Welcome, virtual picket-line crossers.
Assignment 7

Graded soon

Assignment 8

Due today; corrections due Thursday

Exam 2

Graded someday, someday
A plea
“What problems can we solve with a computer?”
“What problems can we solve with a computer?”

What kind of computer?
The *Church–Turing thesis* states:

Every effective method of computation is either equivalent to or weaker than a Turing machine.
Problems solvable by any feasible computing machine

Regular languages

Context-free languages

All languages
Problems solvable by Turing machines

Regular languages
Context-free languages

All languages
“What problems can we solve with a computer?”

What does it mean to “solve” a problem?
Does not reject

Does not accept

Accept

Loop

Reject

Halts
A language $L$ is **Turing-recognizable** if there is a TM $M$ such that

$$\forall w \in \Sigma^* . (M \text{ accepts } w \iff w \in L)$$

This is a “weak” notion of solving a problem:

- If $w \in L$, then $M$ accepts $w$.
- If $w \notin L$, then $M$ does not accept $w$.
  
  It might reject or it might loop forever.

The class **RE** consists of all Turing-recognizable languages.
If a Turing machine $M$ halts on every possible input – i.e., it never goes into an infinite loop – then we call $M$ a *decider*.

For deciders, accepting is the same as not rejecting and rejecting is the same as not accepting:
A language $L$ is **Turing-decidable** if there is a TM $M$ such that

$$\forall w \in \Sigma^* . \ (M \text{ accepts } w \iff w \in L) \land \ (M \text{ halts on all inputs}).$$

This is a “strong” notion of solving a problem:

- If $w \in L$, then $M$ accepts $w$.
- If $w \not\in L$, then $M$ rejects $w$.

The class $\mathcal{R}$ consists of all Turing-decidable languages.
[Exciting new material starts now!]
R matters because it is exactly the class of languages for which there is an algorithm to decide if a string is in the language.

By the Church–Turing thesis, this isn’t just about Turing machines.

If there is any algorithm to decide membership in the language, then there is a decider for it.
A feel for $R$ and $RE$

You have a DFA. You want to see if the DFA accepts any strings of the form $a^nb^n$. 
A feel for $R$ and $RE$

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

Not whether the language is $a^n b^n$; that's impossible for a DFA!
A feel for \textbf{R} and \textbf{RE}

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

\textbf{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.

\textbf{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. \textit{(Can we do this?)}
A feel for R and RE

Say you’re working on a computer science assignment. You wonder if there’s any input that will make your program crash.

An RE perspective: Try running 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 infinitely many possibilities.
Every decider is a Turing machine, but not every Turing machine is a decider.

So, \( R \subseteq \text{RE} \).

But is \( R = \text{RE} \)?

That is, if you can confirm “yes” answers to a problem, can you also solve that problem?
Is this right?

Regular languages

Context-free languages

R

RE

All languages
Or this?

Regular languages

Context-free languages

R

RE

All languages
“What problems can we solve with a computer?”

What is a “problem”?
A **decision problem** is a type of problem where the goal is answer yes or no.

**Example: Bin Packing**

You’re given a list of patients who need to be seen and how much time each 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: Route Planning**

You’re given a transportation grid of a city, a start location, a destination location, and information about the traffic over the course of the day. Given a time limit $T$, is there a way to drive from the start location to the end location in at most $T$ hours?
input

Computational device

Yes

No
How do we represent the inputs?
Two symbols should be enough for anyone.
Everything on your computer is a string over \( \{0, 1\} \).

For instance, every image can be encoded as a sequence of 0s and 1s (though not every sequence of 0s and 1s corresponds to an image)!
Generally speaking, if $Obj$ is some discrete, finite mathematical object, then we’ll use the notation $\langle Obj \rangle$ to refer to some reasonable encoding of that object as a string of characters.

$\langle \quad \rangle = 110011010001011110100101\ldots$
Object encodings

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

Generally, we'll assume that some clever person has already figured out a way to encode what we want, and we can just say, e.g., ⟨137⟩ to mean “some encoding of 137” without worrying about how it’s encoded.

By analogy, consider whether you need to know how the int type is represented in C to do basic C programming.
Caveat

Remember: *discrete* and *finite*! Some things can’t be encoded as strings.

