Class 7
Nancy Lynch
Today

• Basic computability theory

• Topics:
  – Decidable and recognizable languages
  – Recursively enumerable languages
  – Turing Machines that solve problems involving FAs
  – Undecidability of the Turing machine acceptance problem
  – Undecidability of the Turing machine halting problem

• Reading: Sipser, Sections 3.1, 3.2, Chapter 4

• Next: Sections 5.1, 5.2
Decidable and Recognizable Languages
Decidable and recognizable languages

- Last time, we began studying the important notion of computability.
- As a concrete model of computation, we introduced basic one-tape, one-head Turing machines.
- Also discussed some variants.
- Claimed they are all equivalent, so the notion of computability is robust.
- Today: Look more carefully at the notions of computability and equivalence.
Decidable and recognizable languages

• Assume: TM has accepting state $q_{acc}$ and rejecting state $q_{rej}$.
• Definition: TM $M$ recognizes language $L$ provided that $L = \{ w \mid M \text{ on } w \text{ reaches } q_{acc} \} = \{ w \mid M \text{ accepts } w \}$.
• Another important notion of computability:

  • Definition: TM $M$ decides language $L$ provided that both of the following hold:
    – On every $w$, $M$ eventually reaches either $q_{acc}$ or $q_{rej}$.
    – $L = \{ w \mid M \text{ on } w \text{ reaches } q_{acc} \}$.

• Thus, if $M$ recognizes $L$, then:
  – Words in $L$ lead to $q_{acc}$.
  – Words not in $L$ either lead to $q_{rej}$ or never halt (“loop”).

• Whereas if $M$ decides $L$, then:
  – Words in $L$ lead to $q_{acc}$.
  – Words not in $L$ lead to $q_{rej}$. Always halts
Decidable and recognizable languages

- **Theorem 1:** If M decides L then M recognizes L.
- Obviously.
- But not necessarily vice versa.
- In fact, these two notions define **different language classes**:
- **Definition:**
  - L is **Turing-recognizable** if there is some TM that recognizes L.
  - L is **Turing-decidable** if there is some TM that decides L.
- The classes of Turing-recognizable and Turing-decidable languages are different.
- **Theorem 2:** If L is Turing-decidable then L is Turing-recognizable.
- Obviously.
- But the other direction does not hold---there are languages that are Turing-recognizable but not Turing-decidable.
- We’ll see some examples soon.
Decidable and recognizable languages

- **Theorem 3:** If L is Turing-decidable then $L^c$ is T-decidable.

- **Proof:**
  - Suppose that M decides L.
  - Design a new machine $M'$ that behaves just like M, but:
    - If M accepts, $M'$ rejects.
    - If M rejects, $M'$ accepts.
  - Formally, can do this by interchanging $q_{acc}$ and $q_{rej}$.
  - Then $M'$ decides $L^c$. 
Decidable and recognizable languages

• A basic connection between Turing-recognizable and Turing-decidable languages:
  • Theorem 4: L is Turing decidable if and only if L and \( L^c \) are both Turing-recognizable.
  • Proof: \( \Rightarrow \)
    – Suppose that L is Turing-decidable.
    – Then L is Turing-recognizable, by Theorem 2.
    – Also, \( L^c \) is Turing-decidable, by Theorem 3.
    – So \( L^c \) is Turing-recognizable, by Theorem 2.
  • Proof: \( \Leftarrow \)
    – Given \( M_1 \) recognizing L, and \( M_2 \) recognizing \( L^c \).
    – Produce a Turing Machine M that decides whether or not its input w is in L or \( L^c \).
Decidable and recognizable languages

- **Theorem 4:** L is Turing decidable if and only if L and $L^c$ are both Turing-recognizable.

- **Proof:** $\iff$

  - Given $M_1$ recognizing L, and $M_2$ recognizing $L^c$.
  - Produce a Turing Machine M that decides whether or not its input $w$ is in L or $L^c$.
  - **Idea:** Run both $M_1$ and $M_2$ on $w$.
    - One must accept.
    - If $M_1$ accepts, then M accepts.
    - If $M_2$ accepts, then M rejects.
  - But, we can't run $M_1$ and $M_2$ one after the other because the first one might never halt.
  - Run them in parallel, until one accepts?
  - **How?** We don't have a parallel Turing Machine model.
Decidable and recognizable languages

- **Theorem 4:** L is Turing decidable if and only if L and \( L^c \) are both Turing-recognizable.

