $w$ iff there is some $c$ such that checkWillAccept($M, w, c$) returns true?

- Do you see why checkWillAccept always halts?
What languages are verifiable?
**Theorem:** If $L$ is a language, then there is a verifier for $L$ if and only if $L \in \text{RE}$. 
Where We’ve Been

NFA

State Elimination

Regex

Thompson’s Algorithm
Where We’re Going

Verifier

Try all certificates

Recognizer

Enforce a step count
Verifiers and \textbf{RE}

- \textbf{Theorem:} If there is a verifier $V$ for a language $L$, then $L \in \text{RE}$.
- \textbf{Proof goal:} Given a verifier $V$ for a language $L$, find a way to construct a recognizer $M$ for $L$.

\begin{itemize}
  \item \textbf{Requirements on a verifier $V$ for $L$:}
    \begin{align*}
      & V \text{ halts on all inputs.} \\
      & \forall w \in \Sigma^*. (w \in L \leftrightarrow \exists c \in \Sigma^*. V \text{ accepts } \langle w, c \rangle)
    \end{align*}
  \item \textbf{Requirements on a recognizer $M$ for $L$:}
    \begin{align*}
      & \forall w \in \Sigma^*. (w \in L \leftrightarrow M \text{ accepts } w)
    \end{align*}
\end{itemize}
Verifiers and \textbf{RE}

- **Theorem**: If there is a verifier $V$ for a language $L$, then $L \in \text{RE}$.

- **Proof goal**: Given a verifier $V$ for a language $L$, find a way to construct a recognizer $M$ for $L$. 

```
input string (w)  \rightarrow  \text{Verifier V for L}  \rightarrow  \text{Check the answer}
```

```
Certificate (c)  \rightarrow  yes!
```

```
\epsilon, a, b, aa, ab, ba, bb, aaa, aab, aba, abb, baa, ...
```
Verifiers and \textbf{RE}

\begin{itemize}
  \item \textbf{Theorem:} If $V$ is a verifier for $L$, then $L \in \textbf{RE}$.
  \item \textbf{Proof sketch:} Consider the following program:

\begin{verbatim}
bool isInL(string w) {
  for (each string c) {
    if (V accepts \langle w, c \rangle) return true;
  }
}
\end{verbatim}

If $w \in L$, there is some $c \in \Sigma^*$ where $V$ accepts $\langle w, c \rangle$. The function \texttt{isInL} tries all possible strings as certificates, so it will eventually find $c$ (or some other certificate), see $V$ accept $\langle w, c \rangle$, then return true. Conversely, if \texttt{isInL}(w) returns true, then there was some string $c$ such that $V$ accepted $\langle w, c \rangle$, so $w \in L$. ■
Verifiers and \textbf{RE}

- **Theorem:** If $L \in \text{RE}$, then there is a verifier for $L$.
- **Proof goal:** Beginning with a recognizer $M$ for the language $L$, show how to construct a verifier $V$ for $L$.

\textbf{Requirements on a recognizer $M$ for $L$:}

\[
\forall w \in \Sigma^*. \ (w \in L \leftrightarrow M \text{ accepts } w)
\]

\textbf{Requirements on a verifier $V$ for $L$:}

\[
V \text{ halts on all inputs.} \\
\forall w \in \Sigma^*. \ (w \in L \leftrightarrow \exists c \in \Sigma^*. \ V \text{ accepts } \langle w, c \rangle)
\]
We have a recognizer for a language. We want to turn it into a verifier. Where did we see this before?
Some Verifiers

Consider $A_{TM}$:

$$A_{TM} = \{ \langle M, w \rangle \mid M \text{ is a TM and } M \text{ accepts } w \}.$$  

```cpp
bool checkWillAccept(TM M, string w, int c) {
    set up a simulation of M running on w;
    for (int i = 0; i < c; i++) {
        simulate the next step of M running on w;
    }
    return whether M is in an accepting state;
}
```

