Turing Machines
Part Three
Outline for Today

- **More on R and RE**
  - An intuition for decidability and recognizability.

- **Why Languages and Strings?**
  - We’ve been using languages to model problems. Why?

- **Universal Machines**
  - A single computer that can compute anything computable anywhere.
Recap from Last Time
A TM subroutine that sorts all the 0s and 1s in a string.
The *Church-Turing Thesis* claims that every effective method of computation is either equivalent to or weaker than a Turing machine.

“This is not a theorem – it is a falsifiable scientific hypothesis. And it has been thoroughly tested!”

- Ryan Williams
Regular Languages

CFLs

Problems Solvable by Any Feasible Computing Machine

All Languages
Very Important Terminology

- Let $M$ be a Turing machine and let $w$ be a string.
- $M$ accepts $w$ if it enters an accept state when run on $w$.
- $M$ rejects $w$ if it enters a reject state when run on $w$.
- $M$ loops infinitely on $w$ (or just loops on $w$) if when run on $w$ it enters neither an accept nor a reject state.
- $M$ does not accept $w$ if it either rejects $w$ or loops infinitely on $w$.
- $M$ does not reject $w$ if it either accepts $w$ or loops on $w$.
- $M$ halts on $w$ if it accepts $w$ or rejects $w$. 
The Language of a TM

- The language of a Turing machine $M$, denoted $\mathcal{L}(M)$, is the set of all strings that $M$ accepts:
  $$\mathcal{L}(M) = \{ \, w \in \Sigma^* \mid M \text{ accepts } w \, \}$$
- For any $w \in \mathcal{L}(M)$, $M$ accepts $w$.
- For any $w \notin \mathcal{L}(M)$, $M$ does not accept $w$.
  - $M$ might reject $w$, or it might loop on $w$.
- A language is called **recognizable** if it is the language of some TM.
- A TM $M$ where $\mathcal{L}(M) = L$ is called a **recognizer** for $L$.
- Notation: the class $\text{RE}$ is the set of all recognizable languages.
  $$L \in \text{RE} \iff L \text{ is recognizable}$$
What do you think? Does that correspond to what you think it means to solve a problem?
Deciders

- Some Turing machines always halt; they never go into an infinite loop.
- If $M$ is a TM and $M$ halts on every possible input, then we say that $M$ is a **decider**.
- For deciders, accepting is the same as not rejecting and rejecting is the same as not accepting.

\[\begin{array}{c}
\text{halts (always)} \\
\begin{array}{c}
\text{Accept} \\
\text{Reject}
\end{array}
\end{array}\]

- does not reject
- does not accept
Decidable Languages

- A language $L$ is called **decidable** if there is a decider $M$ such that $\mathcal{L}(M) = L$.

- Equivalently, a language $L$ is decidable if there is a TM $M$ such that
  - If $w \in L$, then $M$ accepts $w$.
  - If $w \notin L$, then $M$ rejects $w$.

- The class $\mathbf{R}$ is the set of all decidable languages.
  \[ L \in \mathbf{R} \iff L \text{ is decidable} \]

- Decidable problems, in some sense, problems that can definitely be “solved” by a computer.
New Stuff!
What are $R$ and $RE$?
A Feel for $\mathbb{R}$ and RE

• You want to see if the hailstone sequence terminates for some $n \in \mathbb{N}$.

• An RE perspective:
  • Run the hailstone sequence on $n$. If it stops, output “yes.” But if the hailstone sequence doesn’t terminate, you’ll never learn this.

• An R perspective:
  • Perform some calculation on the number $n$ that determines whether the hailstone sequence terminates, but without actually running the hailstone sequence. (*Can we do this?*)
A Feel for R and RE

- You have a DFA. You want to see if the DFA accepts any strings of the form $a^n b^n$.

Wait – didn’t we prove that this will never happen?

Explain how it’s possible for a DFA to accept a string of this form given what we’ve proved about nonregular languages.
A Feel for $R$ and $RE$

- You have a DFA. You want to see if the DFA accepts any strings of the form $a^n b^n$.

- An $RE$ perspective:
  - Run the DFA on $a^0 b^0$, $a^1 b^1$, $a^2 b^2$, etc. If the DFA ever accepts, return true. But if not, you may never learn this.

- An $R$ perspective:
  - Look at the structure of the DFA and, somehow, determine whether it accepts any strings of this form, but without running the DFA on all of them. (*Can we do this?*)
A Feel for **R** and **RE**

- You have a program. You want to see if the program crashes when run on any input.

