The Big Picture

30 April 2020
End of semester updates
End of semester updates

DON’T PANIC
Assignment 9 corrections are due today.

For any assignments (or Exam 2) that you haven’t submitted, the final deadline is the end of study week.

On Tuesday we’ll have an in-class review for Exam 3.

Thank you for your patience as I’ve fallen behind on grading this semester. I’ll do my best to have everything back to you in enough time to study.
Wrapping up

At this point in the course material, we’d normally dive into doing proof by reduction and Rice’s theorem.

I considered moving the Exam 3 review to be after the end of classes and trying to get through it all, but it’s a lot to lay on you at the end of this semester.

Instead, we’ll go through a very brief overview of what these topics are so you know what we’re missing, and then we’ll take a step back to look at the big picture of the semester.
Reductions
Suppose you want to build a recognizer for

\[ L = \{ \langle M \rangle \mid M \text{ is a TM that accepts } \varepsilon \} \]

Do you need to start from scratch?

What Turing machine could you use to recognize this language?
Suppose you want to build a recognizer for

$L = \{ \langle M \rangle \mid M \text{ is a TM that accepts } \varepsilon \}$

Do you need to start from scratch?

What Turing machine could you use to recognize this language?

*The universal TM $U$ recognizes $A_{TM}$, i.e., for any TM $M$ and string $w$, $\langle M, w \rangle \in A_{TM}$ iff $M$ accepts $w$.***
Suppose you want to build a recognizer for

\[ L = \{ \langle M \rangle \mid M \text{ is a TM that accepts } \varepsilon \} \]

Do you need to start from scratch?

What Turing machine could you use to recognize this language?

The universal TM \( U \) recognizes \( A_{TM} \), i.e., for any TM \( M \) and string \( w \), \( \langle M, w \rangle \in A_{TM} \) iff \( M \) accepts \( w \).
Suppose you want to build a recognizer for

\[ L = \{ \langle M \rangle \mid M \text{ is a TM that accepts } \varepsilon \} \]

Do you need to start from scratch?

What Turing machine could you use to recognize this language?

\[ \langle M \rangle \in L \iff \langle M, \varepsilon \rangle \in A_{TM} \]

The universal TM U recognizes \( A_{TM} \), i.e., for any TM M and string w, \( \langle M, w \rangle \in A_{TM} \) iff M accepts w.
Suppose you want to build a recognizer for

\[ L = \{ \langle M \rangle \mid M \text{ is a TM that accepts } \varepsilon \} \]

Do you need to start from scratch?

What Turing machine could you use to recognize this language?

The universal TM \( U \) recognizes \( A_{TM} \), i.e., for any TM \( M \) and string \( w \), \( \langle M, w \rangle \in A_{TM} \) iff \( M \) accepts \( w \).

\[ \langle M \rangle \in L \text{ iff } \langle M, \varepsilon \rangle \in A_{TM} \]

Make a TM that accept \( \langle M \rangle \) iff \( U \) accepts \( \langle M, \varepsilon \rangle \).
\[ L = \{ \langle M \rangle \mid M \text{ is a TM that accepts } \varepsilon \} \]
$L = \{ \langle M \rangle \mid M \text{ is a TM that accepts } \varepsilon \}$

\[ H = \text{“On input } \langle M \rangle: \]
\begin{enumerate}
  \item Construct the string $\langle M, \varepsilon \rangle$.
  \item Run $U$ on $\langle M, \varepsilon \rangle$.
  \item If $U$ accepts $\langle M, \varepsilon \rangle$, then $H$ accepts $\langle M \rangle$.
  \item If $U$ rejects $\langle M, \varepsilon \rangle$, then $H$ rejects $\langle M \rangle$.”
\end{enumerate}
We can use the same approach for lots of problems:

We give a special input to an existing TM and decide whether to accept or reject based on what it does.
From $\text{HALT}_{\text{TM}}$ to $A_{\text{TM}}$

$\langle M \rangle$ Change $M$ to loop instead of reject $\langle M' \rangle$

$D$ TM for $\text{HALT}_{\text{TM}}$

$w$ $w$
From $\text{HALT}_\text{TM}$ to $\text{A}_\text{TM}$

$H = \text{“On input } \langle M, w \rangle: \text{“}$

1. Build $M$ into $M'$ so $M'$ loops when $M$ rejects.
2. Run $D$ on $\langle M', w \rangle$.
3. If $D$ accepts $\langle M', w \rangle$, then $H$ accepts $\langle M, w \rangle$.
4. If $D$ rejects $\langle M', w \rangle$, then $H$ rejects $\langle M, w \rangle$.\text{“}
Reductions

Intuitively, problem $A$ reduces to problem $B$ iff a solver for $B$ can be used to solve $A$.

