Generality of Decision Problems

- Decision problems are about existence of algorithms to test membership in a language that is the subset of all problem encodings.

- Because Turing machines decide languages, it is possible to ask the question about whether a TM exists that effects a certain algorithm.

- The fundamental question is whether there is an algorithm for an entire class of problems. We are not generally interested in answering the question for just one specific case.
Solvability/Decidability

- A *problem* is called *solvable* if there is an *algorithm* that will determine a *yes/no* answer for every instance of the problem.

- The corresponding *language* is said to be a *decidable language*.

- That is, there an algorithm that determines *membership* in the *language* of instances for which the answer is yes, and non-membership when the answer is no.
Problem vs. Language: Examples

Problem: Does a given DFA accept a given string?

Language: The set of all pairs: \((M, x)\) where

- \(M\) is a DFA encoded as a string
- \(x\) is an input to \(M\)
Possible DFA Acceptance Language over \{0, 1\}

- Assume the states are encoded as 01, 011, 0111, 01111, ... Prefix with an extra 0 for an accepting state, e.g. 00111.
- Assume the first state mentioned is the initial state.
- Assume the input alphabet is \{0, 1\}.
- Put a 0 before the input symbols as a marker.
- Code every transition as state, input, next state
Example DFA Encoded as a String

001000110010101110110001110100101110001110111010111

001 00 011
001 01 0111
011 00 0111
011 01 001
011 01 0111
0111 00 0111
0111 01 0111
A Decision Problem

- To encode the problem: Does DFA M accept string x:

- Follow the encoding of M with 000 followed by x. (000 acts as a marker indicating there are no more transitions and the input follows.)
Examples of the DFA Acceptance Language

0010001100101011101100011101010010111000111011101011100001 is in the language because 01 is accepted by the encoded DFA.

0010001100101011101100011101010010111000111011101011100000 is not in the language as 0 is not accepted by the encoded DFA.

The DFA acceptance problem is thus represented as a language of encoded DFAs and the strings they accept.

Is the DFA acceptance language regular?
A Regular Expression Acceptance Language

• How would you similarly encode pairs of regular expressions (over \{0, 1\}) with strings that are accepted by the regular expression?

• Think about all the symbols that are involved in regular expressions \{\cup, 0, *, \varepsilon, \emptyset, (, ), 0, 1\}.

• A string over \{0, 1\} must represent the regular expression unambiguously.
A Turing Machine Acceptance Language

- The encoding of a Turing machine is similar to the coding of a DFA. The only difference is that the transitions have additional components:
  - a write symbol
  - a motion symbol

i.e. 5-tuples instead of 3-tuples.
There Are Undecidable Languages

- Every decidable language corresponds to a Turing machine that always halts and gives a yes or no answer, per the Church-Turing thesis.

- The set of TM encodings is countably infinite.

- The set of languages is equivalent to \( P(\Sigma^*) \), the power set of \( \Sigma^* \), which is uncountable.

- So there are languages \( L \) for which there is no Turing machine that decides \( L \).
Languages vs. TM Encodings

Languages
\( P(\Sigma^*) \)

\( \Sigma^* \)

TM Encodings
Meaning of “Count” & “Enumerate”

- A set is countable (or “enumerable”) iff it can be put into one-to-one correspondence with the natural numbers or a subset thereof.

- Examples:
  - The set of all pairs of natural numbers is countable.
  - The set of all subsets of natural numbers is not countable.
  - The set of all finite subsets of natural numbers is countable.
Cantor’s Diagonal Argument

• Suppose the set of all languages can be enumerated.

Supposed enumeration of **all** languages

Enumeration of strings in $\Sigma^*$

$X_0 \ X_1 \ X_2 \ X_3 \ldots$

Have a 1 in row $i$ column $j$ if $x_j \in L_i$. Have a 0 otherwise.
Diagonal Argument

- Indicate $x_j \in L_i$ by 1 in row $i$ column $j$.
- Have 0 there otherwise.

Supposed enumeration of all languages

<table>
<thead>
<tr>
<th></th>
<th>$x_0$</th>
<th>$x_1$</th>
<th>$x_2$</th>
<th>$x_3$</th>
<th>...</th>
</tr>
</thead>
<tbody>
<tr>
<td>$L_0$</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>$L_1$</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>$L_2$</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>$L_3$</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Diagonal Argument

The **flattened flipped diagonal** cannot correspond to any language in the enumeration.

Supposed enumeration of **all** languages
Diagonal Argument

- The flattened flipped diagonal cannot be anywhere in the enumeration. It disagrees with each language in at least one bit.