There’s no general way to encode *real numbers* as strings.

Imagine a real number generated by tossing infinitely many coins, one for each digit, heads = 0, tails = 1.

There’s no general way to encode *languages* as strings.

Imagine tossing a coin for each string. Heads = the string is in the language. Tails = the string is not in the language.
Encoding groups of objects

Given a finite group of objects, $Obj_1, Obj_2, \ldots, Obj_n$, we can create a single string encoding all of these objects.

Think of it like a .tar file (or a .zip file without the compression).

We can denote the encoding of all of these objects as a single string by $\langle Obj_1, Obj_2, \ldots, Obj_n \rangle$.

This lets us feed multiple inputs into our computational device at the same time.
Now we can ask “what’s 5 + 3?” by asking for each possible $x$ whether $\langle 5, 3, x \rangle$ is in the language of sums.
A Turing machine receives input that is appropriately encoded.

The Turing machine processes the input and decides whether to accept or reject it.
Our goal is to speak of *computers solving problems*. We model this by looking at *Turing machines recognizing languages*. By turning any problem into an equivalent *decision problem*, this precisely captures what we’re interested in.
“What problems can we solve with a computer?”
We haven’t answered this question yet, but we’re getting closer.

To get to the answer, we’re going to need to step back for a moment.
Let’s think about *emergent properties*.  

An emergent property of a system is a property that arises out of smaller pieces but which doesn’t seem to exist in any of the individual pieces. E.g.,

Individual neurons fire in response to particular combinations of inputs and this gives rise to human consciousness.  
Individual atoms obey the laws of quantum mechanics and just interact with other atoms, and this gives rise to literally everything.
All computing systems equal to Turing machines exhibit several surprising emergent problems.

Because of the Church–Turing thesis, these must be inherent to computation; computation can’t exist without them. They are what ultimately make computation so interesting and powerful. But they’re also computation’s Achilles heel – they’re how we find concrete examples of impossible problems.
The two emergent properties of computation that we’ll discuss are:

*Universality*: There is a single computing device capable of performing any computation.

*Self-reference*: Computing devices can ask questions about their own behavior.

The combination of these properties leads to simple examples of impossible problems and elegant proofs of impossibility.
Emergent property: *Universality*
A central idea in the theory of computation is that of a *universal computer* – a computer powerful enough to simulate any other computing device.
The idea of a universal computer was described by Turing in 1937.

Like many computing pioneers, Turing was interested in the problem of making a computer that could think. Towards this end, he invented a scheme for a general-purpose computing machine.
Turing referred to his imaginary construct as a “universal machine” since at the time “computer” still meant a person – usually a woman – who performed computations.
We’ve been designing Turing machines to solve specific problems.

Do you have a dedicated computer for each task you need to perform?

Your email computer and your word-processing computer and your cute-cat-picture computer?
Most computers we encounter in everyday life are universal computers.

With the right software and enough time and memory, any universal computer can simulate any other type of computer.

To have a real computer perform a particular task, we load a program into it and have the computer execute the program.
Can we make a “repro programmable Turing machine”?
A Turing machine simulator

It’s possible to program a Turing machine simulator on an unbounded memory computer.

If we accept some (vast) limits on the “infinite” tape, we can even do this on a real computer.
This simple Turing machine takes a string of A's and B's and rearranges them so that all the A's come first.

<table>
<thead>
<tr>
<th>State</th>
<th>Significance</th>
</tr>
</thead>
<tbody>
<tr>
<td>Start</td>
<td>No B's so far in sequence</td>
</tr>
<tr>
<td>1</td>
<td>At least one B so far</td>
</tr>
<tr>
<td>2</td>
<td>An A has been found after a B and has been changed to a B</td>
</tr>
<tr>
<td>3</td>
<td>Leftmost B is changed to A</td>
</tr>
</tbody>
</table>

TO RUN: enter a sequence of A's and B's, position read head on leftmost symbol, and start.
Sketch of a Turing machine simulator

```python
state = start
while True:
    if state.isAccepting():
        return True
    if state.isRejecting():
        return False
    c = tape.readSymbol()
    state, dir, out = state.next(c)
    tape.write(out)
    if dir == "left":
        tape.moveLeft()
    elif dir == "right":
        tape.moveRight()
```
A Turing machine simulator

While a simulator like this is an interactive tool to help us understand the theoretical model, we can also imagine it as a method

```c
bool simulateTM(TM M, string w)
```

with the following behavior:

- If \( M \) accepts \( w \), then \( \text{simulateTM}(M, w) \) returns \text{true}.
- If \( M \) rejects \( w \), then \( \text{simulateTM}(M, w) \) returns \text{false}.
- If \( M \) loops on \( w \), then \( \text{simulateTM}(M, w) \) loops infinitely.
simulateTM

true!

false!