**Observation 1:** This trick of enforcing a step count limits how long $M$ can run for!

Do you see why $M$ accepts $w$ iff there is some $c$ such that $checkWillAccept(M, w, c)$ returns true? Do you see why $checkWillAccept$ always halts?
Verifiers and \textbf{RE}

- **Theorem:** If $L \in \text{RE}$, then there is a verifier for $L$.
- **Proof sketch:** Let $L$ be a \textbf{RE} language and let $M$ be a recognizer for it. Consider this function:

  ```
  bool checkIsInL(string w, int c) {
    TM M = /* hardcoded version of a recognizer for L */;
    set up a simulation of M running on w;
    for (int i = 0; i < c; i++) {
      simulate the next step of M running on W;
    }
    return whether M is in an accepting state;
  }
  ```

Note that \texttt{checkIsInL} always halts, since each step takes only finite time to complete. Next, notice that if there is a $c$ where \texttt{checkIsInL}(w, c) returns true, then $M$ accepted $w$ after running for $c$ steps, so $w \in L$. Conversely, if $w \in L$, then $M$ accepts $w$ after some number of steps (call that number $c$). Then \texttt{checkIsInL}(w, c) will run $M$ on $w$ for $c$ steps, watch $M$ accept $w$, then return true. ■
Here’s another proof of this result.

It’s a bit more theoretically elegant, and hypothetically speaking, it might be useful in the future.
Consider \( A_{TM} \):

\[
A_{TM} = \{ \langle M, w \rangle \mid M \text{ is a TM and } M \text{ accepts } w \}.
\]

```c
bool checkWillAccept(TM M, string w, int c) {
    set up a simulation of M running on w;
    for (int i = 0; i < c; i++) {
        simulate the next step of M running on w;
    }
    return whether M is in an accepting state;
}
```

Observation 2: This verifier for \( A_{TM} \) is a “universal verifier!” We can use it as a subroutine!

Do you see why \( M \) accepts \( w \) iff there is some \( c \) such that \( \text{checkWillAccept}(M, w, c) \) returns true?

Do you see why \( \text{checkWillAccept} \) always halts?
Verifiers and \( \text{RE} \)

- **Theorem:** If \( L \in \text{RE} \), then there is a verifier for \( L \).

- **Proof sketch:** Let \( L \in \text{RE} \) and let \( M \) be a recognizer for it. Let \( \text{checkWillAccept} \) be our earlier verifier for \( \text{A}_{\text{TM}} \). Consider this code:

```c
bool checkIsInL(string w, int c) {
    TM M = /* hardcoded version of a recognizer for L */;
    return checkWillAccept(M, w, c);
}
```

First, note that \( \text{checkIsInL} \) always returns a value because it just calls \( \text{checkWillAccept} \), which always returns.

Next, suppose there’s a \( c \) where \( \text{checkIsInL}(w, c) \) returns true. Then there’s a \( c \) where \( \text{checkWillAccept}(M, w, c) \) returns true, so \( \langle M, w \rangle \in \text{A}_{\text{TM}} \). This means that \( M \) accepts \( w \), so \( w \in L \).

Finally, suppose there is no \( c \) where \( \text{checkIsInL}(w, c) \) returns true. Then there is no \( c \) where \( \text{checkWillAccept}(M, w, c) \) returns true, so \( \langle M, w \rangle \notin \text{A}_{\text{TM}} \). This means that \( M \) does not accept \( w \), so \( w \notin L \). ■
Verifiers and recognizers give two different perspectives on the “proof” intuition for RE.

Verifiers are explicitly built to check proofs that strings are in the language.

- If you know that some string $w$ belongs to the language and you have the proof of it, you can convince someone else that $w \in L$.

You can think of a recognizer as a device that “searches” for a proof that $w \in L$.

- If it finds it, great!
- If not, it might loop forever.
RE and Proofs

• If the RE languages represent languages where membership can be proven, what does a non-RE language look like?

