Unsolvable Problems
Part Two
Outline for Today

- **More on Undecidability**
  - Even more problems we can’t solve.
- **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 is possible to build Turing machines that get their own encodings and perform arbitrary computations on them.
What does this program do?

```c
bool willAccept(string program, string input) {
    /* … some implementation … */
}
int main() {
    string me = mySource();
    string input = getInput();

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

This program now has its own source code stored in variable me.

The program now asks – am I going to accept my input?
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();
    }
}
```

Try running this program on any input.
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?

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

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

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

A self-defeating object: Using that object against itself.

What does this program do?

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

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

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

"The largest integer, n."

"The number n + 1."
All Languages

Regular Languages

CFLs

R

RE

$A_{TM}$
New Stuff!
What Does This Mean?

- In one fell swoop, we've proven that
  - A decider for \( A_{\text{TM}} \) is a self-defeating object.
  - \( A_{\text{TM}} \) is **undecidable**; there is no general algorithm that can determine whether a TM will accept a string.
  - \( R \neq \text{RE} \), because \( A_{\text{TM}} \notin R \) but \( A_{\text{TM}} \in \text{RE} \).
- What do these three statements really mean? As in, why should you care?
Self-Defeating Objects

- The fact that a decider for $A_{TM}$ is a self-defeating object is analogous to this classic philosophical question:

  \textit{If you know what you are fated to do, can you avoid your fate?}

- If we have a decider for $A_{TM}$, we could use it to build a TM that determines what it’s supposed to do next, then chooses to do the opposite!
$A_{TM} \notin \mathbb{R}$

- What exactly does it mean for $A_{TM}$ to be undecidable?

  *Intuition: The only general way to find out what a program will do is to run it.*

- As you'll see, this means that it's provably impossible for computers to be able to answer most questions about what a program will do.
\[ A_{TM} \notin R \]

- At a more fundamental level, the existence of undecidable problems tells us the following:

  There is a difference between what is true and what we can discover is true.

- Given a TM \( M \) and a string \( w \), one of these two statements is true:

  \( M \) accepts \( w \) \hspace{1cm} \( M \) does not accept \( w \)

But since \( A_{TM} \) is undecidable, there is no algorithm that can always determine which of these statements is true!
Because $R \neq \text{RE}$, there is a difference between decidability and recognizability:

*In some sense, it is fundamentally harder to solve a problem than it is to check an answer.*

There are problems where, when the answer is “yes,” you can confirm it (run a recognizer), but where if you don’t have the answer, you can’t come up with it in a mechanical way (build a decider).
More Impossibility Results
The Halting Problem

• The most famous undecidable problem is the **halting problem**, which asks:

  Given a TM $M$ and a string $w$, will $M$ halt when run on $w$?

• As a formal language, this problem would be expressed as

  $$HALT = \{ \langle M, w \rangle \mid M \text{ is a TM that halts on } w \}$$

• This is an **RE** language. (*We’ll see why later.*)

• How do we know that it’s undecidable?
Claim: A decider for $HALT$ is a self-defeating object. It therefore doesn’t exist.
A Decider for \textit{HALT}

- Let’s suppose that, somehow, we managed to build a decider for $HALT = \{ \langle M, w \rangle \mid M \text{ is a TM that halts on } w \}$. 
- Schematically, that decider would look like this:

```
participants

\text{Decider for } HALT

inputs: \langle M, w \rangle

outputs: \text{Yes, } M \text{ halts on } w.
\text{No, } M \text{ loops on } w.
```

- We could represent this decider in software as a method

```c++
bool willHalt(string program, string input);
```

that takes as input a program and a string, then returns whether that program will halt on that string.
What does this program do?

```cpp
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();
    }
}
```

This program now has its own source code stored in variable `me`.

The program now asks - will I eventually stop running?
What does this program do?

Imagine running this program on some input. What happens if...

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

... this program loops on this input?
It halts on 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();
    }
}
```

The self-defeating object.

Using that object against itself.
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();
    }
}
```

"The largest integer, n"

"The number n+1"
**Theorem:** \( \text{HALT} \notin \mathbb{R} \).

