Computational Theory

Computability

Dr Russ Ross

Dixie State University—Computer and Information Technologies

Fall 2016

Adapted from notes by Harry Lewis
Reading: Sipser §3.1.
Objective: Define a computational model that is

- **General-purpose:**
  (as powerful as programming languages)

- **Formally Simple:**
  (we can prove what **cannot** be computed)
The Origins of Computer Science

Alan Mathison Turing

“On Computable Numbers, with an Application to the Entscheidungsproblem” 1936

CF also

- David Hilbert
  “Mathematical Problems” 1900

- Kurt Gödel
  “On Formally Undecidable Propositions . . .” 1931

- Alonzo Church
  “An Unsolvable Problem of Elementary Number Theory” 1936
The Basic Turing Machine

- Head can both read and write, and move in both directions
- Tape has unbounded length
- □ is the blank symbol. All but a finite number of tape squares are blank.
A (deterministic) **Turing Machine (TM)** is a 7-tuple $(Q, \Sigma, \Gamma, \delta, q_0, q_{\text{accept}}, q_{\text{reject}})$, where:

- $Q$ is a finite set of states, containing
  - the **start state** $q_0$
  - the **accept state** $q_{\text{accept}}$
  - the **reject state** $q_{\text{reject}}$ ($\neq q_{\text{accept}}$)

- $\Sigma$ is the **input alphabet**

- $\Gamma$ is the **tape alphabet**
  - Contains $\Sigma$
  - Contains “blank” symbol $\sqcup \in \Gamma - \Sigma$
The transition function

\[ Q \times \Gamma \rightarrow Q \times \Gamma \times \{ L, R \} \]

- \( L \) and \( R \) are “move left” and “move right”
- \( \delta(q, \sigma) = (q', \sigma', R) \)
  - Rewrite \( \sigma \) as \( \sigma' \) in current cell
  - Switch from state \( q \) to state \( q' \)
  - And move right
- \( \delta(q, \sigma) = (q', \sigma', L) \)
  - Same, but move left
  - \textit{Unless} at left end of tape, in which case stay put
Computation of TMs

- **A configuration** is $uqv$, where $q \in Q$, $u, v \in \Gamma^*$.  
  - Tape contents $= uv$ followed by all blanks  
  - State $= q$  
  - Head on first symbol of $v$.  
  - Equivalent to $uqv'$, where $v' = vq$.  

- Start configuration $= q_0w$, where $w$ is input.  

- One step of computation:  
  - $uq\sigma v$ yields $u\sigma'q'v$ if $\delta(q, \sigma) = (q', \sigma', R)$.  
  - $u\tau q\sigma v$ yields $uq'\tau \sigma' v$ if $\delta(q, \sigma) = (q', \sigma', L)$.  
  - $q\sigma v$ yields $q'\sigma'v$ if $\delta(q, \sigma) = (q', \sigma', L)$.  

- If $q \in \{q_{accept}, q_{reject}\}$, computation halts.
TMs and Language Membership

- $M$ **accepts** $w$ if there is a sequence of configurations $C_1, \ldots, C_k$ such that

  1. $C_1 = q_0 w$.
  2. $C_i$ yields $C_{i+1}$ for each $i$.
  3. $C_k$ is an accepting configuration (i.e. state of $M$ is $q_{\text{accept}}$).

- $L(M) = \{ w : M$ accepts $w \}$.

- $L$ is **Turing-recognizable** if $L = L(M)$ for some TM $M$, i.e.

  - $w \in L \Rightarrow M$ halts on $w$ in state $q_{\text{accept}}$.
  - $w \notin L \Rightarrow M$ halts on $w$ in state $q_{\text{reject}}$ OR $M$ never halts (it “loops”).
Decidability, a.k.a. Recursiveness

- $L$ is *(Turing-)decidable* if there is a TM $M$ s.t.
  - $w \in L \Rightarrow M$ halts on $w$ in state $q_{\text{accept}}$.
  - $w \notin L \Rightarrow M$ halts on $w$ in state $q_{\text{reject}}$.

