Assignment 6 due on Tuesday
Optional programming assignments 2 & 3 out Friday
Exam 2 graded

Potential: Quiz on Turing machines on Tuesday
What problems can we solve with a computer?

What kind of computer?

A real computer has memory limitations: You have a finite amount of RAM, a finite amount of disk space, etc.

This makes every real computer equivalent to a (large) finite automaton.

However, as computers get more and more powerful, the amount of memory available keeps increasing.

An idealized computer is like a regular computer, but with unlimited RAM and disk space.

It functions just like a regular computer, but never runs out of memory.

Claim 1: Idealized computers can simulate Turing machines.

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

To simulate a Turing machine, the computer would need to be able to keep track of

- the finite-state control
- the current position
- the position of the tape head
- the tape contents

The tape contents are infinite, but that's because there are infinitely many blanks on both sides.

We only need to store the part of the tape that's been read from or written to so far.

<table>
<thead>
<tr>
<th>0</th>
<th>1</th>
<th>□</th>
</tr>
</thead>
<tbody>
<tr>
<td>q0</td>
<td>q1</td>
<td>□</td>
</tr>
<tr>
<td>q1</td>
<td>0</td>
<td>R</td>
</tr>
<tr>
<td>q2</td>
<td>q rej</td>
<td>□</td>
</tr>
<tr>
<td>q3</td>
<td>0</td>
<td>L</td>
</tr>
</tbody>
</table>
Simulating a Turing machine

To simulate a Turing machine, the computer would need to be able to keep track of
- the finite-state control
- the current position
- the position of the tape head
- the tape contents

The tape contents are infinite, but that's because there are infinitely many blanks on both sides.
We only need to store the part of the tape that's been read from or written to so far.

We've seen that Turing machines can
- implement loops
- make function calls (subroutines)
- keep track of natural numbers (written in unary or in decimal on the tape)
- perform elementary arithmetic (equality testing, addition, subtraction, increment, decrement)
- perform if/else tests (different transitions based on different cases)
- Maintain variables using different parts of the tape (e.g., the two numbers being added)

Claim 2: Turing machines can simulate idealized computers.

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

Anything you can do with a computer can be performed by a Turing machine

The resulting Turing machine might be very large, very slow, or both, but it would still faithfully simulate the computer.

This is true for any effective method of computation.
An *effective method of computation* is a form of computation with the following properties:

- The computation consists of a set of steps.
- There are fixed rules governing how one step leads to the next.
- Any computation that yields an answer does so in finitely many steps.
- Any computation that yields an answer always yields the correct answer.

This isn’t a formal definition, but it’s a set of properties we expect out of a computational system.

**The Church–Turing thesis**

Every effective method of computation is either equivalent to or weaker than a Turing machine.

We can consider many other reasonable models of computation: *DNA computing*, neural networks, quantum computing…
We can consider many other reasonable models of computation: DNA computing, *neural networks*, quantum computing…

We can also consider many variations on Turing machines, with multiple tapes, non-determinism, etc.

Experience has confirmed that every such model can be simulated by a standard Turing machine.
The Turing machine model intentionally embodies *implicit physical assumptions* to which all concrete computational processes are subject, e.g.,

1. *The speed of propagation of information is bounded.*
   Therefore the Turing machine can only move its read–write head to adjacent cells.

2. *The amount of information that can be encoded in the state of a finite system is bounded.*
   Therefore, Turing machines can only store one symbol per tape cell.

3. *It’s possible to construct physical devices that perform in a recognizable and reliable way logical functions like AND and OR.*
   Therefore we can fabricate a Turing machine out of physical parts and run it reliably.


As far as we know, no device built in the physical universe can have any more computational power than a Turing machine.

This is a remarkable statement, suggesting that a universal computer with proper programming should be able to simulate the function of a human brain.

In other words, AI is just a “small matter of programming”.
Turing machines \approx computers

Because Turing machines have the same computational power as regular computers, we can (essentially) reason about Turing machines by reasoning about actual computer programs.

Based on what's most convenient, we'll switch back and forth between Turing machines and computer programs – algorithms as high-level descriptions or pseudocode.

Unlike finite automata, which automatically halt after reading the input, Turing machines keep running until they explicitly enter an accept or reject state.

As such, it's possible for a Turing machine to run forever without accepting or rejecting.

“What problems can we solve with a computer?”

What does it mean to “solve” a problem?

If a Turing machine might run forever, how do we formally define what it means to “build a Turing machine for a language”?

What implications does this have for problem-solving?
## Terminology

Let $M$ be a Turing machine.

$M$ accepts a string $w$ if it enters an accept state when run on $w$.