**Proof:** By contradiction; assume that \( \text{HALT} \in \mathbb{R} \). Then there’s a decider \( D \) for \( \text{HALT} \), which we can represent in software as a method \( \text{willHalt} \) that takes as input the source code of a program and an input, then returns true if the program halts on the input and false otherwise.

Given this, we could then construct this program \( P \):

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

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

Choose any string \( w \) and trace through the execution of program \( P \) on input \( w \), focusing on the answer given back by the \( \text{willHalt} \) method. If \( \text{willHalt}(me, input) \) returns true, then \( P \) must halt on its input \( w \). However, in this case \( P \) proceeds to loop infinitely on \( w \). Otherwise, if \( \text{willHalt}(me, input) \) returns false, then \( P \) must not halt its input \( w \). However, in this case \( P \) proceeds to accept its input \( w \).

In both cases we reach a contradiction, so our assumption must have been wrong. Therefore, \( \text{HALT} \notin \mathbb{R} \). ■
\textbf{HALT} \in \textbf{RE}

- \textbf{Claim:} \textit{HALT} \in \textbf{RE}.
- \textbf{Idea:} If you were certain that a TM \( M \) halted on a string \( w \), could you convince me of that?
- Yes – just run \( M \) on \( w \) and see what happens!

```cpp
int main() {
    TM M = getInputTM();
    string w = getInputString();

    feed w into M;
    while (true) {
        if (M is in an accepting state) accept();
        else if (M is in a rejecting state) accept();
        else simulate one more step of M running on w;
    }
}
```
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 $w \in \Sigma^*$ corresponds to a series of votes for the candidates.

• Example: $rrddddrd$ 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
\[ \mathcal{L}(M) = \{ w \in \Sigma^* \mid w \text{ has more r's than d's} \} \]

### Secure Voting Machine

```cpp
def main() -> int:
    input = input()
    numRs = countRsIn(input)
    numDs = countDsIn(input)
    if numRs > numDs:
        accept()
    else:
        reject()
```

A (simple) secure voting machine.

### Insecure Voting Machine 1

```cpp
def main() -> int:
    input = input()
    if input[0] == 'r':
        accept()
    else:
        reject()
```

A (simple) insecure voting machine.

### Insecure Voting Machine 2

```cpp
def main() -> int:
    input = input()
    n = len(input)
    while n > 1:
        if n % 2 == 0:
            n /= 2
        else:
            n = 3 * n + 1
    numRs = countRsIn(input)
    numDs = countDsIn(input)
    if numRs > numDs:
        accept()
    else:
        reject()
```

An (evil) insecure voting machine.

No one knows!
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](attachment://diagram.png)

- 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 $P$:

```cpp
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?)
Beyond R and RE
Beyond $\mathbb{R}$ and $\mathbb{RE}$

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

- Recall that the class **RE** is the class of all recognizable languages:
  \[ \text{RE} = \{ L \mid \text{there is a TM } M \text{ where } L(M) = L \} \]

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

- So what exactly are the sorts of languages in **RE**?
Does this graph contain four mutually adjacent nodes?
Does this graph contain four mutually adjacent nodes?
Does this graph contain four mutually adjacent nodes?
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 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?
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** $(w)$
- $M$ 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"

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

"Check the answer"

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

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

• Some notes about $V$:
  
  • Notice that 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:

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

• Some notes about $V$:
  
  • 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:

$$L = \{ \langle G \rangle \mid \text{G is a graph and G has a Hamiltonian path} \}$$

(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 $\langle G \rangle \in L$ if and only if there is a $c$ where checkHamiltonian($G$, $c$) returns true?
- Do you see why checkHamiltonian always halts?
A Very Nifty Verifier

• Consider $A_{TM}$:

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

• 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 RE language, and it’s possible to build a verifier for it!
A Very Nifty Verifier

- Consider $A_{TM}$:
  $$A_{TM} = \{ \langle M, w \rangle \mid M \text{ is a TM and } M \text{ accepts } w \}.$$  
- We know that $U_{TM}$ is a recognizer for $A_{TM}$. Is it also a verifier for $A_{TM}$?
- No, for two reasons:
  - $U_{TM}$ doesn't always halt. *(Do you see why?)*
  - $U_{TM}$ takes as input a TM $M$ and a string $w$. A verifier also needs a certificate.
A Very Nifty Verifier

• Consider \( A_{\text{TM}} \):

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

• A verifier for \( A_{\text{TM}} \) 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 $A_{TM}$:

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