<table>
<thead>
<tr>
<th></th>
<th>$x_0$</th>
<th>$x_1$</th>
<th>$x_2$</th>
<th>$x_3$</th>
<th>...</th>
</tr>
</thead>
<tbody>
<tr>
<td>$L_0$</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>$L_1$</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>$L_2$</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>$L_3$</td>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Supposed enumeration of all languages

Busted!
Diagonal argument in symbols

From the supposed enumeration of all languages $L_0$, $L_1$, $L_2$, ... and the known enumeration of $\Sigma^*$, $x_0$, $x_1$, $x_2$, ..., we constructed a new “diagonal” language not in the enumeration after all:

$$D = \{x_i \mid x_i \notin L_i\}$$
Diagonal argument in symbols

\[ D = \{ x_i \mid x_i \notin L_i \} \]

If \( D \) were in the enumeration it would be \( L_d \) for some \( d \), i.e.
\[ L_d = D = \{ x_i \mid x_i \notin L_i \} \]

But then what about the corresponding string \( x_d \)? Is it in or out of \( L_d \)?

\[ x_d \in L_d (= D) \text{ iff } x_d \notin L_d. \]

which is a contradiction. The supposition that the languages could be enumerated must be false.
Summary so far, and direction

- The set of all languages cannot be enumerated. There are “too many”.

- The set of all Turing machines *can* be enumerated. Each one can be encoded into a single element of $\Sigma^*$ (the machine’s description).

- Thus there must be *some languages* (many, in fact) for which there is no Turing machine.

- If we accept Turing’s thesis, there are some languages are not effectively computable.
Can’t we simply add D to the list?

- We could add the new diagonal language D to the list, in principle.

- This would yield a new list.

- The diagonal argument can then be repeated on the new list

and so on, until we get tired.
How to “Handle” a TM

- Turing machines exist mathematically, not physically.

- We represent a TM M by its encoding, notated
  \[<M>\]

- For example, \(<M>\) could be a string which is a list of 5-tuples as discussed earlier.
The Sipser Notation \(<M>\) and \(<M, x>\)

- \(<M>\) denotes a string encoding some kind of machine, such as a DFA or TM.

- \(<M, x>\) denotes a string encoding a machine as well as the input to the machine.

- The exact details of the encoding are not usually important, as long as the encoding is “reasonable”, i.e. only encodes information about the structure of the machine and not its behavior.
A Specific Undecidable Language

- Consider language

  \[ K = \{ <M> \mid M \text{ does not recognize } <M> \}. \]

  where by “recognize” we mean that M halts in an accepting state with that input.

- In other words, \( K = \{ <M> \mid <M> \not\in L(M) \} \)

- Look familiar? Recall \( D = \{ x_i \mid x_i \not\in L_i \} \)
A Specific Undecidable Language

• K = \{<M> | <M> \not\in L(M)\}

• **There is no Turing machine that recognizes K, i.e. K is undecidable.**

• **Proof:** Suppose some machine, N, does recognize K, i.e. L(N) = K.

• Then <N> ∈ K iff <N> \not\in L(N) = K, a contradiction.
K as the Fountain of Undecidability

- K is the “fountain” from which all other undecidable languages spring forth, as we shall see.
<M> vs. <M, x>

- In some cases, we deal with an encoding of both a machine M and an input x to the machine.

- This is represented <M, x>.
Sipser’s Notation: A for Accept

- Let *Family* be a family of *machines* (such as DFA, PDA, TM, ...) or *grammars* (such as CFG, CSG, ...), etc.

- $A_{\text{Family}}$ is the language consisting of an encoding of:
  - a member $M$ of *Family*
  - a string $x$ over the alphabet of $M$

such that $M$ recognizes $x$ (or generates $x$ in the case of grammar).
Sipser $\mathcal{A}$ Notation Example

- $A_{DFA} =$
  \[ \{ <M, x> \mid M \text{ is a DFA and } x \in L(M) \} \]

- $A_{TM} =$
  \[ \{ <M, x> \mid M \text{ is a TM and } x \in L(M) \} \]

- $A_{CFG} =$
  \[ \{ <G, x> \mid G \text{ is a CFG and } x \in L(G) \} \]

etc.
Which of these are Decidable?

- $A_{DFA}$ (Deterministic Finite-State Automata)
- $A_{NFA}$ (Non-deterministic Finite-State Automata)
- $A_{PDA}$ (Pushdown Automata)
- $A_{CFG}$ (Context-Free Grammar)
- $A_{REGEX}$ (Regular Expression)
Other problems: $E_{Family}$

- E stands for “Empty”

- e.g. $E_{DFA} = \{<M> \mid M \text{ is a DFA and } L(M) = \emptyset\}$
Which of these are Decidable?

