Computational Models - Lecture 8
Fall 04/05

Computable functions
Computational Models - Lecture 8
Fall 04/05

- Computable functions
- Reductions
Computational Models - Lecture 8
Fall 04/05

- Computable functions
- Reductions
- Reductions by computational histories
Computable functions

Reductions

Reductions by computational histories

Reducing $A$ to $B$ by Mapping reductions
Computable functions

Reductions

Reductions by computational histories

Reducing $A$ to $B$ by Mapping reductions

More undecidable languages
Computable functions

Reductions

Reductions by computational histories

Reducing $A$ to $B$ by Mapping reductions

More undecidable languages

Recursive Inseparability
Computable functions
Reductions
Reductions by *computational histories*
Reducing $A$ to $B$ by *Mapping reductions*
More undecidable languages
Recursive Inseparability
Computable functions

Reductions

Reductions by computational histories

Reducing $A$ to $B$ by Mapping reductions

More undecidable languages

Recursive Inseparability

Sipser’s book, Chapter 5, Sections 5.1, 5.3
More Undecidable Problems

We have already

- Established Turing Machines as the gold standard of computers and computability . . .
More Undecidable Problems

We have already

- Established Turing Machines as the gold standard of computers and computability . . .
- seen examples of solvable problems . . .
More Undecidable Problems

We have already

- Established Turing Machines as the gold standard of computers and computability . . .
- seen examples of solvable problems . . .
- and saw one problem, $A_{TM}$, that is computationally unsolvable.
More Undecidable Problems

We have already

- Established Turing Machines as the gold standard of computers and computability . . .
- seen examples of solvable problems . . .
- and saw one problem, $A_{TM}$, that is computationally unsolvable.

In this lecture, we look at other computationally unsolvable problems, and introduce the notion of reductions. Furthermore, we introduce the techniques of reductions by computational histories and of mapping reducibilities for prove that languages are undecidable/non-enumerable.
Computable Functions

A TM computes a total function

\[ f : \Sigma^* \rightarrow \Sigma^* \]

if the TM
Computable Functions

A TM computes a total function

\[ f : \Sigma^* \longrightarrow \Sigma^* \]

if the TM

starts with input \( w \), and
Computable Functions

A TM computes a total function

\[ f : \Sigma^* \rightarrow \Sigma^* \]

if the TM

- starts with input \( w \), and
- halts with only \( f(w) \) on tape.
Computable Functions

A TM computes a total function

\[ f : \Sigma^* \longrightarrow \Sigma^* \]

if the TM

- starts with input \( w \), and
- halts with only \( f(w) \) on tape.

The definition can be extended to functions of more than one variable, where some special separator symbol indicates end of one variable and beginning of next.
Computable Functions

A TM computes a partial function

\[ f : \Sigma^* \rightarrow (\Sigma^* \cup \bot) \]

if the TM
Computable Functions

A TM computes a partial function

\[ f : \Sigma^* \longrightarrow (\Sigma^* \cup \bot) \]

if the TM

\[ \text{starts with input } w, \text{ and} \]
Computable Functions

A TM computes a partial function

\[ f : \Sigma^* \rightarrow (\Sigma^* \cup \bot) \]

if the TM

- starts with input \( w \), and
- if \( f(w) \) is defined, TM halts with only \( f(w) \) on tape,
Computable Functions

A TM computes a partial function

\[ f : \Sigma^* \rightarrow (\Sigma^* \cup \perp) \]

if the TM

- starts with input \( w \), and
- if \( f(w) \) is defined, TM halts with only \( f(w) \) on tape,
- if \( f(w) \) is undefined, TM does not halt.
Computable Functions

A TM computes a partial function

\[ f : \Sigma^* \rightarrow (\Sigma^* \cup \bot) \]

if the TM

- starts with input \( w \), and
- if \( f(w) \) is defined, TM halts with only \( f(w) \) on tape,
- if \( f(w) \) is undefined, TM does not halt.

Computable functions are also called (total or partial) recursive functions.
Computable Functions

**Claim:** All the usual arithmetic functions on integers are computable.
Computable Functions

**Claim:** All the usual arithmetic functions on integers are computable.

These include addition, subtraction, multiplication, division (quotient and remainder), exponentiation, roots (to a specified precision).
Computable Functions

**Claim:** All the usual arithmetic functions on integers are computable.

These include addition, subtraction, multiplication, division (quotient and remainder), exponentiation, roots (to a specified precision).

Even non-arithmetic functions, like logarithms and trigonometric functions, can be computed (to a specified precision), using Taylor expansion or other numeric mathematic techniques.
Computable Functions

Claim: All the usual arithmetic functions on integers are computable.

These include addition, subtraction, multiplication, division (quotient and remainder), exponentiation, roots (to a specified precision).

Even non-arithmetic functions, like logarithms and trigonometric functions, can be computed (to a specified precision), using Taylor expansion or other numeric mathematic techniques.

Exercise: Design a TM that on input \( \langle m, n \rangle \), halts with \( \langle m + n \rangle \) on tape.
Computable Functions

A useful class of functions modifies TM descriptions. For example:

On input \( w \):

- if \( w = \langle M \rangle \) for some TM,
Computable Functions

A useful class of functions modifies TM descriptions. For example:

On input $w$:

- if $w = \langle M \rangle$ for some TM,
- construct $\langle M' \rangle$, where
Computable Functions

A useful class of functions modifies TM descriptions. For example:

On input $w$:

- if $w = \langle M \rangle$ for some TM,
- construct $\langle M' \rangle$, where
- $L(M') = L(M)$, but
Computable Functions

A useful class of functions modifies TM descriptions. For example:

On input $w$:

- if $w = \langle M \rangle$ for some TM,
  - construct $\langle M' \rangle$, where
    - $L(M') = L(M)$, but
    - $M'$ never tries to move off LHS of tape.
Computable Functions

A useful class of functions modifies TM descriptions. For example:

On input $w$:

- if $w = \langle M \rangle$ for some TM, construct $\langle M' \rangle$, where $L(M') = L(M)$, but $M'$ never tries to move off LHS of tape.
- otherwise write $\varepsilon$ and halt.
Computable Functions

A useful class of functions modifies TM descriptions. For example:

On input $w$:

- if $w = \langle M \rangle$ for some TM,
  - construct $\langle M' \rangle$, where
  - $L(M') = L(M)$, but
  - $M'$ never tries to move off LHS of tape.
- otherwise write $\epsilon$ and halt.

Done in recitation this morning.
Reducibility

Example:

- Finding your way around a new city
Reducibility

Example:
- Finding your way around a new city
- reduces to . . .
Reducibility

Example:

- Finding your way around a new city
- reduces to . . .
- obtaining a city map.
Reducibility, In Our Context

Always involves two problems, $A$ and $B$. 
Reducibility, In Our Context

Always involves two problems, $A$ and $B$.

**Desired Property:** If $A$ reduces to $B$, then any solution of $B$ can be used to find a solution of $A$. 
Reducibility, In Our Context

Always involves two problems, $A$ and $B$.

**Desired Property:** If $A$ reduces to $B$, then any solution of $B$ can be used to find a solution of $A$.

**Remark:** This property says nothing about solving $A$ by itself or $B$ by itself.
Examples

Reductions:

- Traveling from Boshton to Paris ...
Examples

Reductions:
- Traveling from Boshten to Paris . . .
- buying plane ticket . . .
Examples

Reductions:

- Traveling from Boshton to Paris . . .
- buying plane ticket . . .
- earning the money for that ticket . . .
Examples

Reductions:

- Traveling from Boshton to Paris . . .
- buying plane ticket . . .
- earning the money for that ticket . . .
- finding a job
  (or getting the $s from mom and dad . . .)
Examples

Reductions:

- Measuring area of rectangle ...
Examples

Reductions:

- Measuring area of rectangle . . .
- measuring lengths of sides.
Examples

Reductions:
- Measuring area of rectangle . . .
- measuring lengths of sides.

Also:
Examples

Reductions:
- Measuring area of rectangle . . .
- measuring lengths of sides.

Also:
- Solving a system of linear equations . . .
- inverting a matrix.
Reducibility

If $A$ is reducible to $B$, then

- $A$ cannot be harder than $B$
Reducibility

If $A$ is reducible to $B$, then

- $A$ cannot be harder than $B$
- if $B$ is decidable, so is $A$. 
Reducibility

If $A$ is reducible to $B$, then

- $A$ cannot be harder than $B$
- if $B$ is decidable, so is $A$.
- if $A$ is undecidable and reducible to $B$, then $B$ is undecidable.
Reduction via Computation Histories

Important technique for proving undecidability.

- Useful for testing existence of some objects.
Reduction via Computation Histories

Important technique for proving undecidability.

- Useful for testing existence of some objects.
- For example, basis for proof of undecidability in Hilbert’s tenth problem,
Reduction via Computation Histories

Important technique for proving undecidability.

- Useful for testing existence of some objects.
- For example, basis for proof of undecidability in Hilbert’s tenth problem,
- where "object" is integral root of polynomial.
Reduction via Computation Histories

Important technique for proving undecidability.

- Useful for testing existence of some objects.
- For example, basis for proof of undecidability in Hilbert’s tenth problem,
- where "object" is integral root of polynomial.
- Other examples: Does a linear bounded TM accept the empty language?
Reduction via Computation Histories

Important technique for proving undecidability.

- Useful for testing existence of some objects.
- For example, basis for proof of undecidability in Hilbert’s tenth problem,
  where "object" is integral root of polynomial.
- Other examples: Does a linear bounded TM accept the empty language?
- Does a context gree grammar generate $\Sigma^*$?
Reminder: Configurations

Configuration: $1011q_70111$

means:

- state is $q_7$
- LHS of tape is 1011
- RHS of tape is 0111
- head is on RHS 0
Configurations

configuration \( uaq_i bv \) yields \( uq_j acv \) if
\[
\delta(q_i, b) = (q_j, c, L)
\]
Configurations

- configuration \( uaq_i bv \) yields \( uq_j acv \) if \
  \[ \delta(q_i, b) = (q_j, c, L) \]

- Of course, \( uaq_i bv \) yields \( uacq_j v \) if \
  \[ \delta(q_i, b) = (q_j, c, R) \]
Configurations

- configuration $uaq_i bv$ yields $uq_j acv$ if $\delta(q_i, b) = (q_j, c, L)$
- Of course, $uaq_i bv$ yields $uacq_j v$ if $\delta(q_i, b) = (q_j, c, R)$
- Special case (left end of tape): $q_i bv$ yields $q_j cv$ if $\delta(q_i, b) = (q_j, c, blueL)$. 
Computation Histories

Let $M$ be a TM and $w$ an input string.

- An accepting computation history for $M$ on $w$ is a sequence $C_1, C_2, \ldots, C_\ell$, where
Computation Histories

Let $M$ be a TM and $w$ an input string.

- An accepting computation history for $M$ on $w$ is a sequence $C_1, C_2, \ldots, C_\ell$, where
- $C_1$ is the starting configuration of $M$ on $w$
Computation Histories

Let $M$ be a TM and $w$ an input string.

- An **accepting** computation history for $M$ on $w$ is a sequence $C_1, C_2, \ldots, C_\ell$, where
  - $C_1$ is the starting configuration of $M$ on $w$
  - $C_\ell$ is an accepting configuration of $M$,
Computation Histories

Let $M$ be a TM and $w$ an input string.