- An **RE** perspective:
  - Run the program on every possible input. If you see it crash, return true. If it never crashes, you will never learn this.

- An **R** perspective:
  - Look at the source code and somehow determine, with 100% certainty, whether the program will ever crash. *(Can we do this?)*
A Feel for R and RE

• You have an X. You want to see if there’s a Y where X and Y go well together.

• An RE perspective:
  • List off all the Y’s in some order and check if X and Y go well together. If so, return true. If not, you might not learn anything.

• An R perspective:
  • Look at X and, somehow, determine whether such a Y exists without checking all Y’s. (Can we do this?)
**Intuition 1:** Problems in **RE** are ones that can be approached by doing some sort of exhaustive search over a potentially infinite list of options.

**Intuition 2:** Problems in **R** are ones that can be solved without having to exhaustively try one on infinitely many possibilities.
**R and \(\mathbb{RE}\) Languages**

- Every decider is a Turing machine, but not every Turing machine is a decider.
- This means that \(\mathbb{R} \subseteq \mathbb{RE}\).
- Hugely important theoretical question:

\[
\mathbb{R} \quad ? \quad \mathbb{RE}
\]

- That is, if you can just confirm “yes” answers to a problem, can you necessarily solve that problem?
Which Picture is Correct?
Which Picture is Correct?
Strings, Languages, and Encodings
What problems can we solve with a computer?  

What is a "problem?"
Decision Problems

• A *decision problem* is a type of problem where the goal is to provide a yes or no answer.

• Example: Bin Packing
  
  You're given a list of patients who need to be seen and how much time each one needs to be seen for. You're given a list of doctors and how much free time they have. Is there a way to schedule the patients so that they can all be seen?

• Example: Dominating Set Problem
  
  You're given a transportation grid and a number $k$. Is there a way to place emergency supplies in at most $k$ cities so that every city either has emergency supplies or is adjacent to a city that has emergency supplies?
A Model for Solving Problems

Computational Device

input

Yes

No
A Model for Solving Problems

input

Computational Device

Yes

No
A Model for Solving Problems

input

Computational Device

Yes

No
A Model for Solving Problems

input

Computational Device

Yes

No
A Model for Solving Problems

input

Turing Machine

Yes

No
A Model for Solving Problems

Turing Machine

Input

Yes (accept)

No (reject)
A Model for Solving Problems

How do we represent our inputs?
Humbling Thought:

*Everything on your computer is a string over \{0, 1\}.*
Strings and Objects

- Think about how my computer encodes the image on the right.
- Internally, it's just a series of zeros and ones sitting on my hard drive.
Strings and Objects

- A different sequence of 0s and 1s gives rise to the image on the right.
- Every image can be encoded as a sequence of 0s and 1s, though not all sequences of 0s and 1s correspond to images.
Object Encodings

- If $Obj$ is some mathematical object that is *discrete* and *finite*, then we’ll use the notation $\langle Obj \rangle$ to refer to some way of encoding that object as a string.

- Think of $\langle Obj \rangle$ like a file on disk – it encodes some high-level object as a series of characters.

\[ \langle 11011100101110111100010011...110 \rangle = 11011100101110111100010011...110 \]
Object Encodings

• If Obj is some mathematical object that is *discrete* and *finite*, then we’ll use the notation \( \langle \text{Obj} \rangle \) to refer to some way of encoding that object as a string.

• Think of \( \langle \text{Obj} \rangle \) like a file on disk – it encodes some high-level object as a series of characters.

\[
\langle \rangle = 00110101000101000101000100\ldots001
\]
Object Encodings

- **Great intuition:** If you can store an object as a file on disk, then you can encode it as a string.

- Here are a bunch of different types of objects. Which of these objects can *always* be encoded as a string?
  - A DFA over the alphabet \{a, b\}.
  - A regular expression.
  - A subset of \{a, b\}*
  - A binary relation over \[[n]\], for some \(n \in \mathbb{N}\).
  - A graph whose nodes are \[[n]\], for some \(n \in \mathbb{N}\).
Object Encodings

- **Great intuition:** If you can store an object as a file on disk, then you can encode it as a string.