- $E_{DFA}$ (Deterministic Finite-State Automata)
- $E_{NFA}$ (Non-deterministic Finite-State Automata)
- $E_{PDA}$ (Pushdown Automata)
- $E_{CFG}$ (Context-Free Grammar)
- $E_{REGEX}$ (Regular Expression)
Divergence

• Divergence is a fact of life for some families of machines.

• A TM in particular could go on making moves forever on a given input.

• In this case, we say the machine diverges, and the result is undefined. (Sometimes we say the “result” is \( \bot \).)
3 Possible Outcomes

For a Turing Machine $M$ on an input $x$, one of three things can emerge.

- $M$ eventually halts, accepting $x$.
- $M$ eventually halts, rejecting $x$.
- $M$ never halts (diverges).
Decide vs. Recognize: Important Technical Distinction

A machine $M$ "decides" a language $L$ iff

$M$ always halts, and indicates whether its original input is in $L$ or not.
Decide vs. Recognize: Important Technical Distinction

- A machine M **recognizes** a language L iff for each input x:
  
  - **If** $x \in L$ then M **will eventually** halt and indicate acceptance.
  
  - **If** $x \notin L$ then M **will not** indicate acceptance.

  (M **might halt** and reject, or it **might diverge** in this case.)
L(M) notation for a Turing Machine

- Let M be a Turing machine.

- By \( L(M) \) we will mean the language recognized by M.

- **If** M *always halts*, \( L(M) \) is **also** the language decided by M.
Recognizability vs. Decidability

- A language is (Turing-) **recognizable** iff it is recognized by some TM.

- A language is (Turing-) **decidable** iff it is decided by some TM.

- Decidable implies recognizable.

- But is the converse true?
Recognition without Decision is still valuable

- Recognition without decision is analogous to “I can’t say precisely, but I’ll know it when I see it.”
- Examples often occur in searching for a solution to a problem.
- The search may yield a solution, but it might not.
- So mere recognition is better than no information at all.
Specific Search Example

- Consider a conjectured formula in first-order logic.
- In the absence of an algorithm for verifying the conjecture, we might resort to search.
- A naïve search would be to just generate all derivable statements over time, checking to see whether the conjecture comes up.
- A more clever search might “prune” the search space, eliminating possibilities that can never lead to an answer.
Search is not always helpful

• We can prove that in some cases, even search is not going to be effective.

• This is because some languages are not even recognizable.
K is Not Recognizable

We claim the language $K$ defined as

$$K = \{<M> \mid <M> \text{ encodes a TM and } <M> \notin L(M)\}$$

is not recognizable by a Turing Machine.

**Proof:** Suppose $K$ were recognized by some TM, say $N$. In other words, $L(N) = K$.

Then $<N> \in K$ iff $<N> \notin L(N) = K$.

So our supposition that $K$ is recognizable was wrong. $K$ is not recognizable.
Getting rid of noisy language

- “encodes a Turing machine and ...” is something we’d prefer not having to say all the time.

- We can make the convention that every string encodes a Turing machine. The strings that didn’t make sense otherwise can be said to encode the Do-Nothing Turing machine by convention.
K is Not Recognizable

\[ K = \{ <M> \mid <M> \notin L(M) \} \]

is a cleaned up version.
The **complement** of $K$ is recognizable.

The **complement** $K^c = \Sigma^* - K$ is recognizable:

$$K^c = \{<M> \mid <M> \in L(M)\}$$

**Proof:**

- Adapt a **Universal TM** $U$ such that, with input $<M>$, it will simulate $M$ acting on $<M>$.

- If and when $U$ halts, if $M$ would have accepted, then $U$ will accept.
$A_{TM}$ is not decidable

• We proved $K$ is not decidable (not even recognizable).

• If $A_{TM}$ were decidable, we could create an algorithm for deciding $K$ as well:

  With input $<M>$, construct $<M, <M>>$ (the description of $M$ together with an input of its own description) and pass it to the supposed algorithm for $A_{TM}$.
  $<M, <M>> \notin A_{TM}$ iff $M$ does not accept $<M>$
  iff $<M> \in K$.