- An accepting computation history for $M$ on $w$ is a sequence $C_1, C_2, \ldots, C_\ell$, where
  - $C_1$ is the starting configuration of $M$ on $w$
  - $C_\ell$ is an accepting configuration of $M$
  - each $C_i$ yields $C_{i+1}$ according to the transition function.
Computation Histories

Let $M$ be a TM and $w$ an input string.

- An **accepting** computation history for $M$ on $w$ is a sequence $C_1, C_2, \ldots, C_\ell$, where
  - $C_1$ is the starting configuration of $M$ on $w$
  - $C_\ell$ is an accepting configuration of $M$,
  - each $C_i$ yields $C_{i+1}$ according to the transition function.

- An **rejecting** computation history for $M$ on $w$ is the same, except
Computation Histories

Let $M$ be a TM and $w$ an input string.

- An **accepting** computation history for $M$ on $w$ is a sequence $C_1, C_2, \ldots, C_\ell$, where
  - $C_1$ is the starting configuration of $M$ on $w$
  - $C_\ell$ is an accepting configuration of $M$,
  - each $C_i$ yields $C_{i+1}$ according to the transition function.

- An **rejecting** computation history for $M$ on $w$ is the same, except
  - $C_\ell$ is a rejecting configuration of $M$, 

Remarks
Remarks

- Computation sequences are finite.
Remarks

- Computation sequences are finite.
- If $M$ does not halt on $w$, no accepting or rejecting computation history exists.
Remarks

- Computation sequences are finite.
- If $M$ does not halt on $w$, no accepting or rejecting computation history exists.
- Notion is useful for both deterministic (one history) and non-deterministic (many histories) TMs.
Linear Bounded Automata

A restricted form of TM.
Linear Bounded Automata

- A restricted form of TM.
- Cannot move off portion of tape containing input
Linear Bounded Automata

- A restricted form of TM.
- Cannot move off portion of tape containing input
- Rejects attempts to move head beyond input
Linear Bounded Automata

- A restricted form of TM.
- Cannot move off portion of tape containing input
- Rejects attempts to move head beyond input
- Size of input determines size of memory

Slides modified by Benny Chor, based on original slides by Maurice Herlihy, Brown University.
**Linear Bounded Automata**

**Question:** Why linear?

**Answer:** Using a tape alphabet larger than the input alphabet increases memory by a constant factor.
Linear Bounded Automata

Believe it or not, LBAs are quite powerful.
The *deciders* for

are all LBAs.
Linear Bounded Automata

Believe it or not, LBAs are quite powerful. The deciders for $A_{\text{DFA}}$ (does DFA accept?) are all LBAs.
Linear Bounded Automata

Believe it or not, LBAs are quite powerful. The deciders for

- $A_{DFA}$ (does DFA accept?)
- $A_{CFG}$ (is string in CFG?)

are all LBAs.
Linear Bounded Automata

Believe it or not, LBAs are quite powerful. The deciders for

- $A_{\text{DFA}}$ (does DFA accept?)
- $A_{\text{CFG}}$ (is string in CFG?)
- $\text{EMPTY}_{\text{DFA}}$ (is DFA trivial?)

are all LBAs.
Linear Bounded Automata

Believe it or not, LBAs are quite powerful. The deciders for

- $A_{DFA}$ (does DFA accept?)
- $A_{CFG}$ (is string in CFG?)
- $\text{EMPTY}_{DFA}$ (is DFA trivial?)
- $\text{EMPTY}_{CFG}$ (is CFG empty?)

are all LBAs.

Every CFL can be decided by a LBA.
Linear Bounded Automata

Believe it or not, LBAs are quite powerful. The **deciders** for

- $A_{DFA}$ (does DFA accept?)
- $A_{CFG}$ (is string in CFG?)
- $\text{EMPTY}_{DFA}$ (is DFA trivial?)
- $\text{EMPTY}_{CFG}$ (is CFG empty?)

are all LBAs.

Every CFL can be decided by a LBA.

Not easy to find a **natural, decidable language** that **cannot** be decided by an LBA.
Linear Bounded Automata

Believe it or not, LBAs are quite powerful. The deciders for

- $A_{DFA}$ (does DFA accept?)
- $A_{CFG}$ (is string in CFG?)
- $\text{EMPTY}_{DFA}$ (is DFA trivial?)
- $\text{EMPTY}_{CFG}$ (is CFG empty?)

are all LBAs.

Every CFL can be decided by a LBA.

Not easy to find a natural, decidable language that cannot be decided by an LBA.
A Language

Define

$$A_{LBA} = \{ \langle M, w \rangle | M \text{ is an LBA that accepts } w \}$$

**Question:** Is $A_{LBA}$ decidable?
Lemma

Lemma: Let $M$ be a LBA with

- $q$ states
- $g$ symbols in tape alphabet

On an input of size $n$, LBA has exactly $qng^n$ distinct configurations, because Configuration involves:
Lemma

**Lemma:** Let $M$ be a LBA with

- $q$ states
- $g$ symbols in tape alphabet

On an input of size $n$, LBA has exactly $qng^n$ distinct configurations, because Configuration involves:

- control state ($q$ possibilities)
Lemma

**Lemma:** Let $M$ be a LBA with

- $q$ states
- $g$ symbols in tape alphabet

On an input of size $n$, LBA has exactly $qng^n$ distinct configurations, because Configuration involves:

- control state ($q$ possibilities)
- head position ($n$ possibilities)
Lemma

**Lemma:** Let $M$ be a LBA with

- $q$ states
- $g$ symbols in tape alphabet

On an input of size $n$, LBA has exactly $qng^n$ distinct configurations, because Configuration involves:

- control state ($q$ possibilities)
- head position ($n$ possibilities)
- tape contents ($g^n$ possibilities)
Theorem

**Theorem:** $A_{LBA}$ is decidable.

**Idea:**

- Simulate $M$ on $w$. 
Theorem

Theorem: \( A_{LBA} \) is decidable.

Idea:

- Simulate \( M \) on \( w \).
- But what do we do if \( M \) loops?
Theorem

Theorem: \( A_{LBA} \) is decidable.

Idea:

- Simulate \( M \) on \( w \).
- But what do we do if \( M \) loops?
- Must detect looping and reject.
Theorem

**Theorem:** $A_{\text{LBA}}$ is decidable.

**Idea:**

- Simulate $M$ on $w$.
- But what do we do if $M$ loops?
- Must detect looping and reject.
- $M$ loops if and only if it repeats a configuration.
Theorem

**Theorem:** \( A_{LBA} \) is decidable.

**Idea:**
- Simulate \( M \) on \( w \).
- But what do we do if \( M \) loops?
- Must detect looping and reject.
- \( M \) loops if and only if it repeats a configuration.
- **Why?** And is this also true of “regular” TMs?
Theorem

**Theorem:** \( A_{LBA} \) is decidable.

**Idea:**
- Simulate \( M \) on \( w \).
- But what do we do if \( M \) loops?
- Must detect looping and reject.
- \( M \) loops if and only if it repeats a configuration.
- **Why?** And is this also true of “regular” TMs?
- By pigeon hole, if our LBA \( M \) runs long enough, it must repeat a configuration!
**Theorem**

**Theorem:** $A_{LBA}$ is decidable.

On input $\langle M, w \rangle$, where $M$ is an LBA and $w \in \Sigma^*$

1. Simulate $M$ on $w$,
2. While maintaining a counter.
3. Counter incremented by 1 per each simulated step (of $M$).
4. Keep simulating $M$ for $qng^n$ steps, or until it halts (whichever comes first)
5. If $M$ has halted, accept $w$ if it accepted by $M$, and reject $w$ if it rejected by $M$.
6. reject $w$ if counter limit reached ($M$ has not halted).
More LBAs

Here is a related problem.

\[
\text{EMPTY}_{\text{LBA}} = \{ \langle M \rangle | M \text{ is an LBA and } L(M) = \emptyset \}\]

**Question**: Is \text{EMPTY}_{\text{LBA}} decidable?

Surprisingly though, LBAs do have undecidable problems too!
More LBAs

\[ \text{EMPTY}_{LBA} = \{ \langle M \rangle | M \text{ is an LBA and } L(M) = \emptyset \} \]

**Theorem:** \( \text{EMPTY}_{LBA} \) is undecidable.

Proof by reduction using computation histories.
More LBAs

\[
\text{EMPTY}_{LBA} = \{\langle M \rangle | M \text{ is an LBA and } L(M) = \emptyset\}
\]

**Theorem:** \(\text{EMPTY}_{LBA}\) is undecidable.

Proof by reduction from \(A_{TM}\).

If \(\text{EMPTY}_{LBA}\) were decidable, then \(A_{TM}\) would also be.

**Question:** Suppose that \(\text{EMPTY}_{LBA}\) is decidable. How can we use this supposition to decide \(A_{TM}\)?

Let \(R\) be a decider for the language \(\text{EMPTY}_{LBA}\).
More LBAs

Given $M$ and $w$, we will construct an LBA, $B$.

$L(B)$ will contain exactly all accepting computation histories for $M$ on $w$
More LBAs

Given $M$ and $w$, we will construct an LBA, $B$.

- $L(B)$ will contain exactly all accepting computation histories for $M$ on $w$
- $M$ accepts $w$ iff $L(B) \neq \emptyset$. 
More LBAs

Given \( M \) and \( w \), we will construct an LBA, \( B \).

- \( L(B) \) will contain exactly all accepting computation histories for \( M \) on \( w \)

- \( M \) accepts \( w \) iff \( L(B) \neq \emptyset \).

- Will use \( R \) to decide whether \( L(B) = \emptyset \).
More LBAs

Given $M$ and $w$, we will construct an LBA, $B$.

- $L(B)$ will contain exactly all accepting computation histories for $M$ on $w$
- $M$ accepts $w$ iff $L(B) \neq \emptyset$.
- Will use $R$ to decide whether $L(B) = \emptyset$.
- Then we can decide whether $M$ accepts $w$. 
More LBAs

It is not enough to show that $B$ exists. We must show that a TM can construct $\langle B \rangle$ from $\langle M, w \rangle$.