- Here are a bunch of different types of objects. Which of these objects can *always* be encoded as a string?
  - A DFA over the alphabet \{a, b\}.
  - A regular expression.
  - A subset of \{a, b\}*. 
  - A binary relation over \{n\}, for some \(n \in \mathbb{N}\).
  - A graph whose nodes are \{n\}, for some \(n \in \mathbb{N}\).

Yep! That’s what our DFA/NFA tool does.
Object Encodings

- **Great intuition:** If you can store an object as a file on disk, then you can encode it as a string.
- Here are a bunch of different types of objects. Which of these objects can *always* be encoded as a string?
  - A DFA over the alphabet \(\{a, b\}\).
  - A regular expression.
  - A subset of \(\{a, b\}^*\).
  - A binary relation over \([n]\), for some \(n \in \mathbb{N}\).
  - A graph whose nodes are \([n]\), for some \(n \in \mathbb{N}\).
Object Encodings

• **Great intuition:** If you can store an object as a file on disk, then you can encode it as a string.

• Here are a bunch of different types of objects. Which of these objects can always be encoded as a string?
  
  A DFA over the alphabet \{a, b\}.
  A regular expression.
  
  • **A subset of \{a, b\}**.
  A binary relation over \([n]\), for some \(n \in \mathbb{N}\).
  A graph whose nodes are \([n]\), for some \(n \in \mathbb{N}\).
Object Encodings

• Great intuition: If you can store an object as a file on disk, then you can encode it as a string.

• Here are a bunch of different types of objects. Which of these objects can always be encoded as a string?
  
  A DFA over the alphabet \{a, b\}.
  A regular expression.
  
  • A subset of \{a, b\}*. 
  A binary relation over \[n\], for some \(n \in \mathbb{N}\).
  A graph whose nodes are \[n\], for some \(n \in \mathbb{N}\).
Object Encodings

- **Great intuition:** If you can store an object as a file on disk, then you can encode it as a string.

- Here are a bunch of different types of objects. Which of these objects can *always* be encoded as a string?
  - A DFA over the alphabet \{a, b\}.
  - A regular expression.
  - A subset of \{a, b\}*.  
  - A binary relation over \([n]\), for some \(n \in \mathbb{N}\).
  - A graph whose nodes are \([n]\), for some \(n \in \mathbb{N}\).

Yep! Write down \(n\) and a list of pairs of numbers where the relation holds.
Object Encodings

• **Great intuition:** If you can store an object as a file on disk, then you can encode it as a string.

• Here are a bunch of different types of objects. Which of these objects can *always* be encoded as a string?
  
  A DFA over the alphabet \{a, b\}.
  A regular expression.
  A subset of \{a, b\}^*.
  A binary relation over \([n]\), for some \(n \in \mathbb{N}\).

• A graph whose nodes are \([n]\), for some \(n \in \mathbb{N}\).
Object Encodings

• For the purposes of what we’re going to be doing, we aren’t going to worry about exactly how objects are encoded.

• For example, we can say \(\langle 137 \rangle\) to mean “some encoding of 137” without worrying about how it’s encoded.

  • Analogy: do you need to know how the \texttt{int}\ type is represented in C++ to do basic C++ programming? That’s more of a CS107 question.

• We’ll assume, whenever we’re dealing with encodings, that some Smart, Attractive, Witty person has figured out an encoding system for us and that we’re using that encoding system.
Encoding Groups of Objects

• Given a group of objects $Obj_1, Obj_2, \ldots, Obj_n$, we can create a single string encoding all these objects.
  • Think of it like a .zip file, but without the compression.

• We'll denote the encoding of all of these objects as a single string by $\langle Obj_1, \ldots, Obj_n \rangle$.
  • This lets us feed multiple inputs into our computational device at the same time.
A Model for Solving Problems

Turing Machine

(input)
A Model for Solving Problems

Turing Machine

(input string) (probably encoded)

(accept)

Yes

(reject)

No
What problems can we solve with a computer?
Time-Out for Announcements!
Second Midterm Graded

• We’ve finished grading the second midterm exam. We sent out emails with scores last night, and the exams themselves are up on GradeScope.
  • No need to grab a paper copy – thanks to John and Fei for making this happen!
• Solutions and statistics are up on the course website.
• We’ll discuss regrade procedures on Wednesday.
Problem Set Eight

• Problem Set Eight is due this Friday at 2:30PM.
  • Again, you are encouraged to ask questions. Stop by our office hours, or visit Piazza!
• Problem Set Seven is being graded right now. Solutions are up on the course website.
Your Questions
Your Questions

Next time!
Back to CS103!
Emergent Properties
Emergent Properties

• An **emergent property** of a system is a property that arises out of smaller pieces that doesn't seem to exist in any of the individual pieces.

