Lecture 9:
More on Turing Machines:
The Church-Turing Thesis, Recognizability, Decidability
Turing Machine

In each step:
- Reads a symbol
- Writes a symbol
- Changes state
- Moves Left or Right

INPUT  PUT

INFINITE REWRITABLE TAPE
Definition: A Turing Machine is a 7-tuple $T = (Q, \Sigma, \Gamma, \delta, q_0, q_{\text{accept}}, q_{\text{reject}})$, where:

- $Q$ is a finite set of states
- $\Sigma$ is the input alphabet, where $\square \notin \Sigma$
- $\Gamma$ is the tape alphabet, where $\square \in \Gamma$ and $\Sigma \subseteq \Gamma$
- $\delta : Q \times \Gamma \rightarrow Q \times \Gamma \times \{L, R\}$
- $q_0 \in Q$ is the start state
- $q_{\text{accept}} \in Q$ is the accept state
- $q_{\text{reject}} \in Q$ is the reject state, and $q_{\text{reject}} \neq q_{\text{accept}}$

$\square = \text{“blank”}$
Turing Machine Configurations

corresponds to the configuration:

11010\textcolor{yellow}{q_7}0011100 \in (Q \cup \Gamma)^*
Defining Acceptance and Rejection for TMs

Let $C_1$ and $C_2$ be configurations of $M$

**Definition.** $C_1$ *yields* $C_2$ if $M$ is in configuration $C_2$ after running $M$ in configuration $C_1$ for one step.

$L(M) :=$ set of strings $M$ accepts

Let $w \in \Sigma^*$ and $M$ be a Turing machine

$M$ *accepts* $w$ if there are configs $C_0, C_1, \ldots, C_k$, s.t.

- $C_0 = q_0 w$ (*the initial configuration*)
- $C_i$ yields $C_{i+1}$ for $i = 0, \ldots, k-1$, and
- $C_k$ contains the accept state $q_{\text{accept}}$
A TM $M$ **recognizes** a language $L$ if $M$ **accepts** exactly those strings in $L$

A language $L$ is **recognizable** (a.k.a. **recursively enumerable**) if some TM **recognizes** $L$

A TM $M$ **decides** a language $L$ if $M$ **accepts** all strings in $L$ and **rejects** all strings not in $L$

A language $L$ is **decidable** (a.k.a. **recursive**) if some TM **decides** $L$
A Turing machine for deciding \{ 0^{2^n} | n \geq 0 \}

**Turing Machine PSEUDOCODE:**

1. Sweep from left to right, x-out every other 0
2. If in step 1, the tape had only one 0, accept
3. If in step 1, the tape had an odd number of 0’s (at least 3), reject
4. Move the head left to the first input symbol.
5. Go to step 1.

*Why does this work?*

**Observation:** Every time we return to step 1, the number of 0’s on the tape has been halved.
\{ 0^{2n} \mid n \geq 0 \}

**Step 1**
- \( 0 \rightarrow \square, R \)
- \( 0 \rightarrow x, R \)
- \( x \rightarrow x, R \)
- \( x \rightarrow x, L \)
- \( 0 \rightarrow 0, L \)

**Step 2**
- \( x \rightarrow x, R \)
- \( 0 \rightarrow \square, R \)
- \( \square \rightarrow \square, R \)
- \( \square \rightarrow \square, L \)
- \( 0 \rightarrow x, R \)

**Step 3**
- \( \square \rightarrow \square, R \)
- \( q_{\text{reject}} \)
- \( q_{\text{accept}} \)

**Step 4**
- \( x \rightarrow x, R \)
- \( q_2 \)
- \( q_3 \)
- \( q_4 \)

- **even 0's**
- **odd 0's**
\{0^{2n} \mid n \geq 0\}

\begin{align*}
q_00000 \\
\square q_1000 \\
\square xq_300 \\
\square x0q_40 \\
\square x0xq_3 \\
\square x0q_2x \\
\square xq_20x \\
\square q_2x0x \\
q_2\square x0x \\
q_2\square x0x \\
\square q_1x0x \\
\square xq_10x \\
\square \ldots 
\end{align*}
MULT = \{a^i b^j c^k \mid k = i \cdot j, \text{ and } i, j, k \geq 1\}