  ```
  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 $w$ if and only if there is a $c$ 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  Regex

State Elimination

NFA → Regex

Thompson’s Algorithm

Regex → NFA
Where We’re Going

Verifier

Try all certificates

Recognizer

Enforce a step count
Verifiers and 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 $L$.

Requirements on a verifier $V$ for $L$:

$V$ halts on all inputs.
$
\forall w \in \Sigma^*. \ (w \in L \leftrightarrow \exists c \in \Sigma^*. \ V \text{ accepts } \langle w, c \rangle)
$

Requirements on a recognizer $M$ for $L$:

$
\forall w \in \Sigma^*. \ (w \in L \leftrightarrow M \text{ accepts } w)
$
Verifiers and $\text{RE}$

- **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 $L$.

\begin{quote}
**Key intuition:** All devices that give back $\text{RE}$ languages can be thought of as finding evidence that strings are indeed in the language.

A verifier says “someone else needs to give me the evidence; I’ll just check it.”

A recognizer says “let me search far and wide and see if I can find some evidence.”
\end{quote}
Verifiers and $\textbf{RE}$

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

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

```
Verifier V for L
```
Verifiers and RE

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

**Proof sketch:** Consider the following program:

```plaintext
bool isInL(string w) {
    for (each string c) {
        if (V accepts $\langle w, c \rangle$) return true;
    }
}
```

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

\[
\begin{align*}
\text{Requirements on a recognizer } M \text{ for } L: \\
\forall w \in \Sigma^*. (w \in L \leftrightarrow M \text{ accepts } w)
\end{align*}
\]

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

Consider $A_{TM}$:

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

```python
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: 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?
Theorem: If $L \in \text{RE}$, then there is a verifier for $L$.

Proof sketch: Let $L$ be a RE language and let $M$ be a recognizer for it. Consider this function:

```cpp
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 checkIsInL always halts, since each step takes only finite time to complete. Next, notice that if there is a $c$ where 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 checkIsInL($w$, $c$) will run $M$ on $w$ for $c$ steps, watch $M$ accept $w$, then return true.
**RE and Proofs**

- 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 \textbf{RE} languages represent languages where membership can be proven, what does a non-\textbf{RE} language look like?
- Intuitively, a language is not in \textbf{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.

• Problem Set Nine goes out today. It’s due the Friday after break 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.
Thanksgiving Break Logistics

- We will not be holding our regular office hours over the break.
- Jackie will be holding SCPD office hours as usual on Saturday, November 23rd. We won’t have SCPD office hours on Saturday, November 30th.
- We’ll still be monitoring Piazza. Please ping us with questions!
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!
Preparing for the Exam

• We’ve posted *eight* 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 Finals 1, 6, and 7 are the three most recent exams and should give you the best indicator of the expected topic coverage.

• We’ve also posted Extra Practice Problems 3, with solutions. That’s over forty more problems to pick and choose from.
Your Questions
“Can we as humans be thought of as computing devices? And if so, where do we fit into the solvability venn diagram? (Who would win... man or machine?)”

That’s a question that’s so scary that I try not to think about it. But the fact that we ourselves love thinking about self-reference and related ideas like fate, destiny, and time travel might give a sense of what our limits are. 😊
"When the choice is yours, what do you read, listen to, or watch?"

Here’s a list of recommendations in every domain!


SPECIFIC ARTICLES: "Scott and Scurvy," "Mother Board Mother Earth," "The Things That Carried Him."


NON-MUSIC LISTENING: Any and all Supreme Court oral arguments, "99% Invisible," "Fresh Air."