- **Proof:** \( \iff \)
  - \( M_1 \) recognizes L, and \( M_2 \) recognizes \( L^c \).
  - Let M be a 2-tape Turing Machine:
Decidable and recognizable languages

• **Theorem 4:** $L$ is Turing decidable if and only if $L$ and $L^c$ are both Turing-recognizable.

• **Proof:** $\Leftarrow$
  
  – M copies input from 1\textsuperscript{st} tape to 2\textsuperscript{nd} tape.
  – Then emulates $M_1$ and $M_2$ together, step-by-step.
  – No interaction between them.
  – M’s finite-state control keeps track of states of $M_1$ and $M_2$; thus, $Q$ includes $Q_1 \times Q_2$.
  – Also includes new start, accept, and reject states and whatever else is needed for bookkeeping.
Language Classification

• Four possibilities:
  – L and \( L^c \) are both Turing-recognizable.
    • Equivalently, L is Turing-decidable.
  – L is Turing-recognizable, \( L^c \) is not.
  – \( L^c \) is Turing-recognizable, L is not.
  – Neither L nor \( L^c \) is Turing-recognizable.

• All four possibilities occur, as we will see.

• How do we know that there are languages L that are neither Turing-recognizable nor co-Turing-recognizable?

• Cardinality argument:
  – There are uncountably many languages.
  – There are only countably many Turing-recognizable languages and only countably many co-Turing-recognizable languages.
  – Because there are only countably many Turing machines (up to renaming).
Examples

• **Example:** Every regular language $L$ is decidable.
  
  – Let $M$ be a DFA with $L(M) = L$.
  – Design a Turing machine $M'$ that simulates $M$.
  – If, after processing the input, the simulated $M$ is in an accepting state, $M'$ accepts; else $M'$ rejects.
Examples

• Example: Let $X$ be the set of binary representations of natural numbers for which the following procedure halts:
  
  ```plaintext
  while $x \neq 1$ do
    if $x$ is odd then $x := 3x + 1$
    if $x$ is even then $x := x/2$
  halt
  ```

  – Obviously, $X$ is Turing-recognizable: just simulate this procedure and accept if/when it halts.

  – Is it decidable? (?)
Closure Properties

• **Theorem 5:** The set of Turing-recognizable languages is closed under set union and intersection.

  • **Proof:**
    – Run both machines in parallel.
    – For union, accept if either accepts.
    – For intersection, accept if both accept.

• However, the set of Turing-recognizable languages is not closed under complement.

• As we will soon see.

• **Theorem 6:** The set of Turing-decidable languages is closed under union, intersection, and complement.

• **Theorem 7:** Both the Turing-recognizable and Turing-decidable languages are closed under concatenation and star (HW).
Recursively Enumerable Languages
Recursively enumerable languages

• Yet another kind of computability for Turing Machines.
• An **enumerator** is a Turing Machine variant:

  - Starts with a blank work tape (no input).
  - Prints a sequence of finite strings (possibly infinitely many) on output tape.
  - More specifically, e.g.:
    - Enters a special state $q_{\text{print}}$, where contents of work tape, up to first blank, are copied to output tape, followed by blank as a separator.
    - Then machine continues.
    - No accept or reject states.
Recursively enumerable languages

- Starts with a blank work tape (no input).
- Prints a sequence of finite strings (possibly infinitely many) on output tape.
- It may print the same string more than once.
- If $E$ is an enumerator, then define
  \[ L(E) = \{ x \mid x \text{ is printed by } E \}. \]
- If $L = L(E)$ for some enumerator $E$, then we say that $L$ is recursively enumerable (r.e.).
Recursively enumerable languages

- Interesting connection between recursive enumerability and Turing recognizability:

- **Theorem 8:** L is recursively enumerable if and only if L is Turing-recognizable.