• Examples:
  
  • Individual neurons work by firing in response to particular combinations of inputs. Somehow, this leads to consciousness, love, and ennui.

  • Individual atoms obey the laws of quantum mechanics and just interact with other atoms. Somehow, it's possible to combine them together to make iPhones and pumpkin pie.
Emergent Properties of Computation

- All computing systems equal to Turing machines exhibit several surprising emergent properties.
- If we believe the Church-Turing thesis, these emergent properties are, in a sense, “inherent” to computation. Computation can’t exist without them.
- These emergent properties are what ultimately make computation so interesting and so powerful.
- As we'll see, though, they're also computation's Achilles heel – they're how we find concrete examples of impossible problems.
Two Emergent Properties

- There are two key emergent properties of computation that we will discuss:
  - **Universality**: There is a single computing device capable of performing any computation.
  - **Self-Reference**: Computing devices can ask questions about their own behavior.
- As you'll see, the combination of these properties leads to simple examples of impossible problems and elegant proofs of impossibility.
Universal Machines
An Observation

• When we've been discussing Turing machines, we've talked about designing specific TMs to solve specific problems.

• Does this match your real-world experiences? Do you have one computing device for each task you need to perform?
Can we make a “reprogrammable Turing machine?”
A TM Simulator

- It is possible to program a TM simulator on an unbounded-memory computer.
  - In fact, we did this! It's on the CS103 website.
- We could imagine it as a method
  ```java
  boolean simulateTM(TM M, string w)
  ```
with the following behavior:
  - If $M$ accepts $w$, then `simulateTM(M, w)` returns `true`.
  - If $M$ rejects $w$, then `simulateTM(M, w)` returns `false`.
  - If $M$ loops on $w$, then `simulateTM(M, w)` loops infinitely.
A TM Simulator

- It is known that anything that can be done with an unbounded-memory computer can be done with a TM.
A TM Simulator

- It is known that anything that can be done with an unbounded-memory computer can be done with a TM.

- This means that there must be some TM that has the behavior of this simulateTM method.
A TM Simulator

• It is known that anything that can be done with an unbounded-memory computer can be done with a TM.

• This means that there must be some TM that has the behavior of this `simulateTM` method.

• What would that look like?
A TM Simulator

- It is known that anything that can be done with an unbounded-memory computer can be done with a TM.
- This means that there must be some TM that has the behavior of this simulateTM method.
- What would that look like?

Diagram:
- TM that runs other TMs
- Input...
- $M$
- $w$
- accept!
- reject!
- (loop)
A TM Simulator

- It is known that anything that can be done with an unbounded-memory computer can be done with a TM.
- This means that there must be some TM that has the behavior of this simulateTM method.
- What would that look like?
The Universal Turing Machine

- **Theorem (Turing, 1936):** There is a Turing machine $U_{TM}$ called the universal Turing machine that, when run on an input of the form $\langle M, w \rangle$, where $M$ is a Turing machine and $w$ is a string, simulates $M$ running on $w$ and does whatever $M$ does on $w$ (accepts, rejects, or loops).

- The observable behavior of $U_{TM}$ is the following:
  - If $M$ accepts $w$, then $U_{TM}$ accepts $\langle M, w \rangle$.
  - If $M$ rejects $w$, then $U_{TM}$ rejects $\langle M, w \rangle$.
  - If $M$ loops on $w$, then $U_{TM}$ loops on $\langle M, w \rangle$.
Imagine you have some machine $M$ (like a program) that you want to run on input $w$. 

**Input $w$**

... a a a a a a ...

$U_{TM}$, Schematically

**Machine $M$**

Imagine you have some machine $M$ (like a program) that you want to run on input $w$. 

**Input $w$**

... a a a a a a ...

$U_{TM}$, Schematically
$U_{TM}$, Schematically

Machine $M$

Input $w$

Take $M$ and write it down as a string (think like encoding the finite state control as a table)

Slides by Amy Liu.
**U_{TM}, Schematically**

**Machine M**

- **Start State:** $q_0$
- **Accept State:** $q_{acc}$
- **Reject State:** $q_{rej}$

Input $w$

- $\ldots a \ a \ a \ a \ a \ a \ldots$

- $\ldots q_0 \ a \ \Box \ R \ldots q_1 \ a \ldots M \ldots$

Take $M$ and write it down as a string (think like encoding the finite state control as a table)

**Slides by Amy Liu.**
$U_{TM}$, Schematically

Machine $M$

Input $w$

Now take your input $w$ and write it down too.