- Other common terminology
  - Recursive = decidable
  - Recursively enumerable (r.e.) = Turing-recognizable
  - Because of alternate characterizations as sets that can be defined via certain systems of recursive (self-referential) equations.
Example

Claim: \( L = \{ a^n b^n c^n : n \geq 0 \} \) is decidable.
Turing Machines

Questions

▶ Does every TM recognize some language?
▶ Does every TM decide some language?
▶ How many Turing-recognizable languages are there?
▶ How many decidable languages are there?
The Church-Turing Thesis

**Reading**: Sipser §3.2, §3.3.
“Computability”

- Defined in terms of Turing machines
- Computable = recursive/decidable (sets, functions, etc.)
- In fact an abstract, universal notion
- Many other computational models yield exactly the same classes of computable sets and functions
- Power of a model = what is computable using the model (extensional equivalence)
- Not programming convenience, speed (for now...), etc.
- All translations between models are constructive
TM Extensions That Do Not Increase Its Power

- TMs with a 2-way infinite tape, unbounded to left and right

\[
\ldots \quad \square \quad a \quad b \quad a \quad a \quad \ldots
\]

**Proof** that TMs with 2-way infinite tapes are no more powerful than the 1-way infinite tape variety.

"Simulation." Convert any 2-way infinite TM into an equivalent 1-way infinite TM with a "two-track tape."

\[
\begin{array}{cccccc}
\cdots & c & b & a & \square & b & a & \square & b & a & a & \cdots \\
-5 & -4 & -3 & -2 & -1 & 0 & 1 & 2 & 3 & 4
\end{array}
\]

Tape of 2-way infinite TM \( M \)

\[
\begin{array}{c}
b \\
\downarrow \\
a
\end{array} = \langle b, a \rangle
\]

Corresponding tape of 1-way infinite TM \( M' \)
Recall the Formal Definition of a TM:

A (deterministic) **Turing Machine (TM)** is a 7-tuple $(Q, \Sigma, \Gamma, \delta, q_0, q_{\text{accept}}, q_{\text{reject}})$, where:

- $Q$ is a finite set of states, containing
  - the **start state** $q_0$
  - the **accept state** $q_{\text{accept}}$
  - the **reject state** $q_{\text{reject}}$ ($\neq q_{\text{accept}}$)
- $\Sigma$ is the **input alphabet**
- $\Gamma$ is the **tape alphabet**
  - Contains $\Sigma$
  - Contains “blank” symbol $\sqcup \in \Gamma - \Sigma$
- $\delta : Q \times \Gamma \rightarrow Q \times \Gamma \times \{L, R\}$ is the **transition function**.
Formalizing the Simulation of 2-way infinite tape TM

Formally, $\Gamma' = (\Gamma \times \Gamma) \cup \{\$\}$. 

$M'$ includes, for every state $q$ of $M$, **two** states:

- $\langle q, 1 \rangle \sim \text{“}q\text{, but we are working on upper track”}$
- $\langle q, 2 \rangle \sim \text{“}q\text{, but we are working on lower track”}$

e.g. If $\delta_M(q, \sigma) = (q', \sigma', L)$ then $\delta_{M'}(\langle q, 1 \rangle, \langle \sigma, \tau \rangle) = (\langle q', 1 \rangle, \langle \sigma', \tau \rangle, R)$. 

Also need transitions for:

- Lower track
- U-turn on hitting endmarker
- Formatting input into “2-tracks”
Describing Turing Machines

Formal Description

- 7-tuple or state diagram
- Most of the course so far

Implementation Description

- Prose description of tape contents, head movements
- This lecture, some of next lecture, assignment 6

High-Level Description

- Does not refer to specific computational model
- Starting next time!
More extensions

- Adding **multiple tapes** does not increase power of TMs

(CoConvention: First tape used for I/O, like standard TM; Second tape is available for scratch work)
Simulation of multiple tapes

- Simulate a $k$-tape TM by a one-tape TM whose tape is split (conceptually) into $2k$ tracks:
  - $k$ tracks for tape symbols
  - $k$ tracks for head position position markers (one in each track)

(Sipser does a different simulation.)
Simulation steps

- To simulate **one move** of the $k$-tape TM:
Simulation steps

- To simulate **one move** of the $k$-tape TM:
  - Start with the head on the left endmarker
  - Scan down the tape, remembering in the finite control the symbols “scanned” by the $k$ heads
  - Scan back up the tape, revising each track in the vicinity of its head marker
  - Return the head to the left endmarker
Note that the “equivalence” in ability to compute functions or decide languages does not mean comparable speed.

