Week 7 at a glance

Textbook reading: Chapter 4
No class on Monday in observance of UCSD holiday.

Before Wednesday, Introduction to Chapter 4.

Before Friday, Decidable problems concerning regular languages, Sipser pages 194-196.

For Week 8 Monday: An undecidable language, Sipser pages 207-209.

We will be learning and practicing to:

- Clearly and unambiguously communicate computational ideas using appropriate formalism. Translate across levels of abstraction.
  - Use clear English to describe computations of Turing machines informally.
    - **Use high-level descriptions to define and trace Turing machines**
    - **Apply dovetailing in high-level definitions of machines**
  - Give examples of sets that are regular, context-free, decidable, or recognizable (and prove that they are).
    - **Give examples of sets that are decidable.**
    - **Give examples of sets that are recognizable.**
- Know, select and apply appropriate computing knowledge and problem-solving techniques. Reason about computation and systems.
  - Translate a decision problem to a set of strings coding the problem.
    - **Connect languages and computational problems**
    - **Describe and use the encoding of objects as inputs to Turing machines**
    - **Trace high-level descriptions of algorithms for computational problems**
  - Classify the computational complexity of a set of strings by determining whether it is regular, context-free, decidable, or recognizable.
    - **Describe common computational problems with respect to DFA, NFA, regular expressions, PDA, and context-free grammars.**
    - **Give high-level descriptions of Turing machines that decide common computational problems with respect to DFA, NFA, regular expressions, PDA, and context-free grammars.**

TODO:

Review Quiz 6 on PrairieLearn (http://us.prairielearn.com), due 2/19/2025

Homework 4 submitted via Gradescope (https://www.gradescope.com/), due 2/20/2025

Review Quiz 7 on PrairieLearn (http://us.prairielearn.com), due 2/26/2025

Monday: No class, in observance of UCSD holiday

Wednesday: General constructions for Turing machines

Definition: A language L over an alphabet Σ is called **co-recognizable** if its complement, defined as $\Sigma^* \setminus L = \{x \in \Sigma^* \mid x \notin L\}$, is Turing-recognizable.

Notation: The complement of a set X is denoted with a superscript c, X^c, or an overline, $\overline{X}$.

Theorem (Sipser Theorem 4.22): A language is Turing-decidable if and only if both $\boxed{\text{it}}$ and its $\boxed{\text{complement}}$ are Turing-recognizable.

Yes! Yes and No!

Proof, first direction: Suppose language L is Turing-decidable. WTS that both it and its complement are Turing-recognizable.

By definition, we have a TM M that decides L; namely for each string w if $w \in L$, M accepts w and if $w \notin L$, M rejects w.

Goal ① Build TM that recognizes L

Use M as it is!

Goal ② Build TM that recognizes $\overline{L}$.

Build M_{new} = " On input w

1. Run M on w
2. If M accepts w, reject.
3. If M rejects w, accept"

(guaranteed to halt with finitely many steps by assumption on M)

Claim $L(M_{new}) = \overline{L}$ ...

Proof, second direction: Suppose language L is Turing-recognizable, and so is its complement. WTS that L is Turing-decidable.

By definition, we have a TM M_L with $L(M_L) = L$ and another TM M_c with $L(M_c) = \overline{L}$.

Goal: Build TM that recognizes L and is a decider.

deterministic

~~Build M_{new} = " On input w~~

~~1. Run M_L on w~~

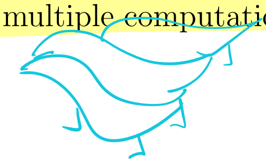
~~2. Run M_c on w~~

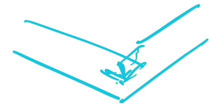
~~3. If M_L halts and accepts, accept~~

~~4. If M_c halts and accepts, reject."~~

UH OH!

Dovetailing: interleaving progress on multiple computations by limiting the number of steps each computation makes in each round.





