The Big Picture

2 December 2021
End of semester updates
End of semester updates

Exam 2 is graded.
End of semester updates

Exam 2 is graded.

Assignments graded soon!
End of semester updates

Exam 2 is graded.

Assignments graded soon!

Assignment 9 is due tomorrow; corrections due on Tuesday.
End of semester updates

Exam 2 is graded.

Assignments graded soon!

Assignment 9 is due tomorrow; corrections due on Tuesday.

We’ll fill out CEQs at the end of class today.
End of semester updates

Exam 2 is graded.

Assignments graded soon!

Assignment 9 is due tomorrow; corrections due on Tuesday.

We’ll fill out CEQs at the end of class today.

We’ll have an in-class review for Exam 3 on Tuesday.
End of semester updates
Thus, for any nondeterministic Turing machine $M$ that runs in some polynomial time $p(n)$, we can devise an algorithm that takes an input $w$ of length $n$ and produces $E_{M,w}$. The running time is $O(p^2(n))$ on a multitape deterministic Turing machine and...

WTF, man. I just wanted to learn how to program video games.
“In some way that I don’t understand, I’m glad that theorists are investigating the equivalence between five-dimensional Turing machines and Edward Scissorhands.”

James Mickens, “The Night Watch”, 2013
The big picture
What problems can be solved by computers?
First we need a definition of a computer!
I’m a computer. I compute.
We have a model of a computer.

We’re not sure what we can solve at this point, but we’ll call the languages we can capture this way the *regular languages*. 
What other machines can we make?
Nondeterminism! Is there any path through the NFA that leads to an accept state?
Wow – these new machines are way cooler than our old ones!
I wonder if they’re more powerful?
The subset construction lets us convert any NFA to a (big) equivalent DFA.
Wow – I guess not! That’s surprising.

So now we have a new way of modeling computers with finite memory!
I wonder how we can combine these machines together.
$M_1$ $M_2$

start
Cool – since we can glue machines together, we can glue languages together as well.
How are we going to do that?
matt@vassar.edu
matthew.vassar@vassar.edu
asprey@cs.vassar.edu
...

$a^+ (.a^+) @ a^+ (.a^+)^+$
Great – we’ve got a new way of describing languages.
So, what sorts of languages can we describe this way?
$\varepsilon R_1^* R_2 \Leftrightarrow$
Any regular expression can be systematically converted into an equivalent NFA and vice versa.
Awesome – we got back the exact same class of languages!
It seems like all our models give us the same power!
Did we get every language?
\[ L = \{a^i b^i \mid i \in \mathbb{N}_0\} \]
\[ L = \{a^i b^i \mid i \in \mathbb{N}_0\} \]

There's no way we can build a DFA for this; we'd need an infinite number of states.
I guess not.

We formalize the argument that a language isn’t regular using the *Pumping Lemma for Regular Languages*. 
I guess not.

We formalize the argument that a language isn’t regular using the **Pumping Lemma for Regular Languages**.
But we did learn something cool:

*We’ve just explored what problems can be solved with finite memory.*
So, what else is out there?
Well, what if we add unbounded memory to our machines?
This stack tells us we’ve seen two left parentheses and need to find two matching right parentheses.
These machines can do more than our old machines!
Can we describe these languages another way?
\[
\begin{align*}
S & \rightarrow 1S1 \\
S & \rightarrow 1S \\
S & \rightarrow \geq
\end{align*}
\]
\[ S \rightarrow 1S1 \]
\[ S \rightarrow 1S \]
\[ S \rightarrow \varepsilon \]

\[ \varepsilon, S \rightarrow 1S1 \]
\[ \varepsilon, S \rightarrow 1S \]
\[ \varepsilon, S \rightarrow \geq \]
\[ \Sigma, \Sigma \rightarrow \varepsilon \]
Awesome! We can call the languages these models generate or recognize the *context-free languages*. 
So, did we get every language yet?
$uv^2xy^2z \in L$
There are languages that don’t satisfy the Pumping Lemma for CFLs, meaning it’s impossible to design a CFG to describe them.
I guess not.
I guess not.