- e.g. A standard TM can decide $L = \{w\#w : w \in \Sigma^*\}$ in time $\sim |w|^2$, but there is a linear-time 2-tape decider.
The Church-Turing Thesis

Multiple Tapes

Speed of the simulation

- Note that the “equivalence” in ability to compute functions or decide languages does not mean comparable speed.
  
  e.g. A standard TM can decide $L = \{w\#w : w \in \Sigma^*\}$ in time $\sim |w|^2$, but there is a linear-time 2-tape decider.

- Let $T_M : \Sigma^* \rightarrow \mathbb{N}$ measure the amount of time a decider $M$ uses on an input. That is, $T_M(w)$ is the number of steps TM $M$ takes to halt on input $w$.

- General fact about multitape to single-tape slowdown:

  **Theorem:** If $M$ is a multitape TM that takes time $T(w)$ when run on input $w$, then there is a 1-tape machine $M'$ and a constant $c$ such that $M'$ simulates $M$ and takes at most $cT(w)^2$ steps on input $w$. 
Nondeterministic TMs

- Like TMs, but $\delta : Q \times \Gamma \rightarrow \mathcal{P}(Q \times \Gamma \times \{L, R\})$

- It mainly makes sense to think of NTMs as **recognizers**

$$L(M) = \{w : M \text{ has some accepting computation on input } w\}$$

**Example:** NTM to recognize

$$\{w : w \text{ is a binary notation for a product of two integers } \geq 2\}$$
Nondeterministic TMs

- Like TMs, but \( \delta : Q \times \Gamma \rightarrow \mathcal{P}(Q \times \Gamma \times \{L, R\}) \)
- It mainly makes sense to think of NTMs as **recognizers**

\[
L(M) = \{ w : M \text{ has some accepting computation on input } w \} 
\]

**Example:** NTM to recognize
\( \{ w : w \text{ is a binary notation for a product of two integers } \geq 2 \} \)

1. Write any binary numeral (except 0 or 1) [N.D.]
2. Write \( \sqcup \)
3. Write any binary numeral (except 0 or 1) [N.D.]
4. Multiply
5. Compare product to the input; halt if they are equal, go into an infinite loop if not.
NTMs recognize the same languages as TMs

- Given a NTM $M$, we must construct a TM $M'$ that determines, on input $w$, whether $M$ has an accepting computation on input $w$.
- $M'$ systematically tries
  - all one-step computations
  - all two-step computations
  - all three-step computations
  - ...
Enumerating computations

- There is a bounded number of $k$-step computations, for each $k$.
  (because for each configuration there is only a constant number of “next” configurations in one step)

- Ultimately $M'$ either:
  - discovers an accepting computation of $M$, and accepts itself,
    or
  - searches forever, and does not halt
In More Detail

- Suppose that the maximum number of different transitions for a given \((q, \sigma)\) is \(b\).
- Number those transitions 1, \ldots, \(b\) (or less)
- Any computation of \(k\) steps is determined by a sequence of \(k\) numbers \(\leq b\) (the “nondeterministic choices”).
- How \(M’\) works: 3 tapes
  - #1: Original input to \(M\)
  - #2: Simulated tape of \(M\)
  - #3: 1213 \(\square\) \(\ldots\) Nondeterministic choices for \(M’\)
Simulating one step of $M$

- Each major phase of the simulation by $M'$ is to simulate one finite computation by $M$, using tape #3 to resolve nondeterministic ambiguities.

- Between major phases, $M'$
  - erases tape #2 and copies tape #1 to tape #2
  - Replaces string in $\{1, \ldots, b\}^*$ on tape #3 with the lexicographically next string to generate the next set of nondeterministic choices to follow.

- **Claim:** $L(M') = L(M)$

- **Q:** Slowdown?
Equivalent Formalisms

Many other formalisms for computation are equivalent in power to the TM formalism:

- TMs with 2-dimensional tapes
- Random-access TMs
- General Grammars
- 2-stack PDAs, 2-counter machines
- Church’s $\lambda$-calculus ($\mu$-recursive functions)
- Markov algorithms
- Your favorite high-level programming language (C, Lisp, Java, . . . )
- . . .
General Grammars

- Like context-free grammars, except that if \( u \rightarrow v \) is a rule, then \( u \) may be any string containing a nonterminal.