Assume an accepting computation history is presented as a string:

$$
\# C_1 \# C_2 \# C_3 \# \cdots \# C_\ell \#
$$

with descriptions of configurations separated by # delimiters.
The LBA

The LBA $B$ works as follows:
On input $x$, the LBA $B$:

- breaks $x$ according to the $\#$ delimiters
The LBA

The LBA $B$ works as follows:

On input $x$, the LBA $B$:

- breaks $x$ according to the $\#$ delimiters
- identifies strings $C_1, C_2, \ldots, C_\ell$. 
The LBA

The LBA $B$ works as follows:

On input $x$, the LBA $B$:

- breaks $x$ according to the $\#$ delimiters
- identifies strings $C_1, C_2, \ldots, C_\ell$.
- then checks that following conditions hold:
The LBA

The LBA $B$ works as follows:

On input $x$, the LBA $B$:

- breaks $x$ according to the $\#$ delimiters
- identifies strings $C_1, C_2, \ldots, C_\ell$.
- then checks that following conditions hold:
  - Each $C_i$ are a configuration of $M$
The LBA

The LBA $B$ works as follows:
On input $x$, the LBA $B$:
- breaks $x$ according to the $\#$ delimiters
- identifies strings $C_1, C_2, \ldots, C_\ell$.
- then checks that following conditions hold:
  - Each $C_i$ are a configuration of $M$
  - $C_1$ is the start configuration of $M$ on $w$
The LBA

The LBA $B$ works as follows:

On input $x$, the LBA $B$:

- breaks $x$ according to the $\#$ delimiters
- identifies strings $C_1, C_2, \ldots, C_\ell$.

then checks that following conditions hold:

- Each $C_i$ are a configuration of $M$
- $C_1$ is the start configuration of $M$ on $w$
- Every $C_{i+1}$ follows from $C_i$ according to $M$
The LBA

The LBA $B$ works as follows:

On input $x$, the LBA $B$:

- breaks $x$ according to the $#$ delimiters
- identifies strings $C_1, C_2, \ldots, C_\ell$.
- then checks that following conditions hold:
  - Each $C_i$ are a configuration of $M$
  - $C_1$ is the start configuration of $M$ on $w$
  - Every $C_{i+1}$ follows from $C_i$ according to $M$
  - $C_\ell$ is an accepting configuration
The LBA

Checking that each $C_i$ is a **configuration** of $M$ is easy: All it means is that $C_i$ includes exactly one $q$ symbols.
The LBA

- Checking that each $C_i$ is a configuration of $M$ is easy: All it means is that $C_i$ includes exactly one $q$ symbols.

- Checking that $C_1$ is the start configuration on $w$, $q_0w_1w_2\cdots w_n$, is easy, because the string $w$ was “wired into” $B$. 
The LBA

- Checking that each $C_i$ is a configuration of $M$ is easy: All it means is that $C_i$ includes exactly one $q$ symbols.

- Checking that $C_1$ is the start configuration on $w$, $q_0 w_1 w_2 \cdots w_n$, is easy, because the string $w$ was “wired into” $B$.

- Checking that $C_\ell$ is an accepting configuration is easy, because $C_\ell$ must include the accepting state $q_a$. 
The LBA

- Checking that each $C_i$ is a configuration of $M$ is easy: All it means is that $C_i$ includes exactly one $q$ symbols.

- Checking that $C_1$ is the start configuration on $w$, $q_0 w_1 w_2 \cdots w_n$, is easy, because the string $w$ was “wired into” $B$.

- Checking that $C_\ell$ is an accepting configuration is easy, because $C_\ell$ must include the accepting state $q_a$.

- The only hard part is checking that each $C_{i+1}$ follows from $C_i$ by $M$’s transition function.
The Hard Part

Checking that for all $i$, $C_{i+1}$ follows from $C_i$ by $M$’s transition function.

- $C_i$ and $C_{i+1}$ almost identical, except for positions under head and adjacent to head.
- These positions should updated according to transition function.

Do this verification by
The Hard Part

Checking that for all $i$, $C_{i+1}$ follows from $C_i$ by $M$’s transition function.

- $C_i$ and $C_{i+1}$ almost identical, except for positions under head and adjacent to head.
- These positions should updated according to transition function.

Do this verification by

- zig-zagging between corresponding positions of $C_i$ and $C_{i+1}$.
The Hard Part

Checking that for all \( i \), \( C_{i+1} \) follows from \( C_i \) by \( M \)'s transition function.

- \( C_i \) and \( C_{i+1} \) almost identical, except for positions under head and adjacent to head.
- These positions should be updated according to transition function.

Do this verification by

- zig-zagging between corresponding positions of \( C_i \) and \( C_{i+1} \).
- use “dots” on tape to mark current position
The Hard Part

Checking that for all $i$, $C_{i+1}$ follows from $C_i$ by $M$’s transition function.

- $C_i$ and $C_{i+1}$ almost identical, except for positions under head and adjacent to head.
- These positions should be updated according to transition function.

Do this verification by

- zig-zagging between corresponding positions of $C_i$ and $C_{i+1}$.
- use “dots” on tape to mark current position
- all this can be done in space allocated by input $x$
Important!

The LBA, $B$, accepts the string $x$ if and only if $x$ equals an accepting computation history of $M$ on $w$. 
Important!

The LBA, $B$, accepts the string $x$ if and only if $x$ equals an accepting computation history of $M$ on $w$. Therefore $L(B)$ is either empty or a singleton $\{x\}$. 
Important!

The LBA, $B$, accepts the string $x$ if and only if $x$ equals an accepting computation history of $M$ on $w$. Therefore $L(B)$ is either empty or a singleton $\{x\}$. We construct $B$ in order to feed it to the claimed decider, $R$, of $\text{EMPTY}_{\text{LBA}}$ (which we assume to exist).
Important!

The LBA, $B$, accepts the string $x$ if and only if $x$ equals an accepting computation history of $M$ on $w$. Therefore $L(B)$ is either empty or a singleton $\{x\}$. We construct $B$ in order to feed it to the claimed decider, $R$, of $\text{EMPTY}_{\text{LBA}}$ (which we assume to exist).

Once this decider returns its answer, we invert this answer to decide whether $M$ accepts $w$. 
The Proof

Suppose TM \( R \) decides \( \text{EMPTY}_{\text{LBA}} \).

Define TM \( S \) that decides \( A_{\text{TM}} \):

On input \( \langle M, w \rangle \)

1. Construct LBA, \( B \), from \( M \) and \( w \) as described above.
2. Run \( R \) on \( \langle B \rangle \).
3. if \( R \) rejects, accept; if \( R \) accepts, reject.
The Proof

Suppose TM \( R \) decides \( \text{EMPTY}_{\text{LBA}} \).

Define TM \( S \) that decides \( A_{\text{TM}} \):

On input \( \langle M, w \rangle \)

1. Construct LBA, \( B \), from \( M \) and \( w \) as described above.

2. Run \( R \) on \( \langle B \rangle \).

3. if \( R \) rejects, accept; if \( R \) accepts, reject.
The Proof

Suppose TM $R$ decides $\text{EMPTY}_{\text{LBA}}$. Define TM $S$ that decides $A_{\text{TM}}$:

On input $\langle M, w \rangle$

1. Construct LBA, $B$, from $M$ and $w$ as described above.
2. Run $R$ on $\langle B \rangle$.
3. if $R$ rejects, accept; if $R$ accepts, reject.
The Proof

Suppose TM $R$ decides $\text{EMPTY}_{\text{LBA}}$. Define TM $S$ that decides $A_{\text{TM}}$:

On input $\langle M, w \rangle$

1. Construct LBA, $B$, from $M$ and $w$ as described above.
2. Run $R$ on $\langle B \rangle$.
3. if $R$ rejects, accept; if $R$ accepts, reject.

If $R$ accepts $\langle B \rangle$

- $M$ has no accepting computation history on $w$
- $M$ does not accept $w$
- So $S$ rejects $\langle M, w \rangle$
The Proof

Suppose TM $R$ decides $\text{EMPTY}_{\text{LBA}}$.

Define TM $S$ that decides $A_{\text{TM}}$:

On input $\langle M, w \rangle$

1. Construct LBA, $B$, from $M$ and $w$ as described above.

2. Run $R$ on $\langle B \rangle$.

3. if $R$ rejects, accept; if $R$ accepts, reject.

If $R$ accepts $\langle B \rangle$

- $M$ has no accepting computation history on $w$
- $M$ does not accept $w$
- So $S$ rejects $\langle M, w \rangle$
The Proof

Suppose TM $R$ decides $\text{EMPTY}_{\text{LBA}}$.

Define TM $S$ that decides $A_{\text{TM}}$:

On input $\langle M, w \rangle$

1. Construct LBA, $B$, from $M$ and $w$ as described above.

2. Run $R$ on $\langle B \rangle$.

3. if $R$ rejects, accept; if $R$ accepts, reject.

If $R$ accepts $\langle B \rangle$

- $M$ has no accepting computation history on $w$
- $M$ does not accept $w$
- So $S$ rejects $\langle M, w \rangle$
The Proof

Suppose TM $R$ decides $\text{EMPTY}_{\text{LBA}}$.

Define TM $S$ that decides $A_{\text{TM}}$:

On input $\langle M, w \rangle$

1. Construct LBA, $B$, from $M$ and $w$ as described above.

2. Run $R$ on $\langle B \rangle$.

3. if $R$ rejects, accept; if $R$ accepts, reject.

If $R$ accepts $\langle B \rangle$

- $M$ has no accepting computation history on $w$
- $M$ does not accept $w$
- So $S$ rejects $\langle M, w \rangle$
The Proof (cont.)

If $R$ rejects $\langle B \rangle$

- the language of $B$ is non-empty
The Proof (cont.)

If $R$ rejects $\langle B \rangle$

- the language of $B$ is non-empty
- the only string $B$ can accept is an accepting computation of $M$ on $w$
The Proof (cont.)

If \( R \) rejects \( \langle B \rangle \)
- the language of \( B \) is non-empty
- the only string \( B \) can accept is an accepting computation of \( M \) on \( w \)
- thus \( M \) accepts \( w \)
The Proof (cont.)

If $R$ rejects $\langle B \rangle$
- the language of $B$ is non-empty
- the only string $B$ can accept is an accepting computation of $M$ on $w$
- thus $M$ accepts $w$
- So $S$ accepts $\langle M, w \rangle$. 
The Proof (cont.)

If $R$ rejects $\langle B \rangle$

- the language of $B$ is non-empty
- the only string $B$ can accept is an accepting computation of $M$ on $w$
- thus $M$ accepts $w$
- So $S$ accepts $\langle M, w \rangle$.

To conclude, $S$ decides $\mathcal{A}_{\mathsf{TM}}$, a contradiction. ♣
A CFG Question

SENTENCE

NOUN-PHRASE

ARTICLE

a

NOUN

boy

VERB

sees
Another Use for Computation Histories

We have already seen an algorithm to check whether a context-free grammar is empty.
Another Use for Computation Histories

We have already seen an algorithm to check whether a context-free grammar is empty. On input $\langle G \rangle$ where $G$ is a CFG:

1. Mark all terminal symbols in $G$.
2. Repeat until no new variables become marked:
Another Use for Computation Histories

We have already seen an algorithm to check whether a context-free grammar is empty. On input $\langle G \rangle$ where $G$ is a CFG:

1. Mark all terminal symbols in $G$.
2. Repeat until no new variables become marked:
3. Mark any $A$ where
   \[ A \rightarrow U_1 U_2 \ldots U_k \]
   and each $U_i$ has already been marked.
Another Use for Computation Histories

We have already seen an algorithm to check whether a context-free grammar is empty. On input $\langle G \rangle$ where $G$ is a CFG:

1. Mark all terminal symbols in $G$.
2. Repeat until no new variables become marked:
   3. Mark any $A$ where $A \rightarrow U_1 U_2 \ldots U_k$

   and each $U_i$ has already been marked.
4. If start symbol marked, accept, otherwise reject.
Another Use for Computation Histories

We have already seen an algorithm to check whether a context-free grammar is empty.

On input $\langle G \rangle$ where $G$ is a CFG:

1. Mark all terminal symbols in $G$.
2. Repeat until no new variables become marked:
   3. Mark any $A$ where
      \[ A \rightarrow U_1 U_2 \ldots U_k \]
      and each $U_i$ has already been marked.
3. If start symbol marked, accept, otherwise reject.
Another Use for Computation Histories

So the language $\text{EMPTY}_{\text{CFG}}$ is decidable.