- **Proof:** $\Rightarrow$
  - Given E, an enumerator for L, construct Turing machine M to recognize L.
  - **M:** On input x:
    - M simulates E (on no input, as usual).
    - Whenever E prints, M checks to see if the new output is x.
    - If it ever sees x, M accepts.
    - Otherwise, M keeps going forever.
Recursively enumerable languages

- **Theorem 8**: \( L \) is recursively enumerable if and only if \( L \) is Turing-recognizable.
- **Proof**: \( \iff \)
  - Given \( M \), a Turing machine that recognizes \( L \), construct \( E \) to enumerate \( L \).
  - **Idea**:
    - Simulate \( M \) on all inputs.
    - If/when any simulated execution reaches \( q_{\text{acc}} \), print out the associated input.
  - As before, we can’t run \( M \) on all inputs sequentially, because some computations might not terminate.
  - So we must run them **in parallel**.
  - But this time we must run **infinitely many** computations, so we can’t just use a multitape Turing machine.
Recursively enumerable languages

• **Theorem 8:** L is recursively enumerable if and only if L is Turing-recognizable.

• **Proof:** \(\iff\)
  – Given M, a Turing machine that recognizes L, construct E to enumerate L.
  – Simulate M on all inputs; when any simulated execution reaches \(q_{\text{acc}}\), print out the associated input.
  – New trick: Dovetailing
    • Run 1 step for 1\textsuperscript{st} input string, \(\varepsilon\).
    • Run 2 steps for 1\textsuperscript{st} and 2\textsuperscript{nd} inputs, \(\varepsilon\) and 0.
    • Run 3 steps for 1\textsuperscript{st}, 2\textsuperscript{nd}, and 3\textsuperscript{rd} inputs, \(\varepsilon\), 0 and 1.
    • …
    • Run more and more steps for more and more inputs.
  – Eventually succeeds in reaching \(q_{\text{acc}}\) for each accepting computation of M, so enumerates all elements of L.
Recursively enumerable languages

• **Theorem 8:** L is recursively enumerable if and only if L is Turing-recognizable.

• **Proof:** $\Leftarrow$
  - Simulate M on all inputs; when any simulated execution reaches $q_{\text{acc}}$, print out the associated input.
  - Dovetail all computations of M.
  - Complicated bookkeeping, messy to work out in detail.
  - But can do algorithmically, hence on a Turing machine.
Turing Machines that solve problems for other domains besides strings
Turing Machines that solve problems for other domains

- [Sipser Section 4.1]
- Our examples of computability by Turing machines have so far involved properties of strings, and numbers represented by strings.
- We can also consider computability by TMs for other domains, such as graphs or DFAs.

Graphs:
- Consider the problem of whether a given graph has a cycle of length > 2.
- Can formalize this problem as a language (set of strings) by encoding graphs as strings over some finite alphabet.
- Graph = (V,E), V = vertices, E = edges, undirected.
Turing Machines that solve graph problems

• Consider the problem of whether a given graph has a cycle of length > 2.
• Formalize as a language (set of strings) by encoding graphs as strings over some finite alphabet.
• Graph = (V,E), V = vertices, E = edges, undirected.
• A standard encoding:
  – Vertices = positive integers (represented in binary)
  – Edges = pairs of positive integers
  – Graph = list of vertices, list of edges.
• Example: ( ( 1, 2, 3 ), ( ( 1, 2 ), (2, 3) ) )
• Write <G> for the encoding of G.
Turing Machines that solve graph problems

- Consider the problem of whether a given graph has a cycle of length $> 2$.
- Graph = $(V,E)$, $V$ = vertices, $E$ = edges, undirected.
- Write $<G>$ for the encoding of $G$.
- Using this representation for the input, we can write an algorithm to determine whether or not a given graph $G$ has a cycle, and formalize the algorithm using a Turing machine.
  - E.g., search and look for repeated vertices.
- So cyclicity is a decidable property of graphs.
Turing Machines that solve problems for other domains