- So the rule \( AXY \rightarrow AYX \) where \( A, X, Y \in V \), “means” that the two-symbol substring \( XY \) can be replaced by \( YX \) whenever it appears with an \( A \) to its left.
A grammar to generate \( \{ a^n b^n c^n : n \geq 0 \} \).

\[ \Sigma = \{ a, b, c \} \quad V = \{ A, B, C, A', B', C', S \} \]

- \( A, B, C \) are “aliases” for the terminal symbols \( a, b, c \).
- Only a single occurrence of \( A', B', \) or \( C' \) can be in the string being derived.
- It “crawls” from right to left, transforming nonterminal symbols into terminals.
Rules for $a^n b^n c^n$

- $S \rightarrow ABCS$  
  $S \rightarrow C'$  
  $S \rightarrow \varepsilon$

  (Thus $S \Rightarrow^* (ABC')^n C'$ for any $n \geq 0$)

- $CA \rightarrow AC$  
  $BA \rightarrow AB$  
  $CB \rightarrow BC$

  (Any inversions of the proper order can be repaired)

- $CC' \rightarrow C' c$  
  $CC' \rightarrow B' c$

  (The $c$-transformer can crawl to the left, and turn into a $b$-transformer)

- $BB' \rightarrow B' b$  
  $BB' \rightarrow A' b$

- $AA' \rightarrow A' a$  
  $A' \rightarrow \varepsilon$

The only way to get a string of terminals yields one of the form $a^n b^n c^n$. 

Dr Russ Ross (Dixie State University)  
CS 3530  
Fall 2016  
31 / 53
**Theorem:** A language is generated by a grammar if and only if it is Turing-recognizable.

**Proof:**

1. $L$ is generated by a grammar $\Rightarrow L$ is Turing-recognizable

   **Pf:** Let $L = L(G)$, $G$ a grammar. To construct a NTM $M$ such that $L(M) = L$, construct $M$ so that
   $M$ nondeterministically carries out a derivation
   $S = w_0 \Rightarrow G w_1 \Rightarrow G w_2 \Rightarrow G \cdots$, checking each step to see if
   $w_i = w$. 

$L$ Turing-recognizable $\Rightarrow$ $L$ is generated by a grammar.

2. $L$ is recognized by a TM $M$ $\Rightarrow$ $L$ is generated by a grammar $G$

**Pf:** Without loss of generality, we assume that if $M$ halts having started on input $w$, right before halting it erases its tape. $G$ will simulate a **backwards computation** by $M$. The intermediate strings will be configurations $uq\sigma v$. 
Rules of the Grammar

- $S \rightarrow \$ q_{\text{accept}}$

- If $\delta(q, \sigma) = (q', \sigma', R)$, then $G$ has
  
  $\sigma' q' \rightarrow q \sigma$

  $\sigma' q' \$ \rightarrow q \$, if $\sigma = \$\$

- If $\delta(q, \sigma) = (q', \sigma', L)$, then $G$ has
  
  $q' \tau \sigma' \rightarrow \tau q \sigma$ for each $\tau \in \Sigma$

  $q' \tau \$ \rightarrow \tau q \sigma \$, if $\sigma' = \$$

  $\$ q' \sigma' \rightarrow \$ q \sigma$

- Finally, $\$ \rightarrow \varepsilon$ and, if $q_0$ is the start state of the TM, $q_0 \rightarrow \varepsilon$
A 2-counter machine (2-CM) has:

- A finite-state control
- Two counters, i.e., $C_1$ and $C_2$, which are registers containing integers $\geq 0$ with only 3 operations:
  - Add 1 to $C_1/C_2$
  - Subtract 1 from $C_1/C_2$
  - Is $C_1/C_2 = 0$?

**Theorem:** For any TM, there is an equivalent 2-CM, in the sense that if you start the 2-CM with an encoding of the TM tape in its counters it will eventually halt with an encoding of what the TM computes.
Simulating a TM tape with 2 pushdown stores:
Split the tape at the head position into two stacks

Moving TM head to left  $\equiv$  Pop from stack #1
                        $\quad$ Push onto stack #2

Moving TM head to right $\equiv$  Pop from stack #2
                        $\quad$ Push onto stack #1

Change scanned symbol $\equiv$  Change top of stack #1

(So 2-PDSs are as powerful as TMs)
Simulating One Stack with Two Counters: Think of the stack as a number in a base $= |\Sigma| + 1$

[Assume $\leq 9$ stack symbols]

- Pop the stack $\equiv$ Divide by 10 and discard the remainder
- Push $a_9$ $\equiv$ Multiply by 10 and add 9
- Is stack top $= a_3$? $\equiv$ Is counter mod 10 $= 3$?