TURING MACHINE PSEUDOCODE:

1. If the input doesn’t match $a^*b^*c^*$, reject.
2. Move the head back to the leftmost symbol.
3. Cross off one $a$, scan to the right until see $b$. Sweep between $b$’s and $c$’s, crossing off one of each until all $b$’s are crossed off. If all $c$’s get crossed off while doing this, reject.
4. Uncross all the $b$’s. If there is some $a$ left, then repeat stage 3. If all $a$’s are crossed off, Check if all $c$’s are crossed off. If yes, then accept, else reject.
\[ \text{MULT} = \{ a^i b^j c^k \mid k = i \times j, \text{ and } i, j, k \geq 1 \} \]

Check matches \(a^*b^*c^*\)

Cross off an a

Cross off one c for each b

“Uncross” the b’s

Repeat the crossing, until all a’s crossed (or reject early)

Accept
Turing Machines are Robust!

Many variants and models can be defined. As long as your new model only reads and writes finitely many symbols in each step, It doesn’t matter!

A good ole TM can still simulate it!
Multitape Turing Machines

\[ \delta : Q \times \Gamma^k \rightarrow Q \times \Gamma^k \times \{L,R\}^k \]
Theorem: Every Multitape Turing Machine can be transformed into a single tape Turing Machine.
Theorem: Every Multitape Turing Machine can be transformed into a single tape Turing Machine
Theorem: Every Multitape Turing Machine can be transformed into a single tape Turing Machine

Finite State Control
Theorem: Every Multitape Turing Machine can be transformed into a single tape Turing Machine.
Theorem: Every Multitape Turing Machine can be transformed into a single tape Turing Machine.
Nondeterministic Turing Machines

Have multiple transitions for a state, symbol pair

Theorem: Every nondeterministic Turing machine $N$ can be transformed into a Turing Machine $M$ that accepts precisely the same strings as $N$.

Proof Idea (more details in Sipser p.178-179)
Pick a natural ordering on the strings in $(Q \cup \Gamma \cup \#)^*$

$M(w)$: For all strings $D \in (Q \cup \Gamma \cup \#)^*$ in the ordering,
Check if $D = C_0 \# \cdots \# C_k$ where $C_0, \ldots, C_k$ is an accepting computation history for $N$ on $w$.
If so, accept.
Fact: We can encode Turing Machines as bit strings

\[ 0^n10^m10^k10^s10^t10^r10^u1 \ldots \]

- n states
- m tape symbols (first k are input symbols)
- start state
- accept state
- reject state
- blank symbol

\[
\begin{align*}
( (p, i), (q, j, L) ) &= 0^p10^i10^q10^j101 \\
( (p, i), (q, j, R) ) &= 0^p10^i10^q10^j1001
\end{align*}
\]
Similarly, we can encode DFAs and NFAs as *bit strings*, and \(w \in \Sigma^*\) as *bit strings*.

For \(x \in \Sigma^*\) define \(b_\Sigma(x)\) to be its binary encoding.

For \(x, y \in \Sigma^*\), define the *pair of \(x\) and \(y\)* as

\[
(x, y) := 0|b_\Sigma(x)|1 b_\Sigma(x) b_\Sigma(y)
\]

Then we define the following languages over \(\{0,1\}\):

\[
A_{DFA} = \{ (B, w) \mid B \text{ encodes a DFA over some } \Sigma, \text{ and } B \text{ accepts } w \in \Sigma^* \}
\]

\[
A_{NFA} = \{ (B, w) \mid B \text{ encodes an NFA, } B \text{ accepts } w \}
\]

\[
A_{TM} = \{ (M, w) \mid M \text{ encodes a TM, } M \text{ accepts } w \}
\]
\[ A_{TM} = \{ (M, w) \mid M \text{ encodes a TM over some } \Sigma, \]
\[ w \text{ encodes a string over } \Sigma \]
\[ \text{and } M \text{ accepts } w \} \]

**Technical Note:**

We assume an decoding of pairs, TMs, and strings so that *every* binary string decodes to *some* pair \((M, w)\).