Build M_{new} = " On input w

1. For $n = 1, 2, 3, \ldots$
2. Run M_L on w for (at most) n steps
3. Run M_C on w for (at most) n steps
4. If M_L accepts w within n steps, accept.
5. If M_C accepts w within n steps, reject.
6. Increment n and continue loop"

Claim $L(M_{new}) = L$

Claim M_{new} is a decider

It's sufficient to prove that each string in L is accepted by M_{new} and each string not in L is rejected by M_{new}.

First, let w be an arbitrary string in L. By assumption that M_L recognizes L, we know that M_L accepts w. Let ℓ be the number of steps it takes M_L to halt and accept w. By assumption that M_C recognizes $\bar{L}$, we know that M_C does not accept w. We trace the computation of M_{new} on w: For all iterations of the loop with $n < \ell$, step 2 and 3 run for at most n steps and the conditions in step 4 and 5 are not satisfied. At the loop iteration with $n = \ell$, the subroutine in step 2 ends with M_L accepting w. After the (at most) ℓ steps of the computation simulated in step 3, in step 4, the condition of the conditional is true, so M_{new} accepts w ✓

Next, let w be an arbitrary string not in L. By assumption that M_C recognizes $\bar{L}$, we know that M_C accepts w. Let ℓ' be the number of steps it takes M_C to halt and accept w. By assumption that M_L recognizes L, we know that M_L does not accept w. Tracing the computation of M_{new} on w (like before) by definition of ℓ', the computation doesn't halt for loop iterations with $n < \ell'$; and at $n = \ell'$ the subroutine in step 2 doesn't halt and accept but in step 3 it does so the condition in step 4 isn't satisfied and the computation continues to step 5, where the condition is satisfied and M_{new} rejects w. ✓

QED

Claim: If two languages (over a fixed alphabet Σ) are Turing-decidable, then their union is as well.

Proof: Let L_1, L_2 be arbitrary decidable language. Let M_1, M_2 be deciders with $L_1 = L(M_1)$, $L_2 = L(M_2)$ guaranteed to exist by definition of L_1, L_2 being decidable. Goal: build decider for $L_1 \cup L_2$.

Define M = " On input w

1. Run M_1 on w [Halts within finitely many steps]
2. If M_1 accepts w, accept.
3. Otherwise, run M_2 on w [Halts within finitely many steps].
4. If M_2 accepts w, accept
5. Otherwise, reject. "

Claim: M decides $L_1 \cup L_2$.

Pf: Let w be arbitrary string WTS M accepts w.
First, assume $w \in L_1 \cup L_2$ and WTS M accepts w.

⋮

Next, assume $w \notin L_1 \cup L_2$ and WTS M rejects w

⋮

Claim: If two languages (over a fixed alphabet Σ) are Turing-recognizable, then their union is as well.

Proof: Let L_1, L_2 be arbitrary recognizable language. Let M_1, M_2 be TMs with $L_1 = L(M_1)$, $L_2 = L(M_2)$ guaranteed to exist by definition of L_1, L_2 being recognizable. Goal: build TM for $L_1 \cup L_2$.

Define M = " On input w

1. For n = 1, 2, ...
2. Run M_1 on w for (at most) n steps
3. If M_1 accepts w, accept.
4. Otherwise, run M_2 on w for (at most) n steps.
5. If M_2 accepts w, accept
6. Otherwise, continue to next loop iteration."

Claim: M recognizes $L_1 \cup L_2$.

Pf: Let w be arbitrary string WTS M accepts w.

First, assume $w \in L_1 \cup L_2$ and WTS M accepts w.

⋮

Next, assume $w \notin L_1 \cup L_2$ and WTS M rejects w

⋮

What about intersection? (see homework)

Friday: Decidable problems about regular languages

The Church-Turing thesis posits that each algorithm can be implemented by some Turing machine.