**Question:** What about the complementary question: Does a CFG generate all strings?
Another Use for Computation Histories

So the language $\text{EMPTY}_{\text{CFG}}$ is decidable.

**Question:** What about the complementary question: Does a CFG generate all strings?

$$\text{All}_{\text{CFG}} = \{ \langle G \rangle | G \text{ is a CFL and } L(G) = \Sigma^* \}$$
Does a CFG Generate All Strings?

**Theorem:** \( \text{All}_{\text{CFG}} \) is undecidable.

Proof by reduction and contradiction:
- Assume \( \text{All}_{\text{CFG}} \) is decidable.
Does a CFG Generate All Strings?

**Theorem:** $\text{All}_{\text{CFG}}$ is undecidable.

Proof by reduction and contradiction:

- Assume $\text{All}_{\text{CFG}}$ is decidable.
- Show that $A_{\text{TM}}$ is then decidable.
Does a CFG Generate All Strings?

**Theorem:** \( \text{All}_{\text{CFG}} \) is undecidable.

Proof by reduction and contradiction:

- Assume \( \text{All}_{\text{CFG}} \) is decidable.
- show that \( A_{\text{TM}} \) is then decidable.
- for a TM, \( M \), and input, \( w \), construct a CFG, \( G \)
Does a CFG Generate All Strings?

**Theorem:** $\text{All}_{\text{CFG}}$ is undecidable.

Proof by reduction and contradiction:

- Assume $\text{All}_{\text{CFG}}$ is decidable.
- Show that $A_{\text{TM}}$ is then decidable.
- For a TM, $M$, and input, $w$, construct a CFG, $G$.
- $G$ generates all strings that are not accepting computation histories for $M$ on $w$. 
Does a CFG Generate All Strings?

**Theorem:** All$_{\text{CFG}}$ is undecidable.

Proof by reduction and contradiction:
- Assume All$_{\text{CFG}}$ is decidable.
- Show that $A_{\text{TM}}$ is then decidable.
- For a TM, $M$, and input, $w$, construct a CFG, $G$.
- $G$ generates all strings that are not accepting computation histories for $M$ on $w$.
- If $M$ does not accept $w$, $G$ generates all strings.
Does a CFG Generate All Strings?

**Theorem:** \( \text{All}_{\text{CFG}} \) is undecidable.

Proof by reduction and contradiction:

- Assume \( \text{All}_{\text{CFG}} \) is decidable.
- show that \( A_{\text{TM}} \) is then decidable.
- for a TM, \( M \), and input, \( w \), construct a CFG, \( G \)
- \( G \) generates all strings that are not accepting computation histories for \( M \) on \( w \)
- if \( M \) does not accept \( w \), \( G \) generates all strings
- if \( M \) does accept \( w \), \( G \) does not generate the accepting computation history.
Does a CFG Generate All Strings?

An accepting computation history appears as
#C_1#C_2# \ldots #C_\ell#, where

\[ C_1 \text{ is the starting configuration of } M \text{ on } w, \]
Does a CFG Generate All Strings?

An accepting computation history appears as \#C_1\#C_2\# \ldots \#C_\ell\#, where

- $C_1$ is the starting configuration of $M$ on $w$,
- $C_\ell$ is an accepting configuration of $M$, 

Does a CFG Generate All Strings?

An accepting computation history appears as

\[ \#C_1\#C_2\# \ldots \#C_\ell\# \],

where

- \( C_1 \) is the starting configuration of \( M \) on \( w \),
- \( C_\ell \) is an accepting configuration of \( M \),
- Each \( C_i \) yields \( C_{i+1} \) by transition function of \( M \).
Does a CFG Generate All Strings?

An accepting computation history appears as
\[#C_1#C_2# \ldots \#C_\ell#\], where

- $C_1$ is the starting configuration of $M$ on $w$,
- $C_\ell$ is an accepting configuration of $M$,
- Each $C_i$ yields $C_{i+1}$ by transition function of $M$.

A string is **not** an accepting computation history if it fails **one or more** of these conditions.
Does a CFG Generate All Strings?

Instead of the CFG, $G$, we construct a PDA, $D$. 
$D$ non-deterministically “guesses” which condition is violated.

Then verifies the guessed violation:
Does a CFG Generate All Strings?

Instead of the CFG, \( G \), we construct a PDA, \( D \). \( D \) non-deterministically “guesses” which condition is violated.

- then verifies the guessed violation:
  - Is there some \( C_i \) that is not a configuration of \( M \) (number of \( q \) symbols \( \neq 1 \))?
Does a CFG Generate All Strings?

Instead of the CFG, $G$, we construct a PDA, $D$. $D$ non-deterministically “guesses” which condition is violated.

- then verifies the guessed violation:
  - Is there some $C_i$ that is not a configuration of $M$ (number of $q$ symbols $\neq 1$)?
  - Is $C_1$ not the starting configuration of $M$ on $w$?
Does a CFG Generate All Strings?

Instead of the CFG, $G$, we construct a PDA, $D$. $D$ non-deterministically “guesses” which condition is violated.

- then verifies the guessed violation:
  - Is there some $C_i$ that is not a configuration of $M$ (number of $q$ symbols $\neq 1$)?
  - Is $C_1$ not the starting configuration of $M$ on $w$?
  - Is $C_\ell$ not an accepting configuration of $M$?
Does a CFG Generate All Strings?

Instead of the CFG, \( G \), we construct a PDA, \( D \). \( D \) non-deterministically “guesses” which condition is violated.

- then verifies the guessed violation:
  - Is there some \( C_i \) that is not a configuration of \( M \) (number of \( q \) symbols \( \neq 1 \))? 
  - Is \( C_1 \) not the starting configuration of \( M \) on \( w \)?
  - Is \( C_\ell \) not an accepting configuration of \( M \)?
  - Does \( C_i \) not yield \( C_{i+1} \) by the transition function of \( M \)?
Does a CFG Generate All Strings?

Instead of the CFG, $G$, we construct a PDA, $D$. $D$ non-deterministically “guesses” which condition is violated.

- then verifies the guessed violation:
  - Is there some $C_i$ that is not a configuration of $M$ (number of $q$ symbols $\neq 1$)?
  - Is $C_1$ not the starting configuration of $M$ on $w$?
  - Is $C_\ell$ not an accepting configuration of $M$?
  - Does $C_i$ not yield $C_{i+1}$ by the transition function of $M$?

- Like before, last condition is the tricky one to check.
Does a CFG Generate All Strings?

Instead of the CFG, $G$, we construct a PDA, $D$. $D$ non-deterministically “guesses” which condition is violated.

- then verifies the guessed violation:
  - Is there some $C_i$ that is not a configuration of $M$ (number of $q$ symbols $\neq 1$)?
  - Is $C_1$ not the starting configuration of $M$ on $w$?
  - Is $C_\ell$ not an accepting configuration of $M$?
  - Does $C_i$ not yield $C_{i+1}$ by the transition function of $M$?

- Like before, last condition is the tricky one to check.
Does a CFG Generate All Strings?

Does $C_i$ not yield $C_{i+1}$?

Idea:

Scan input. Nondeterministically decide "violating configuration" $C_i$ was reached.
Does a CFG Generate All Strings?

- Does $C_i$ not yield $C_{i+1}$?

Idea:

- Scan input. Nondeterministically decide "violating configuration" $C_i$ was reached.
- Push $C_i$ onto the stack till #.
Does a CFG Generate All Strings?

Does $C_i$ not yield $C_{i+1}$?

Idea:

- Scan input. Nondeterministically decide "violating configuration" $C_i$ was reached.
- Push $C_i$ onto the stack till #.
- scan $C_{i+1}$ and pop matching symbols of $C_i$
Does a CFG Generate All Strings?

- Does $C_i$ not yield $C_{i+1}$?

Idea:
- Scan input. Nondeterministically decide "violating configuration" $C_i$ was reached.
- Push $C_i$ onto the stack till #.
- scan $C_{i+1}$ and pop matching symbols of $C_i$
- check if $C_i$ and $C_{i+1}$ match everywhere, except . . .
Does a CFG Generate All Strings?

Does \( C_i \) not yield \( C_{i+1} \)?

Idea:

- Scan input. Nondeterministically decide "violating configuration" \( C_i \) was reached.
- Push \( C_i \) onto the stack till \#.
- Scan \( C_{i+1} \) and pop matching symbols of \( C_i \)
  - check if \( C_i \) and \( C_{i+1} \) match everywhere, except . . .
  - around the head position,
Does a CFG Generate All Strings?

Does $C_i$ not yield $C_{i+1}$?

Idea:

- Scan input. Nondeterministically decide "violating configuration" $C_i$ was reached.
- Push $C_i$ onto the stack till $\#$.
- scan $C_{i+1}$ and pop matching symbols of $C_i$
  - check if $C_i$ and $C_{i+1}$ match everywhere, except . . .
  - around the head position,
  - where difference dictated by transition function for $M$
Wait a Minute

**Problem:** When $D$ pops $C_i$ from stack, $C_i$ is in reverse order. Ignoring the local changes around head position, what we were trying to identify the language $x \# y$, with $x \neq y$. 
Wait a Minute

Problem: When $D$ pops $C_i$ from stack, $C_i$ is in reverse order. Ignoring the local changes around head position, what we were trying to identify the language $x \# y$, with $x \neq y$.

While this can be done in principle by a non deterministic PDA (see problem 2.26 in Sipser), there is a simpler way.
Wait a Minute

**Problem:** When $D$ pops $C_i$ from stack, $C_i$ is in reverse order. Ignoring the local changes around head position, what we were trying to identify the language $x \# y$, with $x \neq y$.

While this can be done in principle by a non deterministic PDA (see problem 2.26 in Sipser), there is a simpler way.

So far, we used a “straight” notion of accepting computation histories

$$\# \xrightarrow{C_1} \# \xrightarrow{C_2} \# \xrightarrow{C_3} \# \xrightarrow{C_4} \# \cdots \# \xrightarrow{C_\ell} \#$$
Does a CFG Generate All Strings?

So far, we used a “straight” notion of accepting computation histories

\[
\# \xrightarrow{C_1} \# \xrightarrow{C_2} \# \xrightarrow{C_3} \# \xrightarrow{C_4} \# \cdots \# \xrightarrow{C_\ell} \#
\]
Does a CFG Generate All Strings?

So far, we used a “straight” notion of accepting computation histories

But in this modern age, why not employ an alternative notion of accepting computation history, one that will make the life of our PDA much easier?
Does a CFG Generate All Strings?

So far, we used a “straight” notion of accepting computation histories

But in this modern age, why not employ an alternative notion of accepting computation history, one that will make the life of our PDA much easier? **Solution:** Write the accepting computation history so that every other configuration is in reverse order.

This takes care of difficulty in the proof.
Wrapping Things Up

Given \((M, w)\), we construct (algorithmically) a PDA, \(D\), which rejects the string \(x\) if and only if \(x\) equals an accepting computation history of \(M\) on \(w\), written in the "alternating format".
Wrapping Things Up

Given \( \langle M, w \rangle \), we construct (algorithmically) a PDA, \( D \), which rejects the string \( x \) if and only if \( x \) equals an accepting computation history of \( M \) on \( w \), written in the "alternating format".