Slides by Amy Liu.
$U_{TM}$, Schematically

Machine $M$

Now take your input $w$ and write it down too.

Input $w$

$\ldots a a a a a \ldots$

$\cdots q_0 a \square R \cdots q_1 a \cdots a a a a a \cdots$

$M \quad w$
$U_{TM}$, Schematically

Machine $M$

Feed this into $U_{TM}$.
$U_{TM}$, Schematically

Machine $M$

- Start state: $q_0$
- Transition: $a \rightarrow \square, R$
- $q_1$
- Transition: $a \rightarrow \square, R$
- Rejected state: $q_{rej}$
- Accepting state: $q_{acc}$

Input $w$

- Input: $\ldots a a a a a \ldots$

Input $\langle M, w \rangle$

- Input: $\ldots q_0 a \square R \ldots q_1 a \ldots a a a a a \ldots$

$M$ and $w$
$U_{\text{TM}}, \text{ Schematically}$

**Machine $M$**

- **Start State**: $q_0$
- **Accept State**: $q_{\text{acc}}$
- **Reject State**: $q_{\text{rej}}$

**Input $w$**

- $\cdots a\ a\ a\ a\ a\ \cdots$

**Input $\langle M, w \rangle$**

- $\cdots q_0\ a\ \square\ R\ \cdots q_1\ a\ \cdots a\ a\ a\ a\ a\ \cdots$

- **Update State and Tape**: $M$ and $w$

- **Look at Next Char of $w$**: if $M$ is in accepting state

- **Look Up What $M$ Should Do Upon Reading $w$**: if $M$ is in rejecting state

- **Update State and Tape**: $q_{\text{rej}}$

- **Start**: $q_{\text{acc}}$

**Slides by Amy Liu.**
$U_{TM}$, Schematically

**Machine $M$**

- **Start State**: $q_0$
- **Accepting State**: $q_{acc}$
- **Rejecting State**: $q_{rej}$

- **Transition Rules**:
  - $\square \rightarrow \square, R$
  - $a \rightarrow \square, R$
  - $a \rightarrow , R$
  - $\square \rightarrow \square, R$

**Input $w$**

```
... a a a a a ...
```

**Input $(M, w)$**

```
... q_0 a \square R ... q_1 a ... a a a a a ...
```

- **Start State**: $q_0$
- **Accepting State**: $q_{acc}$
- **Rejecting State**: $q_{rej}$

- **Transition Rules**:
  - Look at next char of $w$
  - Look up what $M$ should do upon reading $w$
  - Update state and tape

**Notes**

- If $M$ is in accepting state:
  - $q_{acc}$

- If $M$ is in rejecting state:
  - $q_{rej}$

*Slides by Amy Liu.*
U_{TM}, Schematically

Machine M

Input w

... a a a a a ...

Input \langle M, w \rangle

... q_0 a \square R ... q_1 a ... a a a a a ...

Slides by Amy Liu.
$U_{TM}$, Schematically

Machine $M$

- **Start state**: $q_0$
- **Transition 1**: $a \rightarrow \square, R$
- **Transition 2**: $\square \rightarrow \square, R$
- **Transition 3**: $a \rightarrow , R$
- **Transition 4**: $\square \rightarrow \square, R$
- **Accepting state**: $q_{acc}$
- **Rejecting state**: $q_{rej}$

Input $w$

```
... a a a a a ...
```

Input $\langle M, w \rangle$

```
... $q_0$ a $\square$ R ... $q_1$ a ... a a a a a ...
```

$U_{TM}$

- **Start state**: $q_{acc}$
- **Transition**: Look at next char of $w$
- **If $M$ is in accepting state**: $q_{acc} \rightarrow q_{acc}$
- **If $M$ is in rejecting state**: $q_{rej} \rightarrow q_{rej}$
- **Update state and tape**: Look up what $M$ should do upon reading $w$