- We can also consider computability for domains that are sets of machines:
- DFAs:
  - Encode DFAs using bit strings, by defining standard naming schemes for states and alphabet symbols.
  - Then a DFA tuple is again a list.
  - Example:

    Encode as:
    \[
    ( (1, 2), (0, 1), ( (1, 1, 1), (1, 0, 2), (2, 0, 2), (2, 1, 2) ), (1), (2) )
    \]
    
    - Encode the list using bit strings.
    - Write \(<M>\) for the encoding of M.
    - So we can define languages whose elements are (bit strings representing) DFAs.
Turing Machines that solve DFA problems

• Example: \( L_1 = \{ < M > \mid L(M) = \emptyset \} \) is Turing-decidable
• Elements of \( L_1 \) are bit-string representations of DFAs that accept nothing (emptiness problem).
• Already described an algorithm to decide this, based on searching to determine whether any accepting state is reachable from the start state.
• Could formalize this (painfully) as a Turing machine.
• Proves that \( L_1 \) is Turing-decidable.

• Similarly, all the other decision problems we considered for DFAs, NFAs, and regular expressions are Turing-decidable (not just Turing-recognizable).
• Just represent the inputs using standard encodings and formalize the algorithms that we’ve already discussed, using Turing machines.
Turing Machines that solve DFA problems

- **Example:** Equivalence for DFAs
  \[ L_2 = \{ < M_1, M_2 > | L(M_1) = L(M_2) \} \text{ is Turing-decidable.} \]
- Elements of \( L_2 \) are bit-string representations of pairs of DFAs that recognize the same language.
- Note that the domain we encode is **pairs of DFAs**.
- Already described an algorithm to decide this, based on testing inclusion both ways; to test whether \( L(M_1) \subseteq L(M_2) \), just test whether \( L(M_1) \cap (L(M_2))^c = \emptyset \).
- Formalize as a Turing machine.
- Proves that \( L_2 \) is Turing-decidable.
Turing Machines that solve DFA problems

• **Example:** Acceptance for DFAs
  \[ L_3 = \{ < M, w > \mid w \in L(M) \} \text{ is Turing-decidable.} \]

• Domain is (DFA, input) pairs.

• Algorithm simply runs M on w.

• Formalize as a Turing machine.

• Proves that \( L_3 \) is Turing-decidable.
Moving on…

• Now, things get more complicated: we consider inputs that are encodings of Turing machines rather than DFAs.

• In other words, we will discuss Turing machines that decide questions about Turing machines!
Undecidability of the Turing Machine Acceptance Problem
Undecidability of TM Acceptance Problem

- Now (and for a while), we will focus on showing that certain languages are not Turing-decidable, and that some are not even Turing-recognizable.
- It’s easy to see that such languages exist, based on cardinality considerations.
- Now we will show some specific languages are not Turing decidable, and not Turing-recognizable.
- These languages will express questions about Turing machines.
Undecidability of TM Acceptance

• We have been discussing decidability of problems involving DFAs, e.g.:
  \{< M > \mid M \text{ is a DFA and } L(M) = \emptyset \}, \text{ decidable by Turing machine that searches } M\text{'s digraph.}
  \{< M, w > \mid M \text{ is a DFA, } w \text{ is a word in } M\text{'s alphabet, and } w \in L(M) \}, \text{ decidable by a Turing machine that emulates } M \text{ on } w.

• Turing machines compute only on strings, but we can regard them as computing on DFAs by **encoding the DFAs as strings** (using a standard encoding).

• Now we consider **encoding Turing machines as strings**, and allowing other Turing machines to compute on these strings.

• Encoding of Turing machines: Standard state names, lists, etc., similar to DFA encoding.
  • \(<M>\) = encoding of Turing machine M.
  • \(<M, w>\) = encoding Turing machine + input string
  • Etc.
Problems we will consider

- $\text{Acc}_{\text{TM}} = \{ < M, w > | M \text{ is a (basic) Turing machine, } w \text{ is a word in } M\text{'s alphabet, and } M \text{ accepts } w \}$.
- $\text{Halt}_{\text{TM}} = \{ < M, w > | M \text{ is a Turing machine, } w \text{ is a word in } M\text{'s alphabet, and } M \text{ halts (either accepts or rejects) on } w \}$.
- $\text{Empty}_{\text{TM}} = \{ < M > | M \text{ is a Turing machine and } L(M) = \emptyset \}$
  - Recall: $L(M)$ refers to the set of strings $M$ accepts.
- Etc.