Describing algorithms (Sipser p. 185) To define a Turing machine, we could give a

- **Formal definition**: the 7-tuple of parameters including set of states, input alphabet, tape alphabet, transition function, start state, accept state, and reject state. This is the low-level programming view that models the logic computation flow in a processor.
- **Implementation-level definition**: English prose that describes the Turing machine head movements relative to contents of tape, and conditions for accepting / rejecting based on those contents. This level describes memory management and implementing data access with data structures.
  - Mention the tape or its contents (e.g. "Scan the tape from left to right until a blank is seen.")
  - Mention the tape head (e.g. "Return the tape head to the left end of the tape.")
- **High-level description** of algorithm executed by Turing machine: description of algorithm (precise sequence of instructions), without implementation details of machine. High-level descriptions of Turing machine algorithms are written as indented text within quotation marks. Stages of the algorithm are typically numbered consecutively. The first line specifies the input to the machine, which must be a string.
  - Use other Turing machines as subroutines (e.g. "Run M on w")
  - Build new machines from existing machines using previously shown results (e.g. "Given NFA A construct an NFA B such that $L(B) = \overline{L(A)}$")
  - Use previously shown conversions and constructions (e.g. "Convert regular expression R to an NFA N")

$A \rightsquigarrow \tilde{A}$ (DFA) $\rightsquigarrow \tilde{B}$ (DFA for cmp)

B (NFA for cmp)

Formatted inputs to Turing machine algorithms

Type checking

The input to a Turing machine is always a string. The format of the input to a Turing machine can be checked to interpret this string as representing structured data (like a csv file, the formal definition of a DFA, another Turing machine, etc.)

This string may be the encoding of some object or list of objects.

Notation: $\langle O \rangle$ is the string that encodes the object O. $\langle O_1, \ldots, O_n \rangle$ is the string that encodes the list of objects $O_1, \ldots, O_n$.

Assumption: There are algorithms (Turing machines) that can be called as subroutines to decode the string representations of common objects and interact with these objects as intended (data structures). These algorithms are able to "type-check" and string representations for different data structures are unique.

"On input n, integer

1.

2.

"On input G, a graph

1 For each node in G,

2

For example, since there are algorithms to answer each of the following questions, by Church-Turing thesis, there is a Turing machine that accepts exactly those strings for which the answer to the question is "yes"

- Does a string over $\{0, 1\}$ have even length?
- Does a string over $\{0, 1\}$ encode a string of ASCII characters?[1]
- Does a DFA have a specific number of states?

$(\boxed{Q}, \Sigma, \delta, q_0, F)$

- Do two NFAs have any state names in common?
- Do two CFGs have the same start variable?

$(V_1, \Sigma_1, R_1, S_1)$ $(V_2, \Sigma_2, R_2, S_2)$

A **computational problem** is decidable iff language encoding its positive problem instances is decidable.

The computational problem "Does a specific DFA accept a given string?" is encoded by the language

$$\begin{aligned}&\{\text{representations of DFAs } M \text{ and strings } w \text{ such that } w \in L(M)\}\\ =&\{\langle M, w\rangle \mid M \text{ is a DFA}, w \text{ is a string}, w \in L(M)\}\end{aligned}$$

The computational problem "Is the language generated by a CFG empty?" is encoded by the language

$$\begin{aligned}&\{\text{representations of CFGs } G \text{ such that } L(G) = \emptyset\}\\ =&\{\langle G\rangle \mid G \text{ is a CFG}, L(G) = \emptyset\}\end{aligned}$$

The computational problem "Is the given Turing machine a decider?" is encoded by the language

$$\begin{aligned}&\{\text{representations of TMs } M \text{ such that } M \text{ halts on every input}\}\\ =&\{\langle M\rangle \mid M \text{ is a TM and for each string } w, M \text{ halts on } w\}\end{aligned}$$

$(Q, \Sigma, \Gamma, \delta, q_0, q_{acc}, q_{rej})$

Note: writing down the language encoding a computational problem is only the first step in determining if it's recognizable, decidable, or ...

Deciding a computational problem means building / defining a Turing machine that recognizes the language encoding the computational problem, and that is a decider.