Darn right.
So, what if we make our memory a little better?
\[0 \rightarrow 0, R\]
\[0 \rightarrow 0, R\]

\[\square \rightarrow \square, R\]
\[\square \rightarrow \square, L\]

\[0 \rightarrow 0, L\]
\[1 \rightarrow 1, L\]

\[0 \rightarrow 0, R\]
\[1 \rightarrow 0, R\]

\[1 \rightarrow \square, R\]
\[\square \rightarrow \square, R\]

\[q_{\text{acc}}\]
\[q_{\text{rej}}\]
Adding an infinite tape to a finite automaton, we get a *Turing machine*. 
Can we make these any more powerful?
The *Church–Turing thesis* says that we can’t!

Why is that?
Turing machines can simulate other Turing machines!

In fact, any reasonable model of computation could be simulated by a Turing machine.
So, is every language decidable?
Consider what happens when a program is run on its own source code.

(Or – equivalently – when a Turing machine is run on its own encoding.)
def will_accept(program, input):
    ...some implementation...

def main(my_input):
    self = my_source()

    if will_accept(self, my_input):
        return False
    else:
        return True

What happens if...
...this program accepts its input?
    Then it rejects its input!
...this program doesn’t accept its input?
    Then it accepts its input!
The power of self-reference immediately limits what Turing machines can do!
It's not just $A_{TM}$ and the Halting Problem; there are an *infinite* number of undecidable languages.
We can prove a particular language is undecidable by reducing a known undecidable problem to it.

E.g., if we could solve the problem of deciding whether a TM accepts the string Vassar, we could use this decider to solve $A_{TM}$ by constructing a special TM that accepts the string Vassar in exactly the cases where TM $M$ run on string $w$ would accept.
Rice’s Theorem tells us that any language about a non-trivial property of a Turing-recognizable language will be undecidable for exactly this reason.
There are an infinite number of undecidable languages, but is every language at least \textit{recognizable}?
<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>$\cdots$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$M_0$ Acc</td>
<td>No</td>
<td>No</td>
<td>Acc</td>
<td>Acc</td>
<td>No</td>
<td>$\cdots$</td>
</tr>
<tr>
<td>$M_1$ Acc</td>
<td>Acc</td>
<td>Acc</td>
<td>Acc</td>
<td>Acc</td>
<td>Acc</td>
<td>$\cdots$</td>
</tr>
<tr>
<td>$M_2$ Acc</td>
<td>Acc</td>
<td>Acc</td>
<td>Acc</td>
<td>Acc</td>
<td>Acc</td>
<td>$\cdots$</td>
</tr>
<tr>
<td>$M_3$ No</td>
<td>Acc</td>
<td>Acc</td>
<td>No</td>
<td>Acc</td>
<td>Acc</td>
<td>$\cdots$</td>
</tr>
<tr>
<td>$M_4$ Acc</td>
<td>No</td>
<td>Acc</td>
<td>No</td>
<td>Acc</td>
<td>No</td>
<td>$\cdots$</td>
</tr>
<tr>
<td>$M_5$ No</td>
<td>No</td>
<td>Acc</td>
<td>Acc</td>
<td>No</td>
<td>No</td>
<td>$\cdots$</td>
</tr>
<tr>
<td>$\cdots$</td>
<td>$\cdots$</td>
<td>$\cdots$</td>
<td>$\cdots$</td>
<td>$\cdots$</td>
<td>$\cdots$</td>
<td>$\cdots$</td>
</tr>
<tr>
<td>No</td>
<td>No</td>
<td>No</td>
<td>Acc</td>
<td>No</td>
<td>Acc</td>
<td>$\cdots$</td>
</tr>
</tbody>
</table>
Oh great. Some problems are *impossible*. 
Intuitively, for any non-RE language, there will be some string that is in the language but cannot be proven to be in the language.
We can make the general claim:

*There are statements that are true but not provable.*

Which corresponds, roughly, to Gödel’s incompleteness theorem.
In fact, almost all languages – an uncountably infinite number of them – aren’t in RE; they’re unsolvable problems.
“Space,” it says, “is big. Really big. You just won’t believe how vastly hugely mindbogglingly big it is.”