→ All of these can be calculated using a second counter.
Simulating Four Counters With Two:

\[(p, q, r, s) \rightarrow 2^p 3^q 5^r 7^s\]

Add 1 to \(C1\)
\[\equiv \quad p \leftarrow p + 1\]
\[\equiv \quad \text{Double } C1'\]

Is \(C3 \neq 0\)?
\[\equiv \quad r \neq 0?\]
\[\equiv \quad \text{Does } 5 \text{ divide } C1' \text{ evenly?}\]

Subtract 1 from \(s\)
\[\equiv \quad \text{Divide } C1' \text{ by } 7\]
The equivalence of each to the others is a mathematical *theorem*. That these *formal models* of algorithms capture our *intuitive notion* of algorithms is the **Church-Turing Thesis**.

(Church’s thesis = partial recursive functions, Turing’s thesis = Turing machines)

This is an extramathematical proposition, not subject to formal proof.
Reading: Sipser §4.1.
**Def:** A TM $M$ **enumerates** a language $L$ if $M$, when started from a blank tape, runs forever and “emits” all and only the strings in $L$. (For example, by writing the string on a special tape and passing through a designated state.)
Recognizable $\equiv$ enumerable

**Theorem:** $L$ is Turing-recognizable iff $L$ is enumerated by some TM.

**Proof:**

$(\Rightarrow)$ Suppose $L(M) = L$. We want to construct a TM $M'$ that enumerates $L$.

*M' dovetails* all of the computations by $M$:

1. Do 1 step of $M$'s computation on $w_0$
2. Do 2 steps of $M$ on $w_0$ and $w_1$
3. Do 3 steps on each of $w_0$, $w_1$, $w_2$

where $w_0, w_1, \ldots$ = lexicographic enumeration of $\Sigma^*$.

Outputting any strings $w_i$ whose computations have accepted.
(⇐) Conversely, suppose $M$ enumerates $L$. We want to show that $L$ is RE.

Given $w$, run $M$ on the blank tape. Every time $M$ passes through state $q$ (the “enumeration state”) pause to see if $w$ is on the output tape and halt if it is.

The language **recognized** by this algorithm is exactly the language **enumerated** by $M$. QED.

- The Turing-decidable sets are usually called **recursive** because they can be computed using certain systems of recursive equations, rather than via TMs.

- The Turing-recognizable sets are usually called **recursively enumerable**, i.e., “computably enumerable.”
Enumerable in order $\equiv$ decidability

**Theorem:** $L$ is decidable iff $L$ is enumerable in lexicographic order.

(lexicographic order has shorter strings before longer, and alphabetic order among strings of the same length)

Proof of $\Rightarrow$: If $L$ is decidable, then to enumerate $L$ in order, generate all of $\Sigma^*$ in order and test each string for membership in $L$, enumerating those that are members.

Almost proof of $\Leftarrow$: to test if $w \in L$, enumerate $L$ and wait until either $w$ or a lexically later string is enumerated. ????
Recall that a language $L \subseteq \Sigma^*$ is decidable if there is a TM that always halts when started on an input in $\Sigma^*$, in either $q_{\text{accept}}$ if $w \in L$ or $q_{\text{reject}}$ if $w \notin L$.

**Proposition:** Every regular language is decidable.

**Proof:** (By “coding” a DFA as a TM.)
Asking questions about arbitrary finite automata

- **Q:** What if the DFA $D$ is part of the input? That is, can we design a single TM that, given two inputs, $D$ and $w$, decides whether $D$ accepts $w$?

  - The TM needs to use a fixed alphabet & state set for all inputs $D, w$.

**Q:** How to represent $D = (Q, \Sigma_D, \delta, q_0, F)$ and $w$? List each component of the 5-tuple, separated by |’s.

  - Represent elements of $Q$ as binary strings over $\{0, 1\}$, seperated by ,’s.
  - Represent elements of $\Sigma_D$ as binary strings over $\{0, 1\}$, seperated by ,’s.
  - Represent $\delta : Q \times \Sigma_D \rightarrow Q$ as a sequence of triples $(q, \sigma, q')$, separated by ,’s, etc.