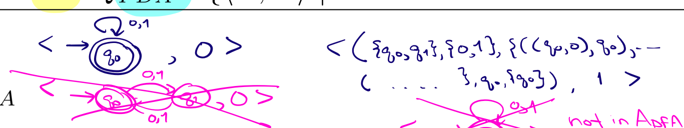
Some classes of computational problems will help us understand the differences between the machine models we've been studying. (Sipser Section 4.1)

[1] An introduction to ASCII is available on the w3 tutorial here.

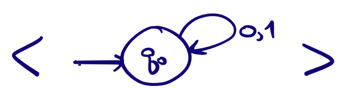
Σ type representation ⟨ ⟩ specific string

Acceptance problem		
...for DFA	A_{DFA}	$\{\langle B, w\rangle \mid B$ is a DFA that accepts input string $w\}$
...for NFA	A_{NFA}	$\{\langle B, w\rangle \mid B$ is a NFA that accepts input string $w\}$
...for regular expressions	A_{REX}	$\{\langle R, w\rangle \mid R$ is a regular expression that generates input string $w\}$
...for CFG	A_{CFG}	$\{\langle G, w\rangle \mid G$ is a context-free grammar that generates input string $w\}$
...for PDA	A_{PDA}	$\{\langle B, w\rangle \mid B$ is a PDA that accepts input string $w\}$
Language emptiness testing		
...for DFA	E_{DFA}	$\{\langle A\rangle \mid A$ is a DFA and $L(A) = \emptyset\}$
...for NFA	E_{NFA}	$\{\langle A\rangle \mid A$ is a NFA and $L(A) = \emptyset\}$
...for regular expressions	E_{REX}	$\{\langle R\rangle \mid R$ is a regular expression and $L(R) = \emptyset\}$
...for CFG	E_{CFG}	$\{\langle G\rangle \mid G$ is a context-free grammar and $L(G) = \emptyset\}$
...for PDA	E_{PDA}	$\{\langle A\rangle \mid A$ is a PDA and $L(A) = \emptyset\}$
Language equality testing		
...for DFA	EQ_{DFA}	$\{\langle A, B\rangle \mid A$ and B are DFAs and $L(A) = L(B)\}$
...for NFA	EQ_{NFA}	$\{\langle A, B\rangle \mid A$ and B are NFAs and $L(A) = L(B)\}$
...for regular expressions	EQ_{REX}	$\{\langle R, R'\rangle \mid R$ and R' are regular expressions and $L(R) = L(R')\}$
...for CFG	EQ_{CFG}	$\{\langle G, G'\rangle \mid G$ and G' are CFGs and $L(G) = L(G')\}$
...for PDA	EQ_{PDA}	$\{\langle A, B\rangle \mid A$ and B are PDAs and $L(A) = L(B)\}$

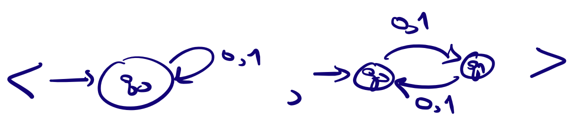
Example strings in A_{DFA}



Example strings in E_{DFA}



Example strings in EQ_{DFA}



$M_1 =$ "On input $\boxed{\langle M, w\rangle}$, where M is a DFA and w is a string:

0. Type check encoding to check input is correct type. If not, reject.
1. Simulate M on input w (by keeping track of states in M, transition function of M, etc.)
2. If the simulation ends in an accept state of M, accept. If it ends in a non-accept state of M, reject. "

finitely many steps

finitely many steps

guaranteed to take finitely many steps, corresponding to $|w|$. *

conditional: take finitely many string

What is $L(M_1)$? A_{DFA}

Is M_1 a decider? Yes!

Alternate description: Sometimes omit step 0 from listing and do implicit type check.

Synonyms: "Simulate", "run", "call".

~~True~~ / False: $A_{REX} = A_{NFA} = A_{DFA}$

if a string is formatted to represent a regular expression it can't represent a NFA or DFA

True / ~~False~~: $A_{REX} \cap A_{NFA} = \emptyset$, $A_{REX} \cap A_{DFA} = \emptyset$, $A_{DFA} \cap A_{NFA} = \emptyset$

But A_{REX} and A_{NFA} are both decidable too!