Reductions can be used to show certain problems are “solvable”:

If $A$ reduces to $B$ and $B$ is “solvable”, then $A$ is “solvable”.
Mapping reductions

A function \( f : \Sigma_1^* \rightarrow \Sigma_2^* \) is called a \textit{mapping reduction} from A to B iff

For any \( w \in \Sigma_1^* \), \( w \in A \) iff \( f(w) \in B \), and

\( f \) is a computable function.

Intuitively, a mapping reduction from A to B says that a computer can transform any instance of A into an instance of B such that the answer to B is the answer to A.
Mapping reducibility

If there is a mapping reduction from language $A$ to language $B$, we say that language $A$ is *mapping reducible* to language $B$.

Notation: $A \leq_m B$ iff language $A$ is mapping reducible to language $B$.

Note that we reduce *languages*, not *machines*. 
$A \leq_m B$

$H =$ “On input $w$:

1. Compute $f(w)$.
2. Run machine $R$ on $f(w)$.
3. If $R$ accepts $f(w)$, then $H$ accepts $w$.
4. If $R$ rejects $f(w)$, then $H$ rejects $w$.”
$A \leq_m B$

$H = "On input w:
1. Compute $f(w)$.
2. Run machine $R$ on $f(w)$.
3. If $R$ accepts $f(w)$, then $H$ accepts $w$.
4. If $R$ rejects $f(w)$, then $H$ rejects $w."$

If $R$ is a decider for $B$, then $H$ is a decider for $A$. 

$\text{TM for language } B$
$A \leq_m B$

$H = \text{"On input } w:\n1. \text{ Compute } f(w).$
2. \text{ Run machine } R \text{ on } f(w).$
3. \text{ If } R \text{ accepts } f(w), \text{ then } H \text{ accepts } w.$
4. \text{ If } R \text{ rejects } f(w), \text{ then } H \text{ rejects } w.$
$A \leq_m B$

$H = \text{"On input } w:\n1. \text{ Compute } f(w).$
2. \text{ Run machine } R \text{ on } f(w).$
3. \text{ If } R \text{ accepts } f(w), \text{ then } H \text{ accepts } w.$
4. \text{ If } R \text{ rejects } f(w), \text{ then } H \text{ rejects } w.$$

If $R$ is a decider for $B$, then $H$ is a decider for $A$.
If $R$ is a recognizer for $B$, then $H$ is a recognizer for $A$.
If $R$ is a co-recognizer for $B$, then $H$ is a co-recognizer for $A$. 
Why mapping reducibility matters

THEOREM. If $B \in \mathbf{R}$ and $A \leq_{m} B$, then $A \in \mathbf{R}$.

THEOREM. If $B \in \mathbf{RE}$ and $A \leq_{m} B$, then $A \in \mathbf{RE}$.

THEOREM. If $B \in \text{co-RE}$ and $A \leq_{m} B$, then $A \in \text{co-RE}$.

*Intuitively*: $A \leq_{m} B$ means “$A$ is not harder than $B$.”
Why mapping reducibility matters

THEOREM. If \( B \not\in R \) and \( A \leq_m B \), then \( A \not\in R \).

THEOREM. If \( B \not\in RE \) and \( A \leq_m B \), then \( A \not\in RE \).

THEOREM. If \( B \not\in co\-RE \) and \( A \leq_m B \), then \( A \not\in co\-RE \).

*Intuitively:* \( A \leq_m B \) means “\( B \) is at least as hard as \( A \)”
If this one is “easy” \((R, RE, co-RE)\) …

\[ A \leq_m B \]

… then this one is “easy” \((R, RE, co-RE)\) too.
If this one is “hard” (not $R$, not $RE$, not co-$RE$) …

$A \leq_m B$

… then this one is “hard” (not $R$, not $RE$, not co-$RE$) too.
Proof by reduction

Suppose that we are given a language $L$ that we believe is undecidable.

We can prove that this is true using the following technique:

Assume, for the sake of contradiction, that $L$ is decidable.

Show how a decider for $L$ could be used to construct a decider for an undecidable language.

Conclude that $L$ must not be decidable.
Rice’s Theorem
A *property of an RE language* is some trait that may apply to RE languages.

For example:

- Does $L = \emptyset$?
- Is $L$ regular?
- Is $L$ context-free?
- Does $L$ contain any string of length exactly 137?
A property of **RE** languages is called *trivial* if all **RE** languages have the property or no **RE** languages have the property, e.g.,

\[
\{\langle M \rangle \mid L(M) \text{ is RE}\} \text{ is trivial}
\]

\[
\{\langle M \rangle \mid L(M) \text{ is not RE}\} \text{ is trivial}
\]

A property of **RE** languages is called *nontrivial* if there exist TMs \( M_1 \) and \( M_2 \) such that \( \langle M_1 \rangle \) has the property, but \( \langle M_2 \rangle \) doesn’t, e.g.,

\[
\{\langle M \rangle \mid L(M) \text{ is infinite}\} \text{ is nontrivial}
\]

\[
\{\langle M \rangle \mid L(M) \text{ is regular}\} \text{ is nontrivial}
\]

\[
\{\langle M \rangle \mid L(M) \text{ is decidable}\} \text{ is nontrivial}
\]
Rice’s Theorem