Thus, we can formulate questions about Turing machines as languages.

Then we can ask if they are Turing-decidable; that is, can some particular TM answer these questions about all (basic) TMs?

We’ll prove that they cannot.
The Acceptance Problem

- \( \text{Acc}_{\text{TM}} = \{ < M, w > | M \text{ is a (basic) Turing machine and } M \text{ accepts } w \} \).

- **Theorem 1**: \( \text{Acc}_{\text{TM}} \) is Turing-recognizable.

**Proof:**
- Construct a TM \( U \) that recognizes \( \text{Acc}_{\text{TM}} \).
- \( U \): On input \( < M, w > \):
  - Simulate \( M \) on input \( w \).
  - If \( M \) accepts, accept.
  - If \( M \) rejects, reject.
  - Otherwise, \( U \) loops forever.
- Then \( U \) accepts exactly \( < M, w > \) encodings for which \( M \) accepts \( w \).

- \( U \) is sometimes called a **universal Turing machine** because it runs all TMs.
  - Like an interpreter for a programming language.
The Acceptance Problem

- \( \text{Acc}_{\text{TM}} = \{ < M, w > | \text{M is a TM and M accepts } w \} \).
- \( U \): On input \(< M, w >\):
  - Simulate \( M \) on input \( w \).
  - If \( M \) accepts, accept.
  - If \( M \) rejects, reject.
  - Otherwise, \( U \) loops forever.
- \( U \) recognizes \( \text{Acc}_{\text{TM}} \).
- \( U \) is a universal Turing machine because it runs all TMs.
- \( U \) uses a fixed, finite set of states, and set of alphabet symbols, but still simulates TMs with arbitrarily many states and symbols.
  - All encoded using the fixed symbols, decoded during emulation.
The Acceptance Problem

- \( \text{Acc}_{\text{TM}} = \{ < M, w > | M \text{ is a TM and } M \text{ accepts } w \} \).
- **U**: On input \( < M, w > \):
  - Simulate \( M \) on input \( w \).
  - If \( M \) accepts, accept.
  - If \( M \) rejects, reject.
  - Otherwise, \( U \) loops forever.

- **U** recognizes \( \text{Acc}_{\text{TM}} \).
- **Does U decide** \( \text{Acc}_{\text{TM}} \) ?
  - **No.**
    - If \( M \) loops forever on \( w \), \( U \) loops forever on \( <M,w> \), never accepts or rejects.
    - To decide, \( U \) would have to detect when \( M \) is looping and reject.
    - Seems difficult…
Undecidability of Acceptance

• **Theorem 2:** \( \text{Acc}_{\text{TM}} \) is not Turing-decidable.

• **Proof:**
  – Assume that \( \text{Acc}_{\text{TM}} \) is Turing-decidable and produce a contradiction.
  – Similar to the diagonalization argument that shows that we can’t enumerate all languages.
  – Since (we assume) \( \text{Acc}_{\text{TM}} \) is Turing-decidable, there must be a particular TM \( H \) that decides \( \text{Acc}_{\text{TM}} \):
    • \( H(<M,w>) \):
      – accepts if \( M \) accepts \( w \),
      – rejects if \( M \) rejects \( w \),
      – rejects if \( M \) loops on \( w \).
Undecidability of Acceptance

• **Theorem 2:** $\text{Acc}_T^M$ is not Turing-decidable.