Slides by Amy Liu.
$U_{TM}$, Schematically

**Machine $M$**

- **Start state**: $q_0$
- **Accepting state**: $q_{acc}$
- **Rejecting state**: $q_{rej}$
- Transition rules:
  - $\Box \rightarrow \Box, R$
  - $a \rightarrow \Box, R$
  - $\Box \rightarrow \Box, R$

**Input $w$**

- $\ldots a a a a a \ldots$

**Input $\langle M, w \rangle$**

- $\ldots q_0 a \Box R \ldots q_1 a \ldots a a a a a \ldots$

---

**$U_{TM}$**

- **Start state**: $q_{acc}$
- **Accepting state**: $q_{acc}$
- **Rejecting state**: $q_{rej}$
- Transition rules:
  - Look at next char of $w$
  - if $M$ is in accepting state
  - Look up what $M$ should do upon reading $w$
  - if $M$ is in rejecting state
  - Update state and tape

Slides by Amy Liu.
$U_{TM}$, Schematically

**Machine $M$**

- Start at state $q_0$.
- Transition: $q_0 \xrightarrow{a} q_1 \xrightarrow{\square} q_{\text{rej}}$.
- Acceptance: $q_{\text{acc}}$.
- Rejection: $q_{\text{rej}}$.

**$U_{TM}$**

- Input: $\langle M, w \rangle$.
- Transition: $q_0 \xrightarrow{a} q_1 \xrightarrow{\square} \cdots$.
- Acceptance: $q_{\text{acc}}$.
- Rejection: $q_{\text{rej}}$.

- Update state and tape:
  - Look at next char of $w$.
  - Look up what $M$ should do upon reading $w$.
  - Update state and tape.

**Input $w$**

- $\cdots a a a a a \cdots$

**Input $\langle M, w \rangle$**

- $\cdots \begin{array}{c} q_0 \ \square \ \square \ \cdots \ q_1 \ a \ \cdots \ a \ a \ a \ a \ a \ a \ a \ a \ a \ a \ a \ \cdots \end{array}$

**Slides by Amy Liu.**
$U_{TM}$, Schematically

Machine $M$

Input $w$

Input $\langle M, w \rangle$

Slides by Amy Liu.
$U_{TM}$, Schematically

Machine $M$

- $\square \rightarrow \square, R$
- $a \rightarrow \square, R$
- $a \rightarrow , R$
- $\square \rightarrow \square, R$

Start state: $q_0$

$\text{Input } w$

```
... a a a a ...
```

$U_{TM}$

- Look at next char of $w$
- if $M$ is in accepting state
  - Update state and tape
  - $q_{acc}$
- if $M$ is in rejecting state
  - $q_{rej}$
  - Look up what $M$ should do upon reading $w$

Input $\langle M, w \rangle$

```
... q_0 a \square R ... q_1 a ... a a a a ...
```

$M$

$w$

Slides by Amy Liu.
U_{TM}, Schematically

Machine $M$

$q_{acc}$

$\square \rightarrow \square, R$

$a \rightarrow \square, R$

$q_0$

Start

$a \rightarrow , R$

$a \rightarrow \square, R$

$q_1$

$q_{rej}$

Input $w$

Input $\langle M, w \rangle$

... a a a a ...

... $q_0$ a $\square$ R ... $q_1$ a ...

$a$ $a$ $a$ $a$ ...

Look at next char of $w$

if $M$ is in accepting state

Look up what $M$ should do upon reading $w$

if $M$ is in rejecting state

Update state and tape

Slides by Amy Liu.
$U_{TM}$, Schematically

Machine $M$

Input $w$

Input $\langle M, w \rangle$

Slides by Amy Liu.
Since $U_{TM}$ is a TM, it has a language.

What is the language of the universal Turing machine?
The Language of $U_{TM}$

- Recall that the language of a TM is the set of all strings that TM accepts.
- $U_{TM}$, when run on a string $\langle M, w \rangle$, where $M$ is a TM and $w$ is a string, will
  - accept $\langle M, w \rangle$ if $M$ accepts $w$,
  - reject $\langle M, w \rangle$ if $M$ rejects $w$, and
  - loop on $\langle M, w \rangle$ if $M$ loops on $w$. 

\[ \mathcal{L}(U_{TM}) = \{ \langle M, w \rangle \mid M \text{ is a TM and } w \in \mathcal{L}(M) \} \]
The Language of $U_{TM}$

- Recall that the language of a TM is the set of all strings that TM accepts.
- $U_{TM}$, when run on a string $(M, w)$, where $M$ is a TM and $w$ is a string, will
  