• Thus an algorithm for $A_{TM}$ can be used to construct an algorithm for $K$, which we previously showed to be impossible.
Proof Diagram for undecidability of $A_{TM}$

$K 
\xrightarrow{<M> \text{ copy}} \xrightarrow{<M>, <M>} \xrightarrow{A_{TM} \text{ hypothetical}} \xrightarrow{<M> \notin L(M)} \xrightarrow{no} \xrightarrow{yes} \xrightarrow{<M> \in L(M)} \xrightarrow{yes} \xrightarrow{no} \xrightarrow{<M> \notin L(K)}$
Note on Sipser’s Proof of $A_{TM}$ Undecidable

- Sipser does not first identify $K$.
- Instead, he constructs $D$ which is similar to $K$ from $A_{TM}$ in the proof.

$$D(<M>) = \begin{cases} 
\text{accept if } M \text{ does not accept } <M> \\
\text{reject if } M \text{ accepts } <M>
\end{cases}$$

- He then observes that $D(<D>)$ yields a contradiction, and notes that this is a diagonalization.
Corollary: $A_{TM}^c$ (the complement of $A_{TM}$) is not recognizable.

- $A_{TM}^c = \{<M, x> | M \text{ diverges on } x\}$

- If $A_{TM}^c$ were recognizable, so would $K$ be recognizable:
  
  $K = \{<M> | M \text{ diverges on } <M>\}$
Proof Diagram for unrecognizability of $\mathbf{A}^c_{\text{TM}}$ (A “no” answer is not needed for recognition.)

$K$

$<M> \in L(K)$

$<M> \notin L(M)$

$\text{copy}$

$<M, <M>>$

$\mathbf{A}^c_{\text{TM}}$

yes
The “Halting Problem”

- \( \text{HALT}_{TM} = \{ <M, w> \mid M \text{ is a TM and } M \text{ halts on input } w \} \)

- \( \text{HALT}_{TM} \) is undecidable.
Proof that $\text{HALT}_{TM}$ is undecidable (Sipser Theorem 5.1)

- Suppose $\text{HALT}_{TM}$ were decidable. That means there is an algorithm for it.

- Use that algorithm to construct an algorithm for $A_{TM}$, something we previously showed to be impossible.
Algorithm for $A_{TM}$, assuming an algorithm for $HALT_{TM}$

Algorithm: With input $<M, w>$, first check if $<M, w> \in HALT_{TM}$ (using the algorithm we assume exists).

- **If** $<M, w> \notin HALT_{TM}$, then **reject** $<M, w>$ as $M$ cannot accept $w$ in this case.

- **If** $<M, w> \in HALT_{TM}$, then simulate $M$ on $w$ (e.g. using a universal TM), and **accept** iff $M$ accepts $w$. 
Proof Diagram

$A_{TM}$

$<M, x>$

$HALT_{TM}$

$<M, x>$

$U$

accept

reject

invoke

yes

no

hypothetical

no
Summary

- An algorithm for $\text{HALT}_{TM}$ could be used to construct an algorithm for $A_{TM}$

  thus

- $\text{HALT}_{TM}$ decidable $\rightarrow A_{TM}$ decidable

  equivalently (contrapositive):

- $A_{TM}$ undecidable $\rightarrow \text{HALT}_{TM}$ undecidable

- Thus $\text{HALT}_{TM}$ undecidable
Complementarity Theorem

- A language $L$ is decidable iff both it and its complement are recognizable.

Proof:
- ($\rightarrow$) If $L$ is decidable, it is recognizable.

Furthermore, $L$’s complement is recognizable by the same machine with accept and reject states interchanged. Hence the complement is decidable.
Complementarity Theorem

• \((\leftarrow)\) Suppose \(L\) and its complement are recognizable.

• Let \(M\) recognize \(L\) and let \(N\) recognize its complement \(\Sigma^* - L\).

• Construct a deterministic TM \(P\) that decides \(L\), as follows: For a given input, \(P\) alternates simulating a step of \(M\) on \(x\) with simulating \(N\) on \(x\). If \(M\) accepts \(x\), then \(P\) accepts \(x\). If \(N\) accepts \(x\), then \(P\) rejects \(x\).

• One of \(M\) or \(N\) will accept \(x\) eventually (by definition of recognition), so \(P\) will decide \(x\). Not both of \(M\) or \(N\) will accept \(x\), because one recognizes the complement of what the other recognizes.
Structure of P in the Proof of the Complementarity Theorem

Note: M and N must run in parallel or interleaved, not sequentially. Why?
Co-recognizability

• A language is called **co-recognizable** iff its complement is recognizable.

• Example: K is co-recognizable, but not recognizable.

• Thus, from the Complementarity Theorem:
  • L decidable iff (L is both recognizable and co-recognizable)
  • (L recognizable but not co-recognizable) implies L and L^c are not decidable.
  • (L co-recognizable but not recognizable) implies L and L^c are not decidable.