Therefore \( L(D) \) is either \( \Sigma^* \) or \( \Sigma^* \setminus \{x\} \).
Wrapping Things Up

Given $\langle M, w \rangle$, we construct (algorithmically) a PDA, $D$, which rejects the string $x$ if and only if $x$ equals an accepting computation history of $M$ on $w$, written in the "alternating format".

Therefore $L(D)$ is either $\Sigma^*$ or $\Sigma^* \setminus \{x\}$.

We construct $D$ in order to feed it to the claimed decider, $R$, of $\text{All}_{\text{CFG}}$ (which we assume to exist).
Wrapping Things Up

Given $\langle M, w \rangle$, we construct (algorithmically) a PDA, $D$, which rejects the string $x$ if and only if $x$ equals an accepting computation history of $M$ on $w$, written in the "alternating format".

Therefore $L(D)$ is either $\Sigma^*$ or $\Sigma^* \setminus \{x\}$.

We construct $D$ in order to feed it to the claimed decider, $R$, of $\text{All}_{\text{CFG}}$ (which we assume to exist).

Once this decider returns its answer, we invert this answer to decide whether $M$ accepts $w$. 
Wrapping Things Up

Given $\langle M, w \rangle$, we construct (algorithmically) a PDA, $D$, which rejects the string $x$ if and only if $x$ equals an accepting computation history of $M$ on $w$, written in the "alternating format".

Therefore $L(D)$ is either $\Sigma^*$ or $\Sigma^* \setminus \{x\}$.

We construct $D$ in order to feed it to the claimed decider, $R$, of All$_{CFG}$ (which we assume to exist).

Once this decider returns its answer, we invert this answer to decide whether $M$ accepts $w$.

But then we can use $R$ to decide $A_{TM}$, a contradiction.

♣
Additional Undecidable Problems

We have already established that $A_{TM}$ is undecidable.

Here is a related problem.

$H_{TM} = \{ \langle M, w \rangle | M \text{ is a TM and } M \text{ halts on input } w \}$

Clarification: How does $H_{TM}$ differ from $A_{TM}$?
Undecidable Problems

\[ H_{TM} = \{ \langle M, w \rangle | M \text{ is a TM and } M \text{ halts on input } w \} \]

**Theorem:** \( H_{TM} \) is undecidable.

**Proof idea:**
- By contradiction.
Undecidable Problems

\[ H_{\text{TM}} = \{ \langle M, w \rangle | M \text{ is a TM and } M \text{ halts on input } w \} \]

**Theorem:** \( H_{\text{TM}} \) is undecidable.

**Proof idea:**
- By contradiction.
- Assume \( H_{\text{TM}} \) is decidable.
Undecidable Problems

\[ H_{TM} = \{ \langle M, w \rangle | M \text{ is a TM and } M \text{ halts on input } w \} \]

**Theorem:** \( H_{TM} \) is undecidable.

Proof idea:
- By contradiction.
- Assume \( H_{TM} \) is decidable.
- Let \( R \) be a TM that decides \( H_{TM} \).
Undecidable Problems

\[ H_{TM} = \{ \langle M, w \rangle | M \text{ is a TM and } M \text{ halts on input } w \} \]

**Theorem:** \( H_{TM} \) is undecidable.

**Proof idea:**
- By contradiction.
- Assume \( H_{TM} \) is decidable.
- Let \( R \) be a TM that decides \( H_{TM} \).
- Use \( R \) to construct \( S \), a TM that decides \( A_{TM} \).
Undecidable Problems

\[ H_{TM} = \{ \langle M, w \rangle | M \text{ is a TM and } M \text{ halts on input } w \} \]

**Theorem:** \( H_{TM} \) is undecidable.

**Proof idea:**

- By contradiction.
- Assume \( H_{TM} \) is decidable.
- Let \( R \) be a TM that decides \( H_{TM} \).
- Use \( R \) to construct \( S \), a TM that decides \( A_{TM} \).
- So \( A_{TM} \) is reduced to \( H_{TM} \).
Undecidable Problems

\[ H_{\text{TM}} = \{\langle M, w \rangle | M \text{ is a TM and } M \text{ halts on input } w \} \]

**Theorem:** \( H_{\text{TM}} \) is undecidable.

**Proof idea:**

- By contradiction.
- Assume \( H_{\text{TM}} \) is decidable.
- Let \( R \) be a TM that decides \( H_{\text{TM}} \).
- Use \( R \) to construct \( S \), a TM that decides \( A_{\text{TM}} \).
- So \( A_{\text{TM}} \) is reduced to \( H_{\text{TM}} \).
- Since \( A_{\text{TM}} \) is undecidable, so is \( H_{\text{TM}} \).
Undecidable Problems

**Theorem:** \( H_{TM} \) is undecidable.

**Proof:** Assume, by way of contradiction, that TM \( R \) decides \( H_{TM} \). Define a new TM, \( S \), as follows:

- On input \( \langle M, w \rangle \),
Undecidable Problems

**Theorem:** $H_{TM}$ is undecidable.

**Proof:** Assume, by way of contradiction, that TM $R$ decides $H_{TM}$. Define a new TM, $S$, as follows:

- On input $\langle M, w \rangle$,
- run $R$ on $\langle M, w \rangle$.
Undecidable Problems

**Theorem:** $H_{TM}$ is undecidable.

**Proof:** Assume, by way of contradiction, that TM $R$ decides $H_{TM}$. Define a new TM, $S$, as follows:

- On input $\langle M, w \rangle$,
- run $R$ on $\langle M, w \rangle$.
- If $R$ rejects, reject.
Undecidable Problems

**Theorem:** $H_{TM}$ is undecidable.

**Proof:** Assume, by way of contradiction, that TM $R$ decides $H_{TM}$. Define a new TM, $S$, as follows:

- On input $\langle M, w \rangle$,
- run $R$ on $\langle M, w \rangle$.
- If $R$ rejects, reject.
- If $R$ accepts (meaning $M$ halts on $w$), simulate $M$ on $w$ until it halts (namely run $U$ on $\langle M, w \rangle$).
Undecidable Problems

**Theorem:** $H_{TM}$ is undecidable.

**Proof:** Assume, by way of contradiction, that TM $R$ decides $H_{TM}$. Define a new TM, $S$, as follows:

- On input $\langle M, w \rangle$,
- run $R$ on $\langle M, w \rangle$.
- If $R$ rejects, reject.
- If $R$ accepts (meaning $M$ halts on $w$), simulate $M$ on $w$ until it halts (namely run $U$ on $\langle M, w \rangle$).
- If $M$ accepted, accept; otherwise reject.

♣ Slides modified by Benny Chor, based on original slides by Maurice Herlihy, Brown University. – p.47
Undecidable Problems (2)

Does a TM accept any string at all?

\[ \text{EMPTY}_{TM} = \{ \langle M \rangle | M \text{ is a TM and } L(M) = \emptyset \} \]
Undecidable Problems (2)

Does a TM accept any string at all?

$$\text{EMPTY}_{TM} = \{ \langle M \rangle | M \text{ is a TM and } L(M) = \emptyset \}$$

**Theorem:** $\text{EMPTY}_{TM}$ is undecidable.
Undecidable Problems (2)

Does a TM accept any string at all?

\[ \text{EMPTY}_{TM} = \{ \langle M \rangle | M \text{ is a TM and } L(M) = \emptyset \} \]

**Theorem:** \( \text{EMPTY}_{TM} \) is undecidable.

**Proof structure:**
Undecidable Problems (2)

Does a TM accept any string at all?

$$\text{EMPTY}_{TM} = \{\langle M \rangle | M \text{ is a TM and } L(M) = \emptyset \}$$

**Theorem:** $\text{EMPTY}_{TM}$ is undecidable.

**Proof structure:**

- By contradiction.
- Assume $\text{EMPTY}_{TM}$ is decidable.
- Let $R$ be a TM that decides $\text{EMPTY}_{TM}$.
- Use $R$ to construct $S$, a TM that decides $A_{TM}$. 
Undecidable Problems (2)

$$\text{EMPTY}_{\text{TM}} = \{ \langle M \rangle | M \text{ is a TM and } L(M) = \emptyset \}$$
Undecidable Problems (2)

\[ \text{EMPTY}_{TM} = \{ \langle M \rangle \mid M \text{ is a TM and } L(M) = \emptyset \} \]

**First attempt:** When \( S \) receives input \( \langle M, w \rangle \), it calls \( R \) with input \( \langle M \rangle \).

- If \( R \) accepts, then reject, because \( M \) does not accept any string, let alone \( w \).
- But what if \( R \) rejects?
Undecidable Problems (2)

\[ \text{EMPTY}_{\text{TM}} = \{ \langle M \rangle | M \text{ is a TM and } L(M) = \emptyset \} \]

First attempt: When \( S \) receives input \( \langle M, w \rangle \), it calls \( R \) with input \( \langle M \rangle \).

- If \( R \) accepts, then reject, because \( M \) does not accept any string, let alone \( w \).
- But what if \( R \) rejects?

Second attempt: Let’s modify \( M \).
Undecidable Problems (2)

$$\text{EMPTY}_{\text{TM}} = \{ \langle M \rangle | M \text{ is a TM and } L(M) = \emptyset \}$$

Define $M_1$: on input $x$,

1. if $x \neq w$, reject.
2. if $x = w$, run $M$ on $w$ and accept if $M$ does.
Undecidable Problems (2)

\[ \text{EMPTY}_{\text{TM}} = \{ \langle M \rangle | M \text{ is a TM and } L(M) = \emptyset \} \]

Define \( M_1 \): on input \( x \),

1. if \( x \neq w \), reject.
2. if \( x = w \), run \( M \) on \( w \) and accept if \( M \) does.

\( M_1 \) either

- accepts just \( w \), or
- accepts nothing.

Slides modified by Benny Chor, based on original slides by Maurice Herlihy, Brown University.
Undecidable Problems (2)

Machine $M_1$: on input $x$,

1. if $x \neq w$, reject.
2. if $x = w$, run $M$ on $w$ and accept if $M$ does.
Undecidable Problems (2)

Machine $M_1$: on input $x$,

1. if $x \neq w$, reject.

2. if $x = w$, run $M$ on $w$ and accept if $M$ does.

**Question**: Can a TM construct $M_1$ from $M$?
Undecidable Problems (2)

Machine $M_1$: on input $x$,
1. if $x \neq w$, reject.
2. if $x = w$, run $M$ on $w$ and accept if $M$ does.

Question: Can a TM construct $M_1$ from $M$?