MUSIC: "Random Access Memories" by Daft Punk, "Live at the Quick" by Bela Fleck, "City Folk" by James Farm, "Converting Vegetarians" by Infected Mushroom, "Aja" by Steely Dan, "We Live Here" by Pat Metheny Group, "Stop Making Sense" by Talking Heads, "Rumours" by Fleetwood Mac, "X&Y" by Coldplay, "MTV Unplugged in New York" by Nirvana, "Little Big" by Aaron Parks, "Taming the Dragon" by Mehliana, "The Nightfly" by Donald Fagen
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>$M_1$</th>
<th>$M_2$</th>
<th>$M_3$</th>
<th>$M_4$</th>
<th>$M_5$</th>
<th>$\ldots$</th>
</tr>
</thead>
<tbody>
<tr>
<td>Acc</td>
<td>No</td>
<td>No</td>
<td>Acc</td>
<td>Acc</td>
<td>No</td>
<td>$\ldots$</td>
</tr>
<tr>
<td>Acc</td>
<td>Acc</td>
<td>Acc</td>
<td>Acc</td>
<td>Acc</td>
<td>Acc</td>
<td>$\ldots$</td>
</tr>
<tr>
<td>Acc</td>
<td>Acc</td>
<td>Acc</td>
<td>Acc</td>
<td>Acc</td>
<td>Acc</td>
<td>$\ldots$</td>
</tr>
<tr>
<td>No</td>
<td>Acc</td>
<td>Acc</td>
<td>No</td>
<td>Acc</td>
<td>Acc</td>
<td>$\ldots$</td>
</tr>
<tr>
<td>Acc</td>
<td>No</td>
<td>Acc</td>
<td>No</td>
<td>Acc</td>
<td>No</td>
<td>$\ldots$</td>
</tr>
<tr>
<td>No</td>
<td>No</td>
<td>Acc</td>
<td>Acc</td>
<td>No</td>
<td>No</td>
<td>$\ldots$</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>
<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>-------</td>
<td>-----</td>
<td>----</td>
<td>----</td>
<td>-----</td>
<td>-----</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>
</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>...</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>
</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>
</tr>
<tr>
<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>
</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>
</tbody>
</table>

Flip all “accept” to “no” and vice-versa.
No TM has this behavior!

<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
No
No
Acc
No
Acc
...
<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>...</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>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
</tr>
</tbody>
</table>

“The language of all TMs that do not accept their descriptions.”
<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>...</th>
</tr>
</thead>
</table>
| $M_0$               | Acc                 | No                  | No                  | Acc                 | Acc                 | No  |...
| $M_1$               | Acc                 | Acc                 | Acc                 | Acc                 | Acc                 | Acc |...
| $M_2$               | Acc                 | Acc                 | Acc                 | Acc                 | Acc                 | Acc |...
| $M_3$               | No                  | Acc                 | Acc                 | No                  | Acc                 | Acc |...
| $M_4$               | Acc                 | No                  | Acc                 | No                  | Acc                 | No  |...
| $M_5$               | No                  | No                  | Acc                 | Acc                 | No                  | No  |...
| ...                 | ...                 | ...                 | ...                 | ...                 | ...                 | ... |}

$\{ \langle M \rangle \mid M \text{ is a TM that does not accept } \langle M \rangle \}$
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:** Assume for the sake of contradiction that \( L_D \in \text{RE}. \) This means that there is a recognizer \( R \) for \( L_D. \)

Now, focus on what happens if we run recognizer \( R \) on its own string encoding (that is, running \( R \) on \( \langle R \rangle \)). Since \( R \) is a recognizer for \( L_D, \) we see that

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

By definition of \( L_D, \) we know that

\[ \langle R \rangle \in L_D \text{ if and only if } R \text{ does not accept } \langle R \rangle. \]

Combining the two above statements tells us that

\[ R \text{ accepts } \langle R \rangle \text{ if and only if } R \text{ does not accept } \langle R \rangle. \]

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

- On a deeper philosophical level, the fact that non-RE languages exist supports the following claim:

  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 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.
The Big Picture

- DFA
- NFA
- Regex
- CFG
- Decider
- Recognizer
- Verifier

REG

CFL

R

RE
Where We've Been

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

• The class \textbf{P} represents problems that can be solved \textit{efficiently} by a computer.

• The class \textbf{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!