  ... accept $(M, w)$ if $M$ accepts $w$,
  ... reject $(M, w)$ if $M$ rejects $w$, and
  ... loop on $(M, w)$ if $M$ loops on $w$.

\[
L(U_{TM}) = \{ (M, w) \mid M \text{ is a TM and } M \text{ accepts } w \}
= \{ (M, w) \mid M \text{ is a TM and } w \in L(M) \} \]
The Language $A_{\text{TM}}$

- The *acceptance language for Turing machines*, denoted $A_{\text{TM}}$, is the language of the universal Turing machine:

$$A_{\text{TM}} = \mathcal{L}(U_{\text{TM}}) = \{ \langle M, w \rangle \mid M \text{ is a TM and } M \text{ accepts } w \}$$

- Useful fact:

  $$\langle M, w \rangle \in A_{\text{TM}} \iff M \text{ accepts } w.$$  

- Because $A_{\text{TM}} = \mathcal{L}(U_{\text{TM}})$, we know that $A_{\text{TM}} \in \text{RE}$.  

Great Question to Ponder

• Simplify this expression:
  \[
  \langle \langle U_{TM}, \langle U_{TM}, \langle U_{TM}, \langle U_{TM}, \langle M, w \rangle \rangle \rangle \rangle \rangle \rangle \in A_{TM}.
  \]

• If you can do this, you probably understand how things fit together.

• If you’re having trouble, no worries! It might be easier to start with this expression:
  \[
  \langle U_{TM}, \langle M, w \rangle \rangle \in A_{TM}.
  \]
Uh... so what?
Reason 1: *It has practical consequences.*
Why Does This Matter?

- The existence of a universal Turing machine has both theoretical and practical significance.
- For a practical example, let's review this diagram from before.
- Previously we replaced the computer with a TM. (This gave us the universal TM.)
- What happens if we replace the TM with a computer program?

![Diagram](image)
Why Does This Matter?

- The existence of a universal Turing machine has both theoretical and practical significance.
- For a practical example, let's review this diagram from before.
- Previously we replaced the computer with a TM. (This gave us the universal TM.)
- What happens if we replace the TM with a computer program?

```java
for (int i = 2; i < n; i++) {
    if (n % i == 0) {
        ...input...
    }
}
```

true!
false!
The fact that there’s a universal TM, combined with the fact that computers can simulate TMs and vice-versa, means that it’s possible to write a program that simulates other programs.

These programs go by many names:

- An **interpreter**, like the Java Virtual Machine or most implementations of Python.
- A **virtual machine**, like VMWare or VirtualBox, that simulates an entire computer.
Why Does This Matter?

- The key idea behind the universal TM is that idea that TMs can be fed as inputs into other TMs.
  - Similarly, an interpreter is a program that takes other programs as inputs.
  - Similarly, an emulator is a program that takes entire computers as inputs.
- This hits at the core idea that **computing devices can perform computations on other computing devices.**
Reason 2: *It’s philosophically interesting.*
Can Computers Think?

• On May 15, 1951, Alan Turing delivered a radio lecture on the BBC on the topic of whether computers can think.

• He had the following to say about whether a computer can be thought of as an electric brain...
“In fact I think [computers] could be used in such a manner that they could be appropriately described as brains. I should also say that

‘If any machine can be appropriately described as a brain, then any digital computer can be so described.’

This last statement needs some explanation. It may appear rather startling, but with some reservations it appears to be an inescapable fact.

It can be shown to follow from a characteristic property of digital computers, which I will call their **universality**. A digital computer is a universal machine in the sense that it can be made to replace any machine of a certain very wide class. It will not replace a bulldozer or a steam-engine or a telescope, but it will replace any rival design of calculating machine, that is to say any machine into which one can feed data and which will later print out results. In order to arrange for our computer to imitate a given machine it is only necessary to programme the computer to calculate what the machine in question would do under given circumstances, and in particular what answers it would print out. The computer can then be made to print out the same answers.

If now some machine can be described as a brain we have only to programme our digital computer to imitate it and it will also be a brain.”
Next Time

• **Self-Reference**
  • Turing machines that compute on themselves!

• **Undecidable Problems**
  • Problems truly beyond the limits of algorithmic problem-solving!

• **Consequences of Undecidability**
  • Why does any of this matter outside of a computer science course?