Answer: Yes, because we need only hardwire $w$, and add a few extra states to perform the “$x = w?$” test.
Undecidable Problems (2)

\[ \text{EMPTY}_{\text{TM}} = \{ \langle M \rangle | M \text{ is a TM and } L(M) = \emptyset \} \]

**Theorem:** \( \text{EMPTY}_{\text{TM}} \) is undecidable.
Undecidable Problems (2)

\[ \text{EMPTY}_{\text{TM}} = \{ \langle M \rangle | M \text{ is a TM and } L(M) = \emptyset \} \]

**Theorem:** \(\text{EMPTY}_{\text{TM}}\) is undecidable.

Define \(S\) as follows:

On input \(\langle M, w \rangle\), where \(M\) is a TM and \(w\) a string,
Undecidable Problems (2)

\[
\text{EMPTY}_{TM} = \{ \langle M \rangle | M \text{ is a TM and } L(M) = \emptyset \}
\]

**Theorem:** \(\text{EMPTY}_{TM}\) is undecidable.

Define \(S\) as follows:

On input \(\langle M, w \rangle\), where \(M\) is a TM and \(w\) a string,

- Construct \(M_1\) from \(M\) and \(w\).
- Run \(R\) on input \(\langle M_1 \rangle\),
- if \(R\) accepts, *reject*; if \(R\) rejects, *accept*.
Undecidable Problems (3)

Does a TM accept a regular language?

\[
\text{REG}_{\text{TM}} = \{ \langle M \rangle | M \text{ is a TM and } L(M) \text{ is regular} \}
\]
Undecidable Problems (3)

Does a TM accept a regular language?

\[ \text{REG}_{\text{TM}} = \{ \langle M \rangle | M \text{ is a TM and } L(M) \text{ is regular} \} \]

**Theorem:** \( \text{REG}_{\text{TM}} \) is undecidable.
Undecidable Problems (3)

Does a TM accept a regular language?

\[
\text{REG}_{\text{TM}} = \{ \langle M \rangle | M \text{ is a TM and } L(M) \text{ is regular} \}
\]

**Theorem:** \(\text{REG}_{\text{TM}}\) is undecidable.

**Skeleton of Proof:**
- By contradiction.
- Assume \(\text{REG}_{\text{TM}}\) is decidable.
- Let \(R\) be a TM that decides \(\text{REG}_{\text{TM}}\).
- Use \(R\) to construct \(S\), a TM that decides \(A_{\text{TM}}\).
Undecidable Problems (3)

Does a TM accept a regular language?

\[ \text{REG}_{\text{TM}} = \{ \langle M \rangle | M \text{ is a TM and } L(M) \text{ is regular} \} \]

**Theorem:** \( \text{REG}_{\text{TM}} \) is undecidable.

**Skeleton of Proof:**

- By contradiction.
- Assume \( \text{REG}_{\text{TM}} \) is decidable.
- Let \( R \) be a TM that decides \( \text{REG}_{\text{TM}} \).
- Use \( R \) to construct \( S \), a TM that decides \( A_{\text{TM}} \).

But how?
Undecidable Problems (3)

\[ \text{REG}_{\text{TM}} = \{ \langle M \rangle | M \text{ is a TM and } L(M) \text{ is regular} \} \]

Modify the TM $M$ so that the resulting TM accepts a regular language if and only if $M$ accepts $w$. 
Undecidable Problems (3)

\[ \text{REG}_{\text{TM}} = \{ \langle M \rangle | M \text{ is a TM and } L(M) \text{ is regular} \} \]

Modify \( M \) so that the resulting TM accepts a regular language if and only if \( M \) accepts \( w \).

Design \( M_2 \) so that
- if \( M \) does not accept \( w \), then \( M_2 \) accepts \( \{0^n1^n | n \geq 0\} \) (non-regular)
- if \( M \) accepts \( w \), then \( M_2 \) accepts \( \Sigma^* \) (regular).
Undecidable Problems (3)

>From $M$ and $w$, define $M_2$:
Undecidable Problems (3)

>From $M$ and $w$, define $M_2$:

On input $x$,

1. If $x$ has the form $0^n1^n$, accept it.
2. Otherwise, run $M$ on input $w$ and accept $x$ if $M$ accepts $w$. 
Undecidable Problems (3)

>From $M$ and $w$, define $M_2$:

On input $x$,

1. If $x$ has the form $0^n1^n$, accept it.
2. Otherwise, run $M$ on input $w$ and accept $x$ if $M$ accepts $w$.

Claim:

- If $M$ does not accept $w$, then $M_2$ accepts $\{0^n1^n | n \geq 0\}$.
- If $M$ accepts $w$, then $M_2$ accepts $\Sigma^*$. 
Undecidable Problems (3)

\[
\text{REG}_{\text{TM}} = \{ \langle M \rangle \mid M \text{ is a TM and } L(M) \text{ is regular} \}
\]

**Theorem:** \( \text{REG}_{\text{TM}} \) is undecidable.
Undecidable Problems (3)

\[ \text{REG}_{\text{TM}} = \{ \langle M \rangle | M \text{ is a TM and } L(M) \text{ is regular} \} \]

**Theorem:** \( \text{REG}_{\text{TM}} \) is undecidable.

Define \( S \):

On input \( \langle M, w \rangle \),

1. Construct \( M_2 \) from \( M \) and \( w \).
2. Run \( R \) on input \( \langle M_2 \rangle \).
3. If \( R \) accepts, accept; if \( R \) rejects, reject.
Undecidable Problems (4)

Are two TMs equivalent?

\[ \text{EQ}_{\text{TM}} = \{ \langle M_1, M_2 \rangle \mid M_1, M_2 \text{ are TMs and } L(M_1) = L(M_2) \} \]

**Theorem:** \( \text{EQ}_{\text{TM}} \) is undecidable.
Undecidable Problems (4)

Are two TMs equivalent?

\[ \text{EQ}_\text{TM} = \{ \langle M_1, M_2 \rangle \mid M_1, M_2 \text{ are TMs and} \]
\[ L(M_1) = L(M_2) \} \]

**Theorem:** \( \text{EQ}_\text{TM} \) is undecidable.

We are getting tired of reducing \( A_{\text{TM}} \) to everything.
Undecidable Problems (4)

Are two TMs equivalent?

\[ \text{EQ}_{\text{TM}} = \{ \langle M_1, M_2 \rangle \mid M_1, M_2 \text{ are TMs and } L(M_1) = L(M_2) \} \]

**Theorem:** \( \text{EQ}_{\text{TM}} \) is undecidable.

We are getting tired of reducing \( A_{\text{TM}} \) to everything.

Let’s try instead a reduction from \( \text{EMPTY}_{\text{TM}} \) to \( \text{EQ}_{\text{TM}} \).
Undecidable Problems (4)

\[ \text{EQ}_{TM} = \{ \langle M_1, M_2 \rangle \mid M_1, M_2 \text{ are TMs and } L(M_1) = L(M_2) \} \]

**Theorem:** \( \text{EQ}_{TM} \) is undecidable.

**Idea:**

\( \text{EMPTY}_{TM} \) is the problem of testing whether a TM language is empty.
Undecidable Problems (4)

\[ \text{EQ}_{\text{TM}} = \{ \langle M_1, M_2 \rangle \mid M_1, M_2 \text{ are TMs and } L(M_1) = L(M_2) \} \]

**Theorem:** \( \text{EQ}_{\text{TM}} \) is undecidable.

**Idea:**
- \( \text{EMPTY}_{\text{TM}} \) is the problem of testing whether a TM language is empty.
- \( \text{EQ}_{\text{TM}} \) is the problem of testing whether two TM languages are the same.
Undecidable Problems (4)

\[ \text{EQ}_{\text{TM}} = \{ \langle M_1, M_2 \rangle \mid M_1, M_2 \text{ are TMs and} \]
\[ L(M_1) = L(M_2) \} \]

**Theorem:** \( \text{EQ}_{\text{TM}} \) is undecidable.

**Idea:**

- \( \text{EMPTY}_{\text{TM}} \) is the problem of testing whether a TM language is empty.
- \( \text{EQ}_{\text{TM}} \) is the problem of testing whether two TM languages are the same.
- If one of these two TM languages happens to be empty, then we are back to \( \text{EMPTY}_{\text{TM}} \).
Undecidable Problems (4)

\[ \text{EQ}_{\text{TM}} = \{ \langle M_1, M_2 \rangle \mid M_1, M_2 \text{ are TMs and } L(M_1) = L(M_2) \} \]

**Theorem:** \( \text{EQ}_{\text{TM}} \) is undecidable.

**Idea:**

- \( \text{EMPTY}_{\text{TM}} \) is the problem of testing whether a TM language is empty.
- \( \text{EQ}_{\text{TM}} \) is the problem of testing whether two TM languages are the same.
- If one of these two TM languages happens to be empty, then we are back to \( \text{EMPTY}_{\text{TM}} \).
- So \( \text{EMPTY}_{\text{TM}} \) is a special case of \( \text{EQ}_{\text{TM}} \).

The rest is easy.
Undecidable Problems (4)

\[ EQ_{TM} = \{ \langle M_1, M_2 \rangle \mid M_1, M_2 \text{ are TMs and} \]
\[ L(M_1) = L(M_2) \}\]

**Theorem:** \( EQ_{TM} \) is undecidable.

Let \( M_{NO} \) be the TM: On input \( x \), reject.
Let \( R \) decide \( EQ_{TM} \).
Undecidable Problems (4)

\[ \text{EQ}_{\text{TM}} = \{ \langle M_1, M_2 \rangle \mid M_1, M_2 \text{ are TMs and} \]
\[ \quad L(M_1) = L(M_2) \} \]

**Theorem:** \( \text{EQ}_{\text{TM}} \) is undecidable.

Let \( M_{\text{NO}} \) be the TM: On input \( x \), reject.

Let \( R \) decide \( \text{EQ}_{\text{TM}} \).

Let \( S \) be: On input \( \langle M \rangle \):

1. Run \( R \) on input \( \langle M, M_{\text{NO}} \rangle \).
2. If \( R \) accepts, accept; if \( R \) rejects, reject.
Undecidable Problems (4)

\[ \text{EQ}_{\text{TM}} = \{ \langle M_1, M_2 \rangle \mid M_1, M_2 \text{ are TMs and} \ L(M_1) = L(M_2) \} \]

**Theorem:** \( \text{EQ}_{\text{TM}} \) is undecidable.

Let \( M_{\text{NO}} \) be the TM: On input \( x \), reject.

Let \( R \) decide \( \text{EQ}_{\text{TM}} \).

Let \( S \) be: On input \( \langle M \rangle \):

1. Run \( R \) on input \( \langle M, M_{\text{NO}} \rangle \).
2. If \( R \) accepts, accept; if \( R \) rejects, reject.

If \( R \) decides \( \text{EQ}_{\text{TM}} \), then \( S \) decides \( \text{EMPTY}_{\text{TM}} \).
Bucket of Undecidable Problems

Same techniques prove undecidability of

- Does a TM accept a **decidable** language?
Bucket of Undecidable Problems

Same techniques prove undecidability of

- Does a TM accept a **decidable** language?
- Does a TM accept an **enumerable** language?
Bucket of Undecidable Problems

Same techniques prove undecidability of

- Does a TM accept a **decidable** language?
- Does a TM accept an **enumerable** language?
- Does a TM accept a **context-free** language?
Bucket of Undecidable Problems

Same techniques prove undecidability of

- Does a TM accept a **decidable** language?
- Does a TM accept an **enumerable** language?
- Does a TM accept a **context-free** language?
- Does a TM accept a **finite** language?
Bucket of Undecidable Problems

Same techniques prove undecidability of

- Does a TM accept a **decidable** language?
- Does a TM accept an **enumerable** language?
- Does a TM accept a **context-free** language?
- Does a TM accept a **finite** language?
- Does a TM halt on all inputs?
Bucket of Undecidable Problems

Same techniques prove undecidability of

- Does a TM accept a **decidable** language?
- Does a TM accept an **enumerable** language?
- Does a TM accept a **context-free** language?
- Does a TM accept a **finite** language?
- Does a TM halt on all inputs?
- Is there an input string that causes a TM to traverse all its states?
Reducibility

So far, we have seen many examples of reductions from one language to another, but the notion was neither defined nor treated formally.