(loop)

...input...
Anything that can be done with an unbounded-memory computer can be done with a Turing machine.

So there must be a Turing machine that has the behavior of the `simulateTM` method.
The diagram illustrates the operation of a Turing Machine (TM) that runs other TMs. The input \( w \) is processed by the TM, which includes a loop structure. The outcome of the computation can be either 'accept!', indicating acceptance of the input, or 'reject!', indicating rejection of the input. The 'TM that runs other TMs' box contains a diagram of a TM with a loop, highlighting the recursive nature of the computation.
THEOREM (Turing, 1936): There is a Turing machine $U$ called the \textit{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 $w$.

- If $M$ accepts $w$, then $U$ accepts $\langle M, w \rangle$.
- If $M$ rejects $w$, then $U$ rejects $\langle M, w \rangle$.
- If $M$ loops on $w$, then $U$ loops on $\langle M, w \rangle$. 
A universal machine
A universal machine

U

program
A universal machine
A universal machine

An encoding of some Turing machine $M$ we want to run
A universal machine

The input to that program is some string
A universal machine

The input has the form $\langle M, w \rangle$ where $M$ is some TM and $w$ is some string.
A universal machine

\[ \langle M, w \rangle \]
**Java program**: Solves one specific problem

**Turing machine**: Solves one specific problem

**Java simulator in Java**: Java program to simulate any Java program

**U**: Turing machine that can simulate any Turing machine
Since $U$ is a Turing machine, it has a language, $L(U)$. What is the language of the universal Turing machine?
The language of $U$

Recall that the language of a Turing machine is the set of all strings that the Turing machine accepts.

$U$, 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 $w$. 
The language of $U$

Recall that the language of a Turing machine is the set of all strings that the Turing machine accepts.

$U$, 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 $w$.

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

$= \{ \langle M, w \rangle \mid M \text{ is a TM and } w \in L(M) \}$
The language of $U$ is called $A_{TM}$:

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

$A_{TM}$ is the *acceptance language for Turing machines*. Because there is a Turing machine, $U$, that recognizes $A_{TM}$, we know $A_{TM} \in \text{RE}$.
Teaser: This language, $A_{TM}$, has some interesting properties beyond what we’ve seen.
Why do we care about universality?
Reason 1: Practical significance
What happens if we replace the Turing machine input with a normal computer program?
What happens if we replace the Turing machine input with a normal computer program?
Programs simulating programs

The fact that there’s a universal Turing machine 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 Python.
- A *virtual machine* like VMWare or VirtualBox that simulates an entire computer.
Party like it’s 1999 1990

What are the first names of the four other members in your party?

1. Alan
2. Turing
3. Alonzo
4. Church
5. Post_

(Enter names or press Enter)

The key idea behind the universal TM is 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: Philosophical interest
Can computers think?

On 15 May 1951, Alan Turing delivered a radio lecture on the BBC, where he argued that “it is not altogether unreasonable to describe digital computers as brains”.

Why would he think this, given the very limited abilities of computers of the time?
“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. If it is accepted that real brains, as found in animals, and in particular in men, are a sort of machine it will follow that our digital computer suitably programmed, will behave like a brain.”
“This argument involves several assumptions which can quite reasonably be challenged.”

http://www.turingarchive.org/browse.php/B/5
Next time

Self-reference

Turing machines that compute on themselves!

Undecidable problems

Problems truly beyond the scope of algorithmic problem-solving!

Consequences of undecidability

Why does any of this matter outside of a CS course?
Acknowledgments

This lecture incorporates material from:

W. Daniel Hillis
Nancy Ide
Keith Schwarz
Michael Sipser