• Intuitively, a language is not in RE if there is no general way to prove that a given string $w \in L$ actually belongs to $L$.

• In other words, even if you knew that a string was in the language, you may never be able to convince anyone of it!
Time-Out for Announcements!
Problem Set Nine

- Problem Set Eight was due today at 2:30PM.
  - You can use late days here to extend the deadline as far as Sunday at 2:30PM, but we don’t recommend this.

- Problem Set Nine goes out today. It’s due next Friday at 2:30PM.
  - Play around with the limits of R and RE languages – the upper extent of computation!
  - See how everything fits together!

- Due to university policies, no late submissions will be accepted for PS9. Please budget at least two hours before the deadline to submit the assignment.
The Last Two Guides

• We’ve posted two final guides to the course website:
  • The *Guide to the Lava Diagram*, which provides an intuition for how different classes of languages relate to one another.
• Give these a read – there’s a ton of useful information in there!
Final Exam Logistics

• Our final exam is Monday, December 10\textsuperscript{th} from 3:30PM – 6:30PM, location TBA.
  • Sorry about how soon that is – the registrar picked this time, not us. If we had a choice, it would be on the last day of finals week.

• The exam is cumulative. You’re responsible for topics from PS0 – PS9 and all of the lectures.

• As with the midterms, the exam is closed-book, closed-computer, and limited-note. You can bring one double-sided sheet of 8.5” × 11” notes with you to the exam, decorated any way you’d like.

• Students with OAE accommodations: if we don’t yet have your OAE letter, please send it to us ASAP.
Preparing for the Exam

• We’ve posted *seven* practice final exams, with solutions, to the course website.

• These exams are essentially the final exams we’ve given out in the last seven quarters, with a few tweaks and modifications.

• Practice Final 6 and Practice Final 7 are the two most recent exams and should give you the best indicator of the expected topic coverage.

• And don’t forget that Extra Practice Problems 3 is available online. After today’s lecture, you know enough to take on any of those questions, including the starred ones.
Your Questions
“Keith, can you please make the last problem set a bit shorter than the others have been? We all have a lot to do. You'd be a homie. Thanks.”

That’s the plan! Hopefully the practice matches the theory.
“What do you think of splitting CS103 over two quarters? And adding more depth to the material. I feel like some topics needed more love”

I wish that we had more time to cover things this quarter. You’d be shocked by how big the Graveyard of Former CS103 Topics is. Every quarter we nix a bunch of topics and introduce new ones, and I’m always tweaking things to try to find the optimal blend of topics.

Way Back In The Day, CS103 was a two-quarter sequence and the CS major required CS103A, CS103B, CS154, and CS161. After the Great CS Major Revamp in the late 2000’s we added in CS109, moved topics from CS103A and CS103B into CS109, shifted content from CS154 into CS103, and settled on CS103/CS109/CS161 as the theory core.

No matter how much time you spend on this material, you won’t have time to see everything. On our last day of class I’ll talk about classes you can take to learn more. Taking all of them essentially gives you a math minor and fills out something like half of the CS major.
Back to CS103!
Finding Non-RE Languages
Finding Non-RE Languages

- Right now, we know that non-RE languages exist, but we have no idea what they look like.
- How might we find one?
Languages, TMs, and TM Encodings

- Recall: The language of a TM $M$ is the set
  \[ \mathcal{L}(M) = \{ w \in \Sigma^* \mid M \text{ accepts } w \} \]

- Some of the strings in this set might be descriptions of TMs.