Reductions play an important role in
- decidability theory
- complexity theory (to come)

Time to get formal.
Mapping Reductions

**Definition:** Let $A$ and $B$ be two languages. We say that there is a mapping reduction from $A$ to $B$, and denote

$$A \leq_m B$$
Mapping Reductions

**Definition:** Let $A$ and $B$ be two languages. We say that there is a mapping reduction from $A$ to $B$, and denote

$$A \leq_m B$$

if there is a computable function

$$f : \Sigma^* \rightarrow \Sigma^*$$

such that, for every $w$,
Mapping Reductions

**Definition:** Let $A$ and $B$ be two languages. We say that there is a mapping reduction from $A$ to $B$, and denote

$$A \leq_m B$$

if there is a computable function

$$f : \Sigma^* \longrightarrow \Sigma^*$$

such that, for every $w$,

$$w \in A \iff f(w) \in B.$$ 

The function $f$ is called the reduction from $A$ to $B$. 
Mapping Reductions
A mapping reduction converts questions about membership in $A$ to membership in $B$.
Mapping Reductions

**Theorem:** If $A \leq_m B$ and $B$ is decidable, then $A$ is decidable.
Mapping Reductions

**Theorem:** If \( A \leq_m B \) and \( B \) is decidable, then \( A \) is decidable.

**Proof:** Let

- \( M \) be the decider for \( B \), and
- \( f \) the reduction from \( A \) to \( B \).
Mapping Reductions

**Theorem:** If $A \leq_m B$ and $B$ is decidable, then $A$ is decidable.

**Proof:** Let

- $M$ be the decider for $B$, and
- $f$ the reduction from $A$ to $B$.

Define $N$: On input $w$

1. compute $f(w)$
2. run $M$ on input $f(w)$ and output whatever $M$ outputs.
Mapping Reductions

Corollary: If $A \leq_m B$ and $A$ is undecidable, then $B$ is undecidable.
Mapping Reductions

Corollary: If $A \leq_m B$ and $A$ is undecidable, then $B$ is undecidable.

In fact, this has been our principal tool for proving undecidability of languages other than $A_{TM}$. 
Example: Halting

Recall that

$$A_{TM} = \{ \langle M, w \rangle | TM \ M \text{ accepts input } w \}$$

$$H_{TM} = \{ \langle M, w \rangle | TM \ M \text{ halts on input } w \}$$
Example: Halting

Recall that

\[ A_{TM} = \{ \langle M, w \rangle | \text{TM } M \text{ accepts input } w \} \]
\[ H_{TM} = \{ \langle M, w \rangle | \text{TM } M \text{ halts on input } w \} \]

Earlier we proved that

- \( H_{TM} \) undecidable
- by (de facto) reduction from \( A_{TM} \).

Let’s reformulate this.
Example: Halting

Define a computable function, $f$:

- input of form $\langle M, w \rangle$
Example: Halting

Define a computable function, \( f \):
- input of form \( \langle M, w \rangle \)
- output of form \( \langle M', w' \rangle \)
Example: Halting

Define a computable function, \( f \):

- input of form \( \langle M, w \rangle \)
- output of form \( \langle M', w' \rangle \)
- where \( \langle M, w \rangle \in A_{TM} \iff \langle M', w' \rangle \in H_{TM} \).
Example: Halting

The following machine computes this function $f$.

$F = \text{on input } \langle M, w \rangle$: 

- Construct the following machine $M'$. 

$M'$: on input $x$
Example: Halting

The following machine computes this function \( f \).
\[ F = \text{on input } \langle M, w \rangle: \]

Construct the following machine \( M' \).
\[ M': \text{on input } x \]

run \( M \) on \( x \)
Example: Halting

The following machine computes this function $f$. $F$ = on input $\langle M, w \rangle$:

- Construct the following machine $M'$. $M'$: on input $x$
  - run $M$ on $x$
  - If $M$ accepts, accept.
Example: Halting

The following machine computes this function $f$. $F = \text{on input } \langle M, w \rangle$:

- Construct the following machine $M'$. 
  $M'$: on input $x$
  - run $M$ on $x$
  - If $M$ accepts, accept.
  - if $M$ rejects, enter a loop.
Example: Halting

The following machine computes this function $f$.

$F = \text{on input } \langle M, w \rangle$:

- Construct the following machine $M'$.
  
  $M'$: on input $x$
  
  - run $M$ on $x$
  
  - If $M$ accepts, accept.
  
  - if $M$ rejects, enter a loop.

- output $\langle M', w \rangle$
Enumerability

**Theorem:** If \( A \leq_m B \) and \( B \) is enumerable, then \( A \) is enumerable.

Proof is same as before, using accepters instead of deciders.
Enumerability

**Corollary:** If $A \leq_m B$ and $A$ is not enumerable, then $B$ is not enumerable.
TM Equality

**Theorem:** Both $EQ_{TM}$ and its complement, $\overline{EQ_{TM}}$, are not enumerable. Stated differently, $EQ_{TM}$ is neither enumerable nor co-enumerable.
TM Equality

**Theorem:** Both $\text{EQ}_{\text{TM}}$ and its complement, $\overline{\text{EQ}_{\text{TM}}}$, are not enumerable. Stated differently, $\text{EQ}_{\text{TM}}$ is neither enumerable nor co-enumerable.

We show that $A_{\text{TM}}$ is reducible to $\text{EQ}_{\text{TM}}$. The same function is also a mapping reduction from $A_{\text{TM}}$ to $\overline{\text{EQ}_{\text{TM}}}$, and thus $\overline{\text{EQ}_{\text{TM}}}$ is not enumerable.
TM Equality

**Theorem:** Both $\text{EQ}_{\text{TM}}$ and its complement, $\overline{\text{EQ}_{\text{TM}}}$, are not enumerable. Stated differently, $\text{EQ}_{\text{TM}}$ is neither enumerable nor co-enumerable.

- We show that $A_{\text{TM}}$ is reducible to $\text{EQ}_{\text{TM}}$. The same function is also a mapping reduction from $\overline{A_{\text{TM}}}$ to $\overline{\text{EQ}_{\text{TM}}}$, and thus $\overline{\text{EQ}_{\text{TM}}}$ is not enumerable.

- We then show that $A_{\text{TM}}$ is reducible to $\overline{\text{EQ}_{\text{TM}}}$. The new function is also a mapping reduction from $\overline{A_{\text{TM}}}$ to $\overline{\text{EQ}_{\text{TM}}}$, and thus $\text{EQ}_{\text{TM}}$ is not enumerable.
TM Equality

Claim: $A_{TM}$ is reducible to $\overline{EQ}_{TM}$.

$f : A_{TM} \rightarrow \overline{EQ}_{TM}$ works as follows:

$F$: On input $\langle M, w \rangle$

Construct machine $M_1$: on any input, reject.
TM Equality

Claim: $A_{TM}$ is reducible to $\overline{EQ_{TM}}$.

$f : A_{TM} \rightarrow \overline{EQ_{TM}}$ works as follows:

$F$: On input $\langle M, w \rangle$

- Construct machine $M_1$: on any input, reject.
- Construct machine $M_2$: on input $x$, run $M$ on $w$. If it accepts, accept.
TM Equality

**Claim:** $A_{TM}$ is reducible to $\overline{EQ_{TM}}$.

$f : A_{TM} \rightarrow \overline{EQ_{TM}}$ works as follows:

$F$: On input $\langle M, w \rangle$

- Construct machine $M_1$: on any input, reject.
- Construct machine $M_2$: on input $x$, run $M$ on $w$. If it accepts, accept.
- Output $\langle M_1, M_2 \rangle$. 
TM Equality

$F$: On input $\langle M, w \rangle$

- Construct machine $M_1$: on any input, reject.
- Construct machine $M_2$: on any input $x$, run $M$ on $w$.
  If it accepts, accept $x$.
- Output $\langle M_1, M_2 \rangle$.

Note

- $M_1$ accepts nothing
TM Equality

\[ F: \text{On input } \langle M, w \rangle \]

- Construct machine \( M_1 \): on any input, reject.
- Construct machine \( M_2 \): on any input \( x \), run \( M \) on \( w \).
  - If it accepts, accept \( x \).
- Output \( \langle M_1, M_2 \rangle \).

Note

- \( M_1 \) accepts nothing
- if \( M \) accepts \( w \) then \( M_2 \) accepts everything, and otherwise nothing.
TM Equality

$F$: On input $\langle M, w \rangle$

- Construct machine $M_1$: on any input, reject.
- Construct machine $M_2$: on any input $x$, run $M$ on $w$.
  
  If it accepts, accept $x$.

- Output $\langle M_1, M_2 \rangle$.

Note

- $M_1$ accepts nothing
- if $M$ accepts $w$ then $M_2$ accepts everything, and otherwise nothing.

so $\langle M, w \rangle \in A_{TM} \iff \langle M_1, M_2 \rangle \in EQ_{TM}$
TM Equality

**Claim:** $A_{\text{TM}}$ is reducible to $E_{\text{QTM}}$.

$f : A_{\text{TM}} \rightarrow E_{\text{QTM}}$ works as follows:

$F$: On input $\langle M, w \rangle$

- Construct machine $M_1$: on any input, *accept*. 
**TM Equality**

**Claim:** $A_{TM}$ is reducible to $EQ_{TM}$.

$f : A_{TM} \rightarrow EQ_{TM}$ works as follows:

**F:** On input $\langle M, w \rangle$

- Construct machine $M_1$: on any input, *accept*.
- Construct machine $M_2$: on any input $x$, run $M$ on $w$.

If it accepts, *accept*. 
TM Equality

Claim: $A_{\text{TM}}$ is reducible to $\text{EQ}_{\text{TM}}$.

$f : A_{\text{TM}} \longrightarrow \text{EQ}_{\text{TM}}$ works as follows:

$F$: On input $\langle M, w \rangle$

- Construct machine $M_1$: on any input, accept.
- Construct machine $M_2$: on any input $x$, run $M$ on $w$.
  If it accepts, accept.
- Output $\langle M_1, M_2 \rangle$. 
TM Equality

$F$: On input $\langle M, w \rangle$  
- Construct machine $M_1$: on any input, accept.
- Construct machine $M_2$: on any input $x$, run $M$ on $w$. If it accepts, accept.
- Output $\langle M_1, M_2 \rangle$.

Note
- $M_1$ accepts everything
TM Equality

\[ F: \text{On input } \langle M, w \rangle \]

- Construct machine \( M_1 \): on any input, accept.
- Construct machine \( M_2 \): on any input \( x \), run \( M \) on \( w \).
  
  If it accepts, accept.

- Output \( \langle M_1, M_2 \rangle \).

Note

- \( M_1 \) accepts everything
- if \( M \) accepts \( w \), then \( M_2 \) accepts everything, and otherwise nothing.
TM Equality

\( F \): On input \( \langle M, w \rangle \)

- Construct machine \( M_1 \): on any input, accept.
- Construct machine \( M_2 \): on any input \( x \), run \( M \) on \( w \).
  
  If it accepts, accept.

