CS 154

The Church-Turing Thesis, Recognizability, Decidability, and Diagonalization
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 \( q_{\text{reject}} \neq q_{\text{accept}} \)
Turing Machine Configurations

\[ 11010q_700110 \in (Q \cup \Gamma)^* \]
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

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

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

- $C_0 = q_0w$ [the initial configuration]
- $C_i$ yields $C_{i+1}$ for $i = 0$, ..., $k-1$, 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, cross 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, reject
4. Move the head back to the first input symbol.
5. Go to step 1.

Why does this work?
Idea: Every time we return to stage 1, the number of 0’s on the tape has been halved.
\{ 0^{2^n} \mid n \geq 0 \}

Step 1

Step 2

Step 3

Step 4

q_0 \rightarrow q_1 \rightarrow q_2 \rightarrow q_3 \rightarrow q_4

\{ \text{reject} \} \rightarrow \{ \text{accept} \}

0 \rightarrow 0, R

x \rightarrow x, R

\square \rightarrow \square, R

\square \rightarrow \square, L

\square \rightarrow \square, R
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.
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: $L$ is decidable iff both $L$ and $\overline{L}$ are recognizable.
Recall: Given \( L \subseteq \Sigma^* \), define \( \overline{L} := \Sigma^* \setminus L \)

Theorem: \( L \) is decidable
iff both \( L \) and \( \overline{L} \) are recognizable

Given:
- a TM \( M_1 \) that recognizes \( L \) and
- a TM \( M_2 \) that recognizes \( \overline{L} \),
we want to build a new machine \( M \) that decides \( L \)

How? Any ideas?

Hint: \( M_1 \) always accepts \( x \), when \( x \) is in \( L \)
\( M_2 \) always accepts \( x \), when \( x \) isn’t in \( L \)
Recall: Given \( L \subseteq \Sigma^* \), define \( \neg L := \Sigma^* \setminus L \)

Theorem: \( L \) is decidable
iff both \( L \) and \( \neg L \) are recognizable

Given: a TM \( M_1 \) that recognizes \( L \) and
a TM \( M_2 \) that recognizes \( \neg L \),
we want to build a new machine \( M \) that decides \( L \)

\( M(x) \): Run \( M_1(x) \) and \( M_2(x) \) on separate tapes.
Alternate between simulating one step of \( M_1 \), and one step of \( M_2 \).
If \( M_1 \) ever accepts, then accept
If \( M_2 \) ever accepts, then reject
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)
Pick a natural ordering on all 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 some 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**

\[
( (p, i), (q, j, L) ) = 0^p10^i10^q10^j10
\]

\[
( (p, i), (q, j, R) ) = 0^p10^i10^q10^j100
\]
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 \( y \)* to be

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

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

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

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

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

Technical Note:

We’ll use 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 \((D, \varepsilon)\) where \(D\) is a “dummy” TM that accepts nothing.

\[ \neg A_{\text{TM}} = \{ z \mid z \text{ decodes to } (M, w) \text{ and } M \text{ does not accept } w \} \]
Universal Turing Machines

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 \textit{fundamental} property of TMs:
There is a Turing Machine that can run arbitrary Turing Machine code!

Note that DFAs/NFAs do \textit{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 = Turing Machines of Algorithms

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

And it has been thoroughly tested!
Thm: There are *unrecognizable* languages

Assuming the Church-Turing Thesis, this means there are problems that *NO* computing device can solve!

We will prove that there is no 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 finite \( \Sigma \))

That is, every mapping from Turing machines to languages *fails to cover* all possible languages
“There are more problems to solve than there are programs to solve them.”

Languages over \{0,1\}

Turing Machines
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:** Assume, for a contradiction, there is an onto function \( f : L \rightarrow 2^L \).

Define \( S = \{ x \in L \mid x \notin f(x) \} \in 2^L \).

If \( f \) is onto, then there is a \( y \in L \) with \( f(y) = S \).

Suppose \( y \in S \). By definition of \( S \), \( y \notin f(y) = S \).

Suppose \( y \notin S \). By definition of \( S \), \( y \in f(y) = S \).

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

Proof: Let $f : L \to 2^L$ be an arbitrary function.

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

For all $x \in L$,

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

In either case, we have $f(x) \neq S$. (Why?) 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 \( L \)
Thm: There are *unrecognizable* languages

Proof: Suppose all languages are recognizable. Then for all L, there’s a Turing machine M for recognizing L. Hence there is an onto R: \{Turing Machines\} \rightarrow \{Languages\}

\{Turing Machines\} \ni \{0,1\}^* \ni \text{Set } M \ni \text{Set of all subsets of } M: 2^M

But there is *no* onto function from \{Turing Machines\} \subseteq M to 2^M. Contradiction!