- What happens if we list off all Turing machines, looking at how those TMs behave given other TMs as input?
All Turing machines, listed in some order.
All descriptions of TMs, listed in the same order.
<table>
<thead>
<tr>
<th>$M_0$</th>
<th>Acc</th>
<th>No</th>
<th>No</th>
<th>Acc</th>
<th>Acc</th>
<th>No</th>
<th>...</th>
</tr>
</thead>
<tbody>
<tr>
<td>$M_1$</td>
<td>Acc</td>
<td>Acc</td>
<td>Acc</td>
<td>Acc</td>
<td>Acc</td>
<td>Acc</td>
<td>...</td>
</tr>
<tr>
<td>$M_2$</td>
<td>Acc</td>
<td>Acc</td>
<td>Acc</td>
<td>Acc</td>
<td>Acc</td>
<td>Acc</td>
<td>...</td>
</tr>
<tr>
<td>$M_3$</td>
<td>No</td>
<td>Acc</td>
<td>Acc</td>
<td>No</td>
<td>Acc</td>
<td>Acc</td>
<td>...</td>
</tr>
<tr>
<td>$M_4$</td>
<td>Acc</td>
<td>No</td>
<td>Acc</td>
<td>No</td>
<td>Acc</td>
<td>No</td>
<td>...</td>
</tr>
<tr>
<td>$M_5$</td>
<td>No</td>
<td>No</td>
<td>Acc</td>
<td>Acc</td>
<td>No</td>
<td>No</td>
<td>...</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>$M_0$</td>
<td>$M_1$</td>
<td>$M_2$</td>
<td>$M_3$</td>
<td>$M_4$</td>
<td>$M_5$</td>
<td>...</td>
<td></td>
</tr>
<tr>
<td>-------</td>
<td>-------</td>
<td>-------</td>
<td>-------</td>
<td>-------</td>
<td>-------</td>
<td>-----</td>
<td></td>
</tr>
<tr>
<td>Acc</td>
<td>No</td>
<td>No</td>
<td>Acc</td>
<td>Acc</td>
<td>No</td>
<td>...</td>
<td></td>
</tr>
<tr>
<td>Acc</td>
<td>Acc</td>
<td>Acc</td>
<td>Acc</td>
<td>Acc</td>
<td>Acc</td>
<td>...</td>
<td></td>
</tr>
<tr>
<td>Acc</td>
<td>Acc</td>
<td>Acc</td>
<td>Acc</td>
<td>Acc</td>
<td>Acc</td>
<td>...</td>
<td></td>
</tr>
<tr>
<td>No</td>
<td>Acc</td>
<td>Acc</td>
<td>No</td>
<td>Acc</td>
<td>Acc</td>
<td>...</td>
<td></td>
</tr>
<tr>
<td>Acc</td>
<td>No</td>
<td>Acc</td>
<td>No</td>
<td>Acc</td>
<td>No</td>
<td>...</td>
<td></td>
</tr>
<tr>
<td>No</td>
<td>No</td>
<td>Acc</td>
<td>Acc</td>
<td>No</td>
<td>No</td>
<td>...</td>
<td></td>
</tr>
<tr>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
<td></td>
</tr>
<tr>
<td>Acc</td>
<td>Acc</td>
<td>Acc</td>
<td>No</td>
<td>Acc</td>
<td>No</td>
<td>...</td>
<td></td>
</tr>
<tr>
<td></td>
<td>(M_0)</td>
<td>(M_1)</td>
<td>(M_2)</td>
<td>(M_3)</td>
<td>(M_4)</td>
<td>(M_5)</td>
<td>…</td>
</tr>
<tr>
<td>---</td>
<td>---------</td>
<td>---------</td>
<td>---------</td>
<td>---------</td>
<td>---------</td>
<td>---------</td>
<td>---</td>
</tr>
<tr>
<td>(M_0)</td>
<td>Acc</td>
<td>No</td>
<td>No</td>
<td>Acc</td>
<td>Acc</td>
<td>No</td>
<td>…</td>
</tr>
<tr>
<td>(M_1)</td>
<td>Acc</td>
<td>Acc</td>
<td>Acc</td>
<td>Acc</td>
<td>Acc</td>
<td>Acc</td>
<td>…</td>
</tr>
<tr>
<td>(M_2)</td>
<td>Acc</td>
<td>Acc</td>
<td>Acc</td>
<td>Acc</td>
<td>Acc</td>
<td>Acc</td>
<td>…</td>
</tr>
<tr>
<td>(M_3)</td>
<td>No</td>
<td>Acc</td>
<td>Acc</td>
<td>No</td>
<td>Acc</td>
<td>Acc</td>
<td>…</td>
</tr>
<tr>
<td>(M_4)</td>
<td>Acc</td>
<td>No</td>
<td>Acc</td>
<td>No</td>
<td>Acc</td>
<td>No</td>
<td>…</td>
</tr>
<tr>
<td>(M_5)</td>
<td>No</td>
<td>No</td>
<td>Acc</td>
<td>Acc</td>
<td>No</td>
<td>No</td>
<td>…</td>
</tr>
<tr>
<td>…</td>
<td>…</td>
<td>…</td>
<td>…</td>
<td>…</td>
<td>…</td>
<td>…</td>
<td>…</td>
</tr>
</tbody>
</table>