Any nontrivial property of the RE languages is \textit{undecidable}. 
Rice’s Theorem tells us that all of the following problems are undecidable:

\[ L_{\text{palindrome}} = \{ \langle M \rangle \mid \text{every string in } L(M) \text{ is a palindrome} \} \]
\[ L_{\text{allodd}} = \{ \langle M \rangle \mid \text{every string in } L(M) \text{ has odd length} \} \]
\[ L_{\text{CFL}} = \{ \langle M \rangle \mid L(M) \text{ is a context-free language} \} \]
\[ L_{\text{short}} = \{ \langle M \rangle \mid L(M) \text{ has no strings of length greater than 5} \} \]
\[ L_{\text{decidable}} = \{ \langle M \rangle \mid L(M) \text{ is decidable} \} \]
\[ E_{\text{TM}} = \{ \langle M \rangle \mid L(M) = \emptyset \} \]
THUS, FOR ANY NONDETERMINISTIC TURING MACHINE M THAT RUNS IN SOME POLYNOMIAL TIME $\rho(n)$, WE CAN DEVISE AN ALGORITHM THAT TAKES AN INPUT $w$ OF LENGTH $n$ AND PRODUCES $E_{M,w}$. THE RUNNING TIME IS $O(\rho^2(n))$ ON A MULTITAPE DETERMINISTIC TURING MACHINE AND...

WTF, MAN. I JUST WANTED TO LEARN HOW TO PROGRAM VIDEO GAMES.

Source unknown; let me know!
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.
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?
\( \varepsilon R_1^* R_2 \)
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?
\[ L = \{ a^i b^i \mid i \in \mathbb{N}_0 \} \]
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?
\[
S \rightarrow 1S1 \\
S \rightarrow 1S \\
S \rightarrow \geq \\
\geq \\
1 \geq 1 \\
11 \geq 1 \\
11 \geq 11 \\
111 \geq 1 \\
111 \geq 11 \\
111 \geq 111 \\
\ldots
\]
S → 1S1
S → 1S
S → ≥

ε, S → 1S1
ε, S → 1S
ε, S → ≥
Σ, Σ → ε

ε, ε → S$
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?
Check for 0

q_{rej}

q_{acc}

0 \rightarrow 0, L
1 \rightarrow 1, L

\square \rightarrow \square, R

\begin{align*}
0 & \rightarrow 0, R \\
1 & \rightarrow 1, R \\
\square & \rightarrow \square, R
\end{align*}

\begin{align*}
0 & \rightarrow \square, R \\
1 & \rightarrow \square, R
\end{align*}

go to start

clear a 1

go to end

\begin{align*}
0 & \rightarrow 0, R \\
1 & \rightarrow 1, R
\end{align*}

\begin{align*}
\square & \rightarrow \square, L \\
\square & \rightarrow \square, R
\end{align*}
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):
    my_source = open("uh-oh.py").read()

    if will_accept(my_source, 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!

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.
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 \textit{Vassar}, we could use this decider to solve \( A_{TM} \) by constructing a special TM that accepts the string \textit{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 recognizable?
<table>
<thead>
<tr>
<th></th>
<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>
<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>
<td>...</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>
<td>...</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>
<td>...</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>
<td>...</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>
<td>...</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td></td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>Acc</td>
<td>No</td>
<td>Acc</td>
<td>...</td>
</tr>
</tbody>
</table>
Oh great. Some problems are impossible.
So, we can’t recognize everything, but can we at least refute membership, i.e., recognize all the things that aren’t members of the language?
\[ \text{REGULAR}_{\text{TM}} = \{ \langle M \rangle \mid M \text{ is a TM and } L(M) \text{ is regular} \} \]

\[ \text{REGULAR}_{\text{TM}} \notin \text{RE} \]

\[ \text{REGULAR}_{\text{TM}} \notin \text{co-RE} \]
Its complement is recognizable by a TM

Decidable by a PDA
Generated by a CFG

Decided by a DFA, NFA, or RE

Recognizable by a TM

All languages
All languages

REG
Decided by a DFA, NFA, or RE

CFL
Decided by a PDA
Generated by a CFG

RE
Recognizable by a TM

co-RE
Its complement is recognizable by a TM

R
Decidable by a TM
In fact, almost all languages – an uncountably infinite number of them – are not in \textbf{RE} or \textbf{co-RE}.
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!
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: 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.
COGS 101: Introduction to Cognitive Science
and
CMPU 365: Artificial Intelligence

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.
CMPU 336: Computational Linguistics

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! Ideas of 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 final exam by working through practice problems and answering your questions.
Acknowledgments

This lecture incorporates material from:

Keith Schwarz