To prove this, we can convert input to DFA and then use M_1.

For example, to decide A_{NFA}:

"On input $\langle M, w \rangle$ where M is NFA and w is a string

1. Use macro state construction (Theorem 1.39) to construct a DFA M_D with $L(M_D) = L(M)$
2. Run M_1 on $\langle M_D, w \rangle$
3. If it accepts, accept; if it rejects, reject"

For example, to decide A_{REX}:

"On input $\langle R, w \rangle$ where R is NFA and w is a string

1. Use recursive construction and macro states (Lemma 1.55 and Theorem 1.39) to construct a DFA M_D with $L(M_D) = L(R)$
2. Run M_1 on $\langle M_D, w \rangle$
3. If it accepts, accept; if it rejects, reject"

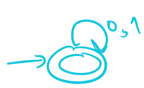
$E_{DFA} = \{\langle A \rangle \mid A \text{ is a DFA and } L(A) = \emptyset\}$. A Turing machine that decides E_{DFA} is

~~M_2 = "On input $\langle M \rangle$ where M is a DFA,~~

1. ~~For integer $i = 1, 2, \ldots$~~
2. ~~Let s_i be the ith string over the alphabet of M (ordered in string order).~~
3. ~~Run M on input s_i.~~
4. ~~If M accepts, reject. If M rejects, increment i and keep going."~~

M_3 = " On input $\langle M \rangle$ where M is a DFA,

1. Mark the start state of M.
2. Repeat until no new states get marked:
3. Loop over the states of M.
4. Mark any unmarked state that has an incoming edge from a marked state.
5. If no accept state of M is marked, ________; otherwise, ________".

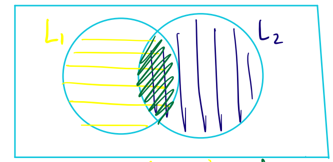
M_2 rejects <

>
(which is good)

But M_2 doesn't accept any strings at all so doesn't accept <

> even though it should.

Reachability!

$L(M) = \emptyset$ means there are no accepting states reachable from the start state of M



$L_1 = L_2$ iff [yellow-hatched region] $= \emptyset$ and [blue-hatched region] $= \emptyset$

To build a Turing machine that decides EQ_{DFA}, notice that

$$L_1 = L_2 \qquad \text{iff} \qquad \underbrace{(\ (L_1 \cap \overline{L_2}) \cup (L_2 \cap \overline{L}_1)\ )}_{} = \emptyset$$

There are no elements that are in one set and not the other

Build DFA recognizing this language and then run M_3 with string representing this DFA as input

$M_{EQDFA} =$ "On input $\langle M, \tilde{M} \rangle$ where M and $\tilde{M}$ both DFA,

1. Use Cartesian product construction and flipping status of states construction to build M_a with $L(M_a) = L(M) \cap \overline{L(\tilde{M})}$

2. Use Cartesian product construction and flipping status of states construction to build M_b with $L(M_b) = \overline{L(M)} \cap L(\tilde{M})$

3. Use cartesian product construction to build X with $L(X) = L(M_a) \cup L(M_b)$.

4. Run M_3 on $\langle X \rangle$.

5. If M_3 accepts, accept ; If M_3 rejects, reject "

Summary: We can use the decision procedures (Turing machines) of decidable problems as subroutines in other algorithms. For example, we have subroutines for deciding each of A_{DFA}, E_{DFA}, EQ_{DFA}. We can also use algorithms for known constructions as subroutines in other algorithms. For example, we have subroutines for: counting the number of states in a state diagram, counting the number of characters in an alphabet, converting DFA to a DFA recognizing the complement of the original language or a DFA recognizing the Kleene star of the original language, constructing a DFA or NFA from two DFA or NFA so that we have a machine recognizing the language of the union (or intersection, concatenation) of the languages of the original machines; converting regular expressions to equivalent DFA; converting DFA to equivalent regular expressions, etc.