Flip all “accept” to “no” and vice-versa.
<table>
<thead>
<tr>
<th>$M_0$</th>
<th>$M_1$</th>
<th>$M_2$</th>
<th>$M_3$</th>
<th>$M_4$</th>
<th>$M_5$</th>
<th>...</th>
</tr>
</thead>
<tbody>
<tr>
<td>Acc</td>
<td>No</td>
<td>No</td>
<td>Acc</td>
<td>Acc</td>
<td>No</td>
<td>...</td>
</tr>
<tr>
<td>Acc</td>
<td>Acc</td>
<td>Acc</td>
<td>Acc</td>
<td>Acc</td>
<td>Acc</td>
<td>...</td>
</tr>
<tr>
<td>Acc</td>
<td>Acc</td>
<td>Acc</td>
<td>Acc</td>
<td>Acc</td>
<td>Acc</td>
<td>...</td>
</tr>
<tr>
<td>No</td>
<td>Acc</td>
<td>Acc</td>
<td>No</td>
<td>Acc</td>
<td>Acc</td>
<td>...</td>
</tr>
<tr>
<td>Acc</td>
<td>No</td>
<td>Acc</td>
<td>No</td>
<td>Acc</td>
<td>No</td>
<td>...</td>
</tr>
<tr>
<td>No</td>
<td>No</td>
<td>Acc</td>
<td>Acc</td>
<td>No</td>
<td>No</td>
<td>...</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
</tr>
</tbody>
</table>

No TM has this behavior!
\[
\{ \langle M \rangle \mid M \text{ is a TM that does not accept } \langle M \rangle \} 
\]

<table>
<thead>
<tr>
<th>\langle M_0 \rangle</th>
<th>\langle M_1 \rangle</th>
<th>\langle M_2 \rangle</th>
<th>\langle M_3 \rangle</th>
<th>\langle M_4 \rangle</th>
<th>\langle M_5 \rangle</th>
<th>\ldots</th>
</tr>
</thead>
<tbody>
<tr>
<td>(M_0)</td>
<td>Acc</td>
<td>No</td>
<td>No</td>
<td>Acc</td>
<td>Acc</td>
<td>No</td>
</tr>
<tr>
<td>(M_1)</td>
<td>Acc</td>
<td>Acc</td>
<td>Acc</td>
<td>Acc</td>
<td>Acc</td>
<td>Acc</td>
</tr>
<tr>
<td>(M_2)</td>
<td>Acc</td>
<td>Acc</td>
<td>Acc</td>
<td>Acc</td>
<td>Acc</td>
<td>Acc</td>
</tr>
<tr>
<td>(M_3)</td>
<td>No</td>
<td>Acc</td>
<td>Acc</td>
<td>No</td>
<td>Acc</td>
<td>Acc</td>
</tr>
<tr>
<td>(M_4)</td>
<td>Acc</td>
<td>No</td>
<td>Acc</td>
<td>No</td>
<td>Acc</td>
<td>No</td>
</tr>
<tr>
<td>(M_5)</td>
<td>No</td>
<td>No</td>
<td>Acc</td>
<td>Acc</td>
<td>No</td>
<td>No</td>
</tr>
<tr>
<td>\ldots</td>
<td>\ldots</td>
<td>\ldots</td>
<td>\ldots</td>
<td>\ldots</td>
<td>\ldots</td>
<td>\ldots</td>
</tr>
</tbody>
</table>