• **Proof, cont’d:**
  – $H(<M,w>)$ accepts if $M$ accepts $w$, rejects if $M$ rejects $w$ or if $M$ loops on $w$.
  – Use $H$ to construct another TM $H'$ that decides a special case of the same language.
  – Instead of considering whether $M$ halts on an arbitrary $w$, just consider $M$ on its own representation:
    – $H'(\langle M \rangle)$:
      • accepts if $M$ accepts $\langle M \rangle$,
      • rejects if $M$ rejects $\langle M \rangle$ or if $M$ loops on $\langle M \rangle$.
  – If $H$ exists, then so does $H'$: $H'$ simply runs $H$ on certain arguments.
Undecidability of Acceptance

- **Theorem 2**: $\text{Acc}_{\text{TM}}$ is not Turing-decidable.
- **Proof, cont’d:**
  - $H'(\langle M \rangle)$:
    - accepts if $M$ accepts $\langle M \rangle$,
    - rejects if $M$ rejects $\langle M \rangle$ or if $M$ loops on $\langle M \rangle$.
  - Now define $D$ (the diagonal machine) to do the opposite of $H'$:
    - $D(\langle M \rangle)$:
      - rejects if $M$ accepts $\langle M \rangle$,
      - accepts if $M$ rejects $\langle M \rangle$ or if $M$ loops on $\langle M \rangle$.
  - If $H'$ exists, then so does $D$: $D$ runs $H'$ and outputs the opposite.
Undecidability of Acceptance

• **Theorem 2:** \( \text{Acc}_{TM} \) is not Turing-decidable.
• **Proof, cont’d:**
  – \( D(<M>) \):
    • rejects if \( M \) accepts \(<M>\),
    • accepts if \( M \) rejects \(<M>\) or if \( M \) loops on \(<M>\).
  – Now, what happens if we run \( D \) on \(<D>\)?
  – Plug in \( D \) for \( M \):
    – \( D(<D>) \):
      • rejects if \( D \) accepts \(<D>\),
      • accepts if \( D \) rejects \(<D>\) or if \( D \) loops on \(<D>\).
  – Then \( D \) accepts \(<D>\) if and only if \( D \) does not accept \(<D>\), contradiction!
  – So \( \text{Acc}_{TM} \) is not Turing-decidable.
Diagonalization Proofs

- This undecidability proof for Acc$_{TM}$ is an example of a diagonalization proof.
- Earlier, we used diagonalization to show that the set of all languages is not countable.
- Consider a big matrix, with TMs labeling rows and strings that represent TMs labeling columns.
- The major diagonal describes results for M(<M>), for all M.
- D is a diagonal machine, constructed explicitly to differ from the diagonal entries: D(<M>)’s result differs from M(<M>)’s.
- Implies that D itself can’t appear as a label for a row in the matrix, a contradiction since the matrix is supposed to include all TMs.
Summary: $\text{Acc}_{\text{TM}}$

- We have shown that $\text{Acc}_{\text{TM}} = \{ < M, w > \mid M \text{ is a Turing machine and } M \text{ accepts } w \}$ is Turing-recognizable but not Turing-decidable.

- **Corollary:** $(\text{Acc}_{\text{TM}})^c$ is not Turing-recognizable.

- **Proof:**
  - By Theorem 4.
  - If $\text{Acc}_{\text{TM}}$ and $(\text{Acc}_{\text{TM}})^c$ were both Turing-recognizable, then $\text{Acc}_{\text{TM}}$ would be Turing-decidable.
Undecidability of the Turing Machine Halting Problem
The Halting Problem

- $\text{Halt}_{TM} = \{ < M, w > \mid M \text{ is a Turing machine and } M \text{ halts on (either accepts or rejects) } w \}$. 
- Compare with $\text{Acc}_{TM} = \{ < M, w > \mid M \text{ is a Turing machine and } M \text{ accepts } w \}$. 
- Terminology caution: Sipser calls $\text{Acc}_{TM}$ the “halting problem”, and calls $\text{Halt}_{TM}$ just $\text{Halt}_{TM}$.
- Theorem: $\text{Halt}_{TM}$ is not Turing-decidable.
- Proof:
  - Let’s not use diagonalization.
  - Rather, take advantage of diagonalization work already done for $\text{Acc}_{TM}$, using new method: reduction.
  - Prove that, if we could decide $\text{Halt}_{TM}$, then we could decide $\text{Acc}_{TM}$.
  - Reduction is a very powerful, useful technique for showing undecidability; we’ll use it several times.
  - Also useful (later) to show inherent complexity results.
The Halting Problem