$M$ rejects a string $w$ if it enters a reject state when run on $w$.

$M$ loops infinitely (or just loops) on a string $w$ if, when run on $w$, it never enters an accept or reject state.

### Diagram

- **Accept**
- **Loop**
- **Reject**

**Halts**

$M$ 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 $w$.

## Recognizable languages

The **language of a Turing machine** $M$,

$$L(M) = \{ w \in \Sigma^* \mid M \text{ accepts } w \}$$

If $w \in L(M)$, $M$ accepts $w$.

If $w \notin L(M)$, $M$ does not accept $w$.

That is, when $M$ is run on $w$, either it rejects or it loops forever.

A language is called **Turing-recognizable** (or just **recognizable**) if it is the language of some Turing machine.

A Turing machine $M$ where $L(M) = L$ is called a **recognizer** for $L$.

The set of all languages that are Turing-recognizable is called **RE**.

$$L \in \text{RE} \iff L \text{ is Turing-recognizable.}$$
Does this correspond to what you think it means to “solve a problem”?

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:

<table>
<thead>
<tr>
<th>Does not reject</th>
<th>Accept</th>
</tr>
</thead>
<tbody>
<tr>
<td>Does not accept</td>
<td>Reject</td>
</tr>
</tbody>
</table>

Halts (always)

Decidable languages

A language is called Turing-decidable (or just decidable) if it is the language of some decider.

Equivalently, a language $L$ is Turing-decidable if there is a Turing machine $M$ such that

- If $w \in L$, then $M$ accepts $w$.
- If $w \notin L$, then $M$ rejects $w$.

The set of all languages that are Turing-decidable is called $R$.

$L \in R \iff L$ is Turing-decidable.

Decidable problems – the languages in $R$ – are problems that can be “solved” by a computer.

(Though that solution isn’t guaranteed to be acceptably fast.)
All regular languages are in $R$.
We can use a Turing machine to simulate a DFA, and DFAs always halt.
$$\{0^n1^n \mid n \in \mathbb{N}_0\} \in R.$$ Proof: The Turing machine we built is a decider; it always halts.

In fact, all context-free languages are in $R$.
The proof of this is trickier. It relies on using CFGs rather than PDAs. See Sipser page 200.

Say you’re working on a computer science assignment. You wonder if your program has a bug.

**The RE perspective**: If you find a bug, you know the answer is yes. If you can’t find a bug, that doesn’t mean there isn’t one.

**The R perspective**: You know there is or isn’t. (A program that could do this would be magic.)

Every decider is a Turing machine, but not every Turing machine is a decider.

So, $R \subseteq RE$.

But is $R = RE$?
That is, if you can confirm “yes” answers to a problem, can you also solve that problem?

Is this right?
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?

Note: We're not asking what that way is.

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?
Two digits should be enough for anyone.

In other words, 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.

\[ \langle \rangle = 1100110100101110100101 \ldots \]
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 brilliant 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, Obj₁, Obj₂, …, Objₙ, 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 ⟨Obj₁, Obj₂, …, Objₙ⟩.

This lets us feed multiple inputs into our computational device at the same time.
Our goal is to speak of computers solving problems.

We model this by looking at Turing machines recognizing languages.

For decision problems that we're interested in solving, this precisely captures what we're interested in capturing.

“What problems can we solve with a computer?”

We haven’t answered this question yet, but we’re getting closer.
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.

- According to 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 “reprogrammable 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.
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 $true$.
- If $M$ rejects $w$, then $\text{simulateTM}(M, w)$ returns $false$.
- If $M$ loops on $w$, then $\text{simulateTM}(M, w)$ loops infinitely.

Sketch of a Turing machine simulator

```c
State state = start;
while (true) {
    if (state.isAccepting())
        return true;
    else if (state.isRejecting())
        return false;
    char c = tape.readSymbol();
    tape.write(state.symbolToWrite(c));
    state = state.next(c);
    if (state.isLeft())
        tape.moveLeft();
    else if (state.isRight())
        tape.moveRight();
}
```
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 Turing machine that runs other Turing machines is \(U_{TM}\), the \textit{Universal Turing machine}.

When \(U_{TM}\) is run on an input of the form \(\langle M, w \rangle\), where \(M\) is a Turing machine and \(w\) is a string, \(U_{TM}\) simulates \(M\) running on \(w\) and does whatever \(M\) does on \(w\).

- 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 \(\langle M, w \rangle\).

Acknowledgments
This lecture incorporates material from:

- W. Daniel Hillis
- Nancy Ide
- Keith Schwarz
- Michael Sipser