| No | No | No | Acc | No | Acc | \ldots |
Diagonalization Revisited

• The *diagonalization language*, which we denote $L_D$, is defined as

$$L_D = \{ \langle M \rangle \mid M \text{ is a TM and } M \text{ does not accept } \langle M \rangle \}$$

• We constructed this language to be different from the language of every TM.

• Therefore, $L_D \notin \text{RE}$! Let’s go prove this.
\( L_D = \{ \langle M \rangle \mid M \text{ is a TM and } M \text{ does not accept } \langle M \rangle \} \)

**Theorem:** \( L_D \notin \text{RE} \).

**Proof:** By contradiction; assume that \( L_D \in \text{RE} \). This means that there is a TM \( R \) where \( \mathcal{L}(R) = L_D \).

Now, what happens when we run \( R \) on \( \langle R \rangle \)? We know that

\[
R \text{ accepts } \langle R \rangle \text{ if and only if } \langle R \rangle \in \mathcal{L}(R).
\]

Since \( \mathcal{L}(R) = L_D \), the above expression simplifies to

\[
R \text{ accepts } \langle R \rangle \text{ if and only if } \langle R \rangle \in L_D.
\]

Finally, by definition of \( L_D \), we know that \( \langle R \rangle \in L_D \) if and only if \( R \) does not accept \( \langle R \rangle \). Therefore, we see that

\[
R \text{ accepts } \langle R \rangle \text{ if and only if } R \text{ doesn’t accept } \langle R \rangle.
\]

This is impossible. We’ve reached a contradiction, so our assumption was wrong, and so \( L_D \notin \text{RE} \). \( \blacksquare \)
What This Means

- On a deeper philosophical level, the fact that non-RE languages exist supports the following claim:
  
  \textit{There are statements that are true but not provable.}

- Intuitively, given any non-RE language, there will be some string in the language that cannot be proven to be in the language.

- This result can be formalized as a result called \textit{Gödel's incompleteness theorem}, one of the most important mathematical results of all time.

- Want to learn more? Take Phil 152 or CS154!
What This Means

• On a more philosophical note, you could interpret the previous result in the following way:

  There are inherent limits about what mathematics can teach us.

• There's no automatic way to do math. There are true statements that we can't prove.

• That doesn't mean that mathematics is worthless. It just means that we need to temper our expectations about it.
Where We Stand

- We've just done a crazy, whirlwind tour of computability theory:
  - The Church-Turing thesis tells us that TMs give us a mechanism for studying computation in the abstract.
  - Universal computers – computers as we know them – are not just a stroke of luck. The existence of the universal TM ensures that such computers must exist.
  - Self-reference is an inherent consequence of computational power.
  - Undecidable problems exist partially as a consequence of the above and indicate that there are statements whose truth can't be determined by computational processes.
  - Unrecognizable problems are out there and can be discovered via diagonalization. They imply there are limits to mathematical proof.
Where We've Been

- The class $\mathbf{R}$ represents problems that can be solved by a computer.
- The class $\mathbf{RE}$ represents problems where “yes” answers can be verified by a computer.
Where We're Going

- The class $\mathbf{P}$ represents problems that can be solved \textit{efficiently} by a computer.

- The class $\mathbf{NP}$ represents problems where “yes” answers can be verified \textit{efficiently} by a computer.
Next Time

• *Introduction to Complexity Theory*
  • Not all decidable problems are created equal!

• *The Classes P and NP*
  • Two fundamental and important complexity classes.

• *The $P \neq NP$ Question*
  • A literal million-dollar question!