If \( z \in \{0,1\}^* \) doesn’t decode to \((M, w)\) in the usual way, then we *define* that \( z \) decodes to the pair \((D, \epsilon)\) where \( D \) is a “dummy” TM that accepts nothing.

\[ \neg A_{TM} = \{ z \mid z \text{ decodes to } (M, w) \text{ and } M \text{ does not accept } w \} \]
**Theorem:** There is a Turing machine $U$ which takes as input:
- the code of an arbitrary TM $M$
- and an input string $w$
such that $U$ accepts $(M, w) \iff M$ accepts $w$.

This is a *fundamental* property of TMs: There is a Turing Machine that can run arbitrary Turing Machine code!

Note that DFAs/NFAs do *not* have this property. That is, $A_{\text{DFA}}$ and $A_{\text{NFA}}$ are not regular.
\[ A_{\text{DFA}} = \{ (D, w) \mid D \text{ is a DFA that accepts string } w \} \]

**Theorem:** \( A_{\text{DFA}} \) is decidable

**Proof:** A DFA is a special case of a TM. Run the universal \( U \) on \( (D, w) \) and output its answer.

\[ A_{\text{NFA}} = \{ (N, w) \mid N \text{ is an NFA that accepts string } w \} \]

**Theorem:** \( A_{\text{NFA}} \) is decidable. (Why?)

\[ A_{\text{TM}} = \{ (M, w) \mid M \text{ is a TM that accepts string } w \} \]

**Theorem:** \( A_{\text{TM}} \) is recognizable
The Church-Turing Thesis

Everyone’s Intuitive Notion of Algorithms = Turing Machines

*This is not a theorem – it is a falsifiable scientific hypothesis.*

And it has been *thoroughly* tested!
Thm: There are \textit{unrecognizable} languages

Assuming the Church-Turing Thesis, this means there are problems that \textit{no} computing device can ever solve!

We will prove there is \textit{no} \textit{onto} function from the set of all Turing Machines to the set of all languages over \{0,1\}. (But the proof will work for any \textit{finite} \(\Sigma\))

Therefore, the function mapping every TM M to its language L(M), \textit{fails to cover all possible languages}
“There are more problems to solve than there are programs to solve them.”

Turing Machines

Languages over \{0,1\}
Let $L$ be any set and $2^L$ be the power set of $L$

**Theorem:** There is *no* onto function from $L$ to $2^L$

**Proof:** Let $f : L \rightarrow 2^L$ be arbitrary

Define $S = \{ x \in L \mid x \not\in f(x) \} \in 2^L$

For all $x \in L$,

- If $x \in S$ then $x \not\in f(x)$ [by definition of $S$]
- If $x \not\in S$ then $x \in f(x)$

In either case, we have $f(x) \neq S$: the element $x$ is in *exactly one* of the sets!

Therefore $f$ is *not* onto!
What does this mean?

No function from \( L \) to \( 2^L \) can “cover” all the elements in \( 2^L \)

No matter what the set \( L \) is, the power set \( 2^L \) *always* has strictly larger cardinality than \( L \) (and all subsets of \( L \))
Thm: There are \textit{unrecognizable} languages

\textbf{Proof:} Suppose all languages are recognizable. Then for all \( L \), there’s a TM \( M \) that recognizes \( L \). Thus the function \( R: \{ \text{Turing Machines} \} \to \{ \text{Languages} \} \) defined by \( M \mapsto L(M) \) is an onto function.

- \{\text{Turing Machines}\} \quad \{\text{Languages over \{0,1\}}\}
- \{0,1\}^* \quad \{\text{All possible subsets of \{0,1\}^*}\}
- \text{Set } T \quad \text{Set of all subsets of } S: \ 2^T

But we showed there is \textit{no} onto function from \{\text{Turing Machines}\} \subseteq T to its power set \( 2^T \). Contradiction!