• There are languages that are **neither** recognizable nor co-recognizable, as we shall see.
More Standard Terminology

- Recall: A language is (Turing-) **decidable** iff it is decided by some TM.

  Much of the computability literature uses the term

  “**recursive**”

*as a synonym for “**decidable**”.*
Why “recursive”?

- It has to do with the Gödel/Kleene notion of **recursive functions** as a representation of the effectively computable functions.

- Note: Do not read into this anything about the function calling itself, etc. which deals with the way in which the function can be **expressed**.
More Standard Terminology

• Recall: A language is (Turing-) **recognizable** iff it is recognized by some TM:

Most literature uses the term

“**recursively-enumerable**”

(r.e. or RE) instead of “**recognizable**.

The term “**semi-decidable**” is also descriptive for this case.
Why “recursively enumerable”? 

- It means there is a recursive function that enumerates the language. We’ll soon see that this is equivalent to the language being recognizable by a Turing machine.
Enumerability

- The Sipser definition of an Enumerator:

- A Turing machine with a metaphorical “printer” that prints out all the elements of a language over time (possibly with repetitions).

This could be replaced with a linear “output tape”, or *interleaved* with the squares of the work tape.
The Language Enumerated

- An enumerator enumerates a language if, running autonomously, each element of the language will get printed at some step (“in finite time”).

- If the language enumerated is infinite, then the enumerator will never halt.

- If the language enumerated is finite, the enumerator might or might not halt (it could print the same element multiple times).
An Alternate Definition of Enumerating

- A Turing machine that interprets its input as an encoded natural number $<i>$, and

- for each value of $i$, the machine **halts** with an element of $\Sigma^*$ in the output area of its tape. This element is declared to be the $i^{th}$ element $x_i$ of a language $L$.

- The language enumerated is $L = \{x_0, x_1, x_2, \ldots \}$ the set of strings produced by halting computations.

- (If $L$ is finite, there will be repetitions in the sequence.)
Why is the Alternate Definition Equivalent to Sipser’s?

• Given a Sipser enumerator S, we can construct an Alternate enumerator A:
  With input <i>, where i \geq 0, A simulates S up until i+1 strings have been “printed”. Then A outputs the last string printed as x_i.

• Given an alternate enumerator A, we can construct a Sipser enumerator S:
  S simulates calls A on successive arguments: A(<0>), A(<1>), ..., to produce the elements of the set being enumerated.
A function $f : \Sigma^* \rightarrow \Sigma^*$ is *computable* if there is a TM such that if $M$ is started with $x$ on its tape, $M$ will eventually halt with $f(x)$ on its tape.

Computable functions are also called *recursive functions* in the computability literature.
Partial Functions

A partial function is like a function, except that it can be undefined for some or all values of arguments.

So it has the uniqueness property of a function: \( x = y \) implies \( f(x) = f(y) \), but may lack the definedness property, that \( f(x) \) is defined for all \( x \) in the domain.

The same notation is usually used for function and partial function, relying on context to resolve the distinction.

Sometimes we write \( f(x) = \perp \) to designate “\( f(x) \) is undefined”. But be aware that \( \perp \) is no ordinary value.
Computable Partial Function

A *partial function* $f: \Sigma^* \rightarrow \Sigma^*$ is *computable* if there is a TM such that if $M$ is started with $x$ on its tape, $M$ will *eventually halt* with $f(x)$ on its tape, or *diverge* (never halt).

Common notation:

$f(x)\downarrow$ means $f(x)$ is defined.

$f(x)\uparrow$ means $f(x)$ diverges ($f(x) = \perp$).

In general, there is **no computable test for** $f(x) = \perp$.

It is just a notational convenience.

Computable partial functions are also called *partial recursive functions* in the literature.
Characteristic Functions

- The **characteristic function** of a language $L \subseteq \Sigma^*$ is a function $ch_L : \Sigma^* \rightarrow \{0, 1\}$ defined by
  
  $\forall x \in \Sigma^* \quad ch_L(x) = 1$ if $x \in L$
  $0$ otherwise

- Every language has a characteristic function, and every function of this form determines a language.
Observations

• A language is **decidable** iff it has a computable characteristic function.

• i.e. a recursive language has a recursive **characteristic function**.

• A language is **recognizable** iff it has a computable “partial characteristic function”:
  \[ \forall x \in \Sigma^* \ pch_L(x) = 1 \text{ if } x \in L \]
  undefined otherwise