Douglas Adams, *The Hitchhiker’s Guide to the Galaxy*
“Space,” it says, “is big. Really big. You just won’t believe how vastly hugely mindbogglingly big it is.”

Douglas Adams, *The Hitchhiker’s Guide to the Galaxy*
We’ve gone to the absolute limits of computing.
Discovery isn’t a straight road

The Transfăgărășan, a road in the Carpathian Mountains of Romania
Discovery isn’t a straight road

The ideas and results we’ve seen weren’t discovered in this order.

The class of regular languages was introduced by Kleene in 1951, 15 years after Turing machines!

DFAs were invented by Rabin & Scott 8 years after regular expressions.

And they weren’t always intended for these purposes.

Context-free grammars were invented by Chomsky in 1957 for modeling the syntax of natural languages.

The state-elimination method was introduced for circuit design!
Where to go from here?
Congratulations on making it this far!
<table>
<thead>
<tr>
<th>Formal languages</th>
<th>Non-regular languages</th>
<th>TM encodings</th>
</tr>
</thead>
<tbody>
<tr>
<td>Regular languages</td>
<td>Context-free grammars</td>
<td>Universal Turing machines</td>
</tr>
<tr>
<td>DFAs</td>
<td>Chomsky normal form</td>
<td>Self-reference</td>
</tr>
<tr>
<td>NFAs</td>
<td>Derivability</td>
<td>Decidability</td>
</tr>
<tr>
<td>5-tuples</td>
<td>Parse trees</td>
<td>Recognizability</td>
</tr>
<tr>
<td>Transition functions</td>
<td>CYK parsing</td>
<td>Self-defeating objects</td>
</tr>
<tr>
<td>Closure properties</td>
<td>Ambiguity</td>
<td>Undecidable problems</td>
</tr>
<tr>
<td>Subset construction</td>
<td>Pushdown automata</td>
<td>Halting Problem</td>
</tr>
<tr>
<td>Kleene closure</td>
<td>Pumping Lemma for CFLs</td>
<td>Diagonalization language</td>
</tr>
<tr>
<td>Regular expressions</td>
<td>Turing machines</td>
<td>Proof by reduction</td>
</tr>
<tr>
<td>State elimination</td>
<td>Subroutines</td>
<td>Rice’s Theorem</td>
</tr>
<tr>
<td>Pumping Lemma for Regular Languages</td>
<td>Church–Turing thesis</td>
<td>…</td>
</tr>
</tbody>
</table>
You’ve done more than tick off a bunch of boxes.

You’ve given yourself the foundation to tackle problems from all over computer science.
CMPU 241: Analysis of Algorithms

A mix of theory and practice, where you’ll learn about computational complexity: Which decidable problems can we compute efficiently and which are so inefficient they might as well be undecidable? Expect more reductions!
The ideas we've presented on defining languages, writing grammars, parsing strings, and writing finite automata form the basis for turning computer programs from strings of symbols into action. All computer programming rests on what we’ve learned.
If there’s a hero of this course, it’s Alan Turing. His work in theoretical computer science was motivated by the question of how we might create a thinking machine. What would this mean? How can we go from finite automata to artificial intelligence? Fewer proofs, but plenty of big ideas and big problems.
This is my area of research, where many of the ideas we use are applied to human languages. While programming languages are unambiguous and we know when we’ve understood them correctly, human languages are fascinating collections of ambiguity! Formal grammars, Chomsky normal form, and parse trees are important here.
Final thoughts
“Present-day computers are built of transistors and wires, but they could just as well be built, according to the same principles, from valves and water pipes, or from sticks and string. The principles are the essence of what makes a computer compute. One of the most remarkable things about computers is that their essential nature transcends technology.”

W. Daniel Hillis, *The Pattern on the Stone*
CS theory is all about asking what's possible in computer science.
There’s so much more to explore and so many big questions to ask – *many of which haven’t been asked yet!*
A whole world of theory and practice awaits.
That’s it!
Next time, we’ll review for the Exam 3 by working through practice problems and answering your questions.
Acknowledgments

This lecture incorporates material from:

Keith Schwarz, Stanford University