- Output \( \langle M_1, M_2 \rangle \).

Note

- \( M_1 \) accepts everything
- if \( M \) accepts \( w \), then \( M_2 \) accepts everything, and otherwise nothing.

\( \langle M, w \rangle \in A_{TM} \iff \langle M_1, M_2 \rangle \in EQ_{TM} \).
Recursive Inseparability

Two disjoint languages $L_1$ and $L_2$ are recursively inseparable if there is no decidable language $D$ such that

$$L_1 \cap D = \emptyset,$$

and
Recursive Inseparability

Two disjoint languages $L_1$ and $L_2$ are recursively inseparable if there is no decidable language $D$ such that

- $L_1 \cap D = \emptyset$, and
- $L_2 \subset D$. 
Recursive Inseparability

Two disjoint languages $L_1$ and $L_2$ are recursively inseparable if there is no decidable language $D$ such that

- $L_1 \cap D = \emptyset$, and
- $L_2 \subset D$.

Example of recursively separable languages:
Recursive Inseparability

$A_{TM}$ and $\overline{A}_{TM}$ are a trivial example.
Recursive Inseparability

$A_{TM}$ and $\overline{A_{TM}}$ are a trivial example.

Why?
Recursive Inseparability

$A_{TM}$ and $\overline{A_{TM}}$ are a trivial example.

Why?

Are there non-trivial examples?
Recursive Inseparability

Define

\[ A_{\text{yes}} = \{ \langle M \rangle | \text{M is a TM that accepts } \langle M \rangle \} \]

and

\[ A_{\text{no}} = \{ \langle M \rangle | \text{M is a TM that halts and rejects } \langle M \rangle \} \]

**Theorem:** \( A_{\text{yes}} \) and \( A_{\text{no}} \) are recursively inseparable.
Proof by Contradiction

Let $D$ be a decidable language that separates them.
Proof by Contradiction

- Let $D$ be a decidable language that separates them.
- Assume $A_{\text{no}} \subset D$ and $D \cap A_{\text{yes}} = \emptyset$. 
Proof by Contradiction

- Let $D$ be a decidable language that separates them.
- Assume $A_{\text{no}} \subset D$ and $D \cap A_{\text{yes}} = \emptyset$.
- Let $M_D$ be the TM that decides $D$
Proof by Contradiction

Let $D$ be a decidable language that separates them.

Assume $A_{\text{no}} \subset D$ and $D \cap A_{\text{yes}} = \emptyset$.

Let $M_D$ be the TM that decides $D$.

What does $M_D$ do with input $\langle M_D \rangle$?
Proof by Contradiction

- Let $D$ be a decidable language that separates them.
- Assume $A_{\text{no}} \subset D$ and $D \cap A_{\text{yes}} = \emptyset$.
- Let $M_D$ be the TM that decides $D$.
- What does $M_D$ do with input $\langle M_D \rangle$?
- It must halt. (why?)
Proof by Contradiction

- Let $D$ be a decidable language that separates them.
- Assume $A_{\text{no}} \subset D$ and $D \cap A_{\text{yes}} = \emptyset$.
- Let $M_D$ be the TM that decides $D$.
- What does $M_D$ do with input $\langle M_D \rangle$?
- It must halt. (why?)
- If $M_D$ accepts $\langle M_D \rangle$: 
Proof by Contradiction

- Let $D$ be a decidable language that separates them.
- Assume $A_{\text{no}} \subset D$ and $D \cap A_{\text{yes}} = \emptyset$.
- Let $M_D$ be the TM that decides $D$.
- What does $M_D$ do with input $\langle M_D \rangle$?
- It must halt. (why?)
- If $M_D$ accepts $\langle M_D \rangle$:
  - $\langle M_D \rangle \in A_{\text{yes}}$
Proof by Contradiction

- Let $D$ be a decidable language that separates them.
- Assume $A_{\text{no}} \subset D$ and $D \cap A_{\text{yes}} = \emptyset$.
- Let $M_D$ be the TM that decides $D$
- What does $M_D$ do with input $\langle M_D \rangle$?
- It must halt. (why?)
- If $M_D$ accepts $\langle M_D \rangle$:
  - $\langle M_D \rangle \in A_{\text{yes}}$
  - $\langle M_D \rangle \notin D$
Proof by Contradiction

- Let $D$ be a decidable language that separates them.
- Assume $A_{\text{no}} \subset D$ and $D \cap A_{\text{yes}} = \emptyset$.
- Let $M_D$ be the TM that decides $D$.
- What does $M_D$ do with input $\langle M_D \rangle$?
- It must halt. (why?)
- If $M_D$ accepts $\langle M_D \rangle$:
  - $\langle M_D \rangle \in A_{\text{yes}}$
  - $\langle M_D \rangle \notin D$
  - so $M_D$ rejects $\langle M_D \rangle$. 
Proof by Contradiction

Let $D$ be a decidable language that separates them.
Assume $A_{\text{no}} \subset D$ and $D \cap A_{\text{yes}} = \emptyset$.
Let $M_D$ be the TM that decides $D$

What does $M_D$ do with input $\langle M_D \rangle$?

It must halt. (why?)

If $M_D$ accepts $\langle M_D \rangle$:

$\langle M_D \rangle \in A_{\text{yes}}$
$\langle M_D \rangle \not\in D$
so $M_D$ rejects $\langle M_D \rangle$.

If $M_D$ rejects $\langle M_D \rangle$:
Proof by Contradiction

- Let $D$ be a decidable language that separates them.
- Assume $A_{\text{no}} \subset D$ and $D \cap A_{\text{yes}} = \emptyset$.
- Let $M_D$ be the TM that decides $D$.
- What does $M_D$ do with input $\langle M_D \rangle$?
- It must halt. (why?)
- If $M_D$ accepts $\langle M_D \rangle$:
  - $\langle M_D \rangle \in A_{\text{yes}}$
  - $\langle M_D \rangle \notin D$
  - so $M_D$ rejects $\langle M_D \rangle$.
- If $M_D$ rejects $\langle M_D \rangle$:
  - $\langle M_D \rangle \in A_{\text{no}}$
Proof by Contradiction

- Let $D$ be a decidable language that separates them.
- Assume $A_{\text{no}} \subset D$ and $D \cap A_{\text{yes}} = \emptyset$.
- Let $M_D$ be the TM that decides $D$.
- What does $M_D$ do with input $\langle M_D \rangle$?
- It must halt. (why?)
- If $M_D$ accepts $\langle M_D \rangle$:
  - $\langle M_D \rangle \in A_{\text{yes}}$
  - $\langle M_D \rangle \notin D$
  - so $M_D$ rejects $\langle M_D \rangle$.
- If $M_D$ rejects $\langle M_D \rangle$:
  - $\langle M_D \rangle \in A_{\text{no}}$
  - $\langle M_D \rangle \in D$
Proof by Contradiction

Let $D$ be a decidable language that separates them.
Assume $A_{no} \subset D$ and $D \cap A_{yes} = \emptyset$.
Let $M_D$ be the TM that decides $D$

What does $M_D$ do with input $\langle M_D \rangle$?
It must halt. (why?)

If $M_D$ accepts $\langle M_D \rangle$:
- $\langle M_D \rangle \in A_{yes}$
- $\langle M_D \rangle \notin D$
  so $M_D$ rejects $\langle M_D \rangle$.

If $M_D$ rejects $\langle M_D \rangle$:
- $\langle M_D \rangle \in A_{no}$
- $\langle M_D \rangle \in D$
  so $M_D$ accepts $\langle M_D \rangle$.  ♣
Recursive Inseparability

Define

\[ B_{\text{yes}} = \{ \langle M \rangle | M \text{ is a TM that accepts } \varepsilon \} \]

and

\[ B_{\text{no}} = \{ \langle M \rangle | M \text{ is a TM that halts and rejects } \varepsilon \} \]

**Theorem:** \( B_{\text{yes}} \) and \( B_{\text{no}} \) are recursively inseparable.

Proof by reduction and contradiction.
Recursive Inseparability

**Theorem:**  \( B_{yes} \) and \( B_{no} \) are recursively inseparable.

By reduction and contradiction.

Assume \( B_{yes} \) and \( B_{no} \) can be separated by \( E \), decided by TM \( M_E \).
Recursive Inseparability

**Theorem:** $B_{\text{yes}}$ and $B_{\text{no}}$ are recursively inseparable.

By reduction and contradiction.

- Assume $B_{\text{yes}}$ and $B_{\text{no}}$ can be separated by $E$, decided by TM $M_E$.
- For TM $M$, define $M'$: On any input,
Recursive Inseparability

Theorem: \( B_{\text{yes}} \) and \( B_{\text{no}} \) are recursively inseparable.

By reduction and contradiction.

- Assume \( B_{\text{yes}} \) and \( B_{\text{no}} \) can be separated by \( E \), decided by TM \( M_E \).
- For TM \( M \), define \( M' \): On any input,
  1. run \( M \) on input \( \langle M \rangle \).
Recursive Inseparability

**Theorem:** $B_{\text{yes}}$ and $B_{\text{no}}$ are recursively inseparable.

By reduction and contradiction.

- Assume $B_{\text{yes}}$ and $B_{\text{no}}$ can be separated by $E$, decided by TM $M_E$.
- For TM $M$, define $M'$: On any input,
  1. run $M$ on input $\langle M \rangle$.
  2. if $M$ accepts, accept; if $M$ rejects, reject;
Proof (Concluded)

Define $N$: On input $\langle M \rangle$, 
Proof (Concluded)

Define $N$: On input $\langle M \rangle$,
1. construct description of $M'$. 
Define $N$: On input $\langle M \rangle$,
1. construct description of $M'$.
2. run $M_E$ on $\langle M' \rangle$. 
Proof (Concluded)

- Define $N$: On input $\langle M \rangle$,
  1. construct description of $M'$.
  2. run $M_E$ on $\langle M' \rangle$.
  3. if $M_E$ accepts, accept; if $M_E$ rejects, reject;
Proof (Concluded)

Define $N$: On input $\langle M \rangle$,

1. construct description of $M'$.
2. run $M_E$ on $\langle M' \rangle$.
3. if $M_E$ accepts, accept; if $M_E$ rejects, reject;

Claim:
Proof (Concluded)

- Define $N$: On input $\langle M \rangle$,
  1. construct description of $M'$.
  2. run $M_E$ on $\langle M' \rangle$.
  3. if $M_E$ accepts, accept; if $M_E$ rejects, reject;

- Claim:
  $N$ is a decider. (why?)
Proof (Concluded)

- Define \( N \): On input \( \langle M \rangle \),
  1. construct description of \( M' \).
  2. run \( M_E \) on \( \langle M' \rangle \).
  3. if \( M_E \) accepts, accept; if \( M_E \) rejects, reject;

- Claim:
  - \( N \) is a decider. (why?)
  - So \( N \) decides a language \( D \).
Proof (Concluded)

Define $N$: On input $\langle M \rangle$,
1. construct description of $M'$.
2. run $M_E$ on $\langle M' \rangle$.
3. if $M_E$ accepts, accept; if $M_E$ rejects, reject;

Claim:
- $N$ is a decider. (why?)
- So $N$ decides a language $D$.
- $D$ separates $A_{\text{yes}}$ and $A_{\text{no}}$, contradiction. ♣️