We denote the encoding of $D$ and $w$ as $\langle D, w \rangle$. 
A “Universal” algorithm for deciding regular languages

**Proposition:** \( A_{\text{DFA}} = \{ \langle D, w \rangle : D \text{ a DFA that accepts } w \} \) is decidable.

**Proof sketch:**

- First check that input is of proper form.
- Then simulate \( D \) on \( w \). Implementation on a multitape TM:
  - Tape 2: String \( w \) with head at current position (or to be precise, its representation).
  - Tape 3: Current state \( q \) of \( D \) (i.e., its representation).

- Could work with other encodings, e.g., transition function as a matrix rather than list of triples.
Representation independence

- **General point:** Notions of computability (e.g. decidability and recognizability) are independent of data representation.
  - A TM can convert any reasonable encoding to any other reasonable encoding.
  - We will use \langle \cdot \rangle to mean “any reasonable encoding”.
  - We will revisit representation issues when we discuss computational **speed**.
  - For the moment we are interested only in whether problems are decidable, undecidable, recognizable, etc., so we can be content knowing that there is **some** representation on which an algorithm could work.
Describing Turing Machines

**Formal Description**

- 7-tuple or state diagram
- Most of the course so far

**Implementation Description**

- Prose description of tape contents, head movements
- Previous lecture and today’s lecture so far

**High-Level Description**

- Does not refer to specific computational model, data representation
- From now on!
More Decidable Problems

- $\{\langle R, w \rangle : R$ is a regular expression that generates $w \}$.
- $\{\langle X \rangle : X$ is a DFA/NFA/RE such that $L(X) = \emptyset \}$.
- $\{\langle X \rangle : X$ is a DFA/NFA/RE such that $|L(X)| = \infty \}$.
- $\{\langle M, w \rangle : M$ is a PDA that accepts $w \}$.
- Every context-free language.
A Universal Turing machine

**Theorem:** There is a Turing machine $U$, such that when $U$ is given $\langle M, w \rangle$ for any TM $M$ and $w$, $U$ produces the same result (accept/reject/loop) as running $M$ on $w$.

**Proof:** Initially,
- First tape contains $\langle M \rangle$, including in particular its transition function $\delta_M$.
- Second tape contains $\langle w \rangle$.
- Third tape contains $\langle q_{start} \rangle$.
- Simulate steps of $M$ by multiple steps of $U$.

(Brief return to implementation description.)

$\Rightarrow$ Turing machines can be “programmed”.
Consequences of the existence of Universal Turing Machines

▶ **Corollary:** $A_{TM} = \{ \langle M, w \rangle : M \text{ accepts } w \}$ is Turing-recognizable (r.e.).

▶ **Corollary:** $HALT_{TM} = \{ \langle M, w \rangle : M \text{ eventually halts on } w \}$ (“The Halting Problem”) is Turing-recognizable.

▶ **Corollary:** “The Turing Machines that halt on some input are an r.e. set” (What does this mean?)

▶ **Q:** What about $\{ \langle M, w, n \rangle : M \text{ halts on } w \text{ in at most } n \text{ steps} \}$?

▶ **Q:** Are these sets decidable?

▶ **Q:** Are there undecidable languages?
Three basic facts on decidability vs. recognizability

1. If $L$ is recursive, then $L$ is r.e.
   
   **Proof:**
Three basic facts on decidability vs. recognizability

1. If $L$ is recursive, then $L$ is r.e.

   **Proof:**
   If $M$ decides $L$, then a machine can recognize $L$ by running $M$, and then going into an infinite loop if $M$ would have halted in the $q_{\text{reject}}$ state.

2. If $L$ is recursive then so is $\overline{L}$.

   **Proof:**
Three basic facts on decidability vs. recognizability

1. If $L$ is recursive, then $L$ is r.e.
   
   **Proof:**
   
   If $M$ decides $L$, then a machine can recognize $L$ by running $M$, and then going into an infinite loop if $M$ would have halted in the $q_{\text{reject}}$ state.

2. If $L$ is recursive then so is $\overline{L}$.
   
   **Proof:**
   
   A machine can decide $\overline{L}$ by running $M$ and then giving a “no” answer when $M$ would give “yes” and vice versa.

3. $L$ is recursive if and only if both $L$ and $\overline{L}$ are r.e.
   
   **Proof:**
   
   ...