- \( \text{Halt}_\text{TM} = \{ \langle M, w \rangle \mid M \text{ halts on (accepts or rejects) } w \} \).
- **Theorem:** \( \text{Halt}_\text{TM} \) is not Turing-decidable.
- **Proof:**
  - Suppose for contradiction that \( \text{Halt}_\text{TM} \) is Turing-decidable, say by Turing machine \( \text{R} \):
    - \( \text{R}(\langle M, w \rangle) \):
      - accepts if \( M \) halts on (accepts or rejects) \( w \),
      - rejects if \( M \) loops (neither accepts nor rejects) on \( w \).
  - Using \( \text{R} \), define new TM \( \text{S} \) to decide \( \text{Acc}_\text{TM} \):
    - \( \text{S} \): On input \( \langle M, w \rangle \):
      - Run \( \text{R} \) on \( \langle M, w \rangle \); \( \text{R} \) must either accept or reject; can’t loop, by definition of \( \text{R} \).
      - If \( \text{R} \) accepts then \( M \) must halt (accept or reject) on \( w \). Then simulate \( M \) on \( w \), knowing this must terminate. If \( M \) accepts, accept. If \( M \) rejects, reject.
      - If \( \text{R} \) rejects, then reject.
The Halting Problem

- **Theorem:** \( \text{Halt}^\text{TM} \) is not Turing-decidable.
- **Proof:**
  - Suppose \( \text{Halt}^\text{TM} \) is Turing-decidable by TM R.
    - S: On input \(<M,w>\>:
      - Run R on \(<M,w>\>\; R \text{ must either accept or reject; can’t loop, by definition of } R.
      - If R accepts then \( M \) must halt (accept or reject) on \( w \). Then simulate \( M \) on \( w \), knowing this must terminate. If \( M \) accepts, accept. If rejects, reject.
      - If R rejects, then reject.
    - Claim S decides \( \text{Acc}^\text{TM} \): 3 cases:
      - If \( M \) accepts \( w \), then R accepts \(<M,w>\>, and the simulation leads S to accept.
      - If \( M \) rejects \( w \), then R accepts \(<M,w>\>, and the simulation leads S to reject.
      - If \( M \) loops on \( w \), then R rejects \(<M,w>\>, and S rejects.
      - That’s what’s supposed to happen in three cases, for \( \text{Acc}^\text{TM} \).
The Three Cases

R distinguishes these two cases

M accepts w    M rejects w    M loops on w

Now S knows that M will terminate, simulates M on w.

S knows that M loops, rejects.

S distinguishes these two cases.
The Halting Problem

- **Theorem:** $\text{Halt}_{\text{TM}}$ is not Turing-decidable.
- **Proof:**
  - Suppose $\text{Halt}_{\text{TM}}$ is Turing-decidable by TM $R$.
  - $S$: On input $<M,w>$:
    - Run $R$ on $<M,w>$; $R$ must either accept or reject; can’t loop, by definition of $R$.
    - If $R$ accepts then $M$ must halt (accept or reject) on $w$. Then simulate $M$ on $w$, knowing this must terminate. If $M$ accepts, accept. If rejects, reject.
    - If $R$ rejects, then reject.
  - $S$ decides $\text{Acc}_{\text{TM}}$.
  - So $\text{Acc}_{\text{TM}}$ is decidable, contradiction.
  - Therefore, $\text{Halt}_{\text{TM}}$ is not Turing-decidable.
The Halting Problem

• **Theorem:** $\text{Halt}_{TM}$ is not Turing-decidable.
• Also:
  • **Theorem:** $\text{Halt}_{TM}$ is Turing-recognizable.
• So:
  • **Corollary:** $(\text{Halt}_{TM})^c$ is not Turing-recognizable.
Next time…

• More undecidable problems:
  – About Turing machines:
    • Emptiness, etc.
  – About other things:
    • Post Correspondence Problem (a string matching problem).

• **Reading:** Sipser